Guide du développeur

AWS AppSync

AWS AppSync: Guide du développeur

Table of Contents

Qu'est-ce que AWS AppSync ?

AWSAppSyncpermet aux développeurs de connecter leurs applications et services à des données et à des événements grâce à des API GraphQL et Pub/Sub sécurisées, sans serveur et hautement performantes. Vous pouvez effectuer les opérations suivantes avec AWS AppSync :

- Accédez à vos données à partir d'une ou plusieurs sources de données à partir d'un seul point de terminaison API GraphQL.
- Combinez plusieurs API GraphQL sources en une seule API GraphQL fusionnée.
- Publiez des mises à jour de données en temps réel pour vos applications.
- Tirez parti de la sécurité, de la surveillance, de la journalisation et du suivi intégrés, avec la mise en cache en option pour une faible latence.
- Payez uniquement pour les demandes d'API et les messages en temps réel qui sont envoyés.

Rubriques

AWSAppSyncfonctionnalités

- Accès aux données et requêtes simplifiés, optimisés par GraphQL
- Serverless WebSockets pour les abonnements GraphQL et les canaux pub/sub
- Mise en cache côté serveur pour rendre les données disponibles dans des caches en mémoire à haut débit pour une faible latence
- JavaScriptet TypeScript aide à la rédaction de la logique métier
- Sécurité d'entreprise avec des API privées pour restreindre l'accès aux API et l'intégration avec AWS WAF
- Contrôles d'autorisation intégrés, avec prise en charge des clés d'API, d'IAM, d'Amazon Cognito, des fournisseurs OpenID Connect et de l'autorisation Lambda pour une logique personnalisée.
- API fusionnées pour prendre en charge les cas d'utilisation fédérés

Pour plus de détails sur chacune de ces fonctionnalités, consultez la section AWSAppSyncFonctionnalités.

Utilisez-vous AWS AppSync pour la première fois ?

Nous recommandons aux nouveaux AWS AppSync utilisateurs de commencer par lire les sections suivantes :

- Si vous ne connaissez pas GraphQL, consultez le. Mise en route : création de votre première API GraphQL
- Si vous créez des applications qui utilisent des API GraphQL, consultez Création d'une application client etthe section called “Données en temps réel”.
- Si vous recherchez des informations sur le résolveur GraphQL, voir ce qui suit :

  JavaScript/TypeScript

  - Tutoriels Resolver () JavaScript
  - Référence du résolveur () JavaScript

  VTL

  - Tutoriels Resolver (VTL)
  - Référence du modèle de mappage du résolveur (VTL)
- Si vous recherchez des AWS AppSync exemples de projets, de mises à jour, etc., consultez le AppSyncblog.

Services connexes

Si vous créez une application Web ou mobile à partir de zéro, pensez à utiliser AWS Amplify. Amplify s'appuie sur AWS AppSync d'autres AWS services pour vous aider à créer des applications Web et mobiles plus robustes et plus puissantes avec moins de travail.

Tarification de AWS AppSync

AWS AppSyncest facturé en fonction de millions de demandes et de mises à jour. La mise en cache entraîne des frais supplémentaires. Pour en savoir plus, consultez PricingAWS AppSync (Tarification).

Voici la liste des exceptions à la AWS AppSync tarification générale :

- La mise en cache des API n'AWS AppSyncest pas éligible à l'AWSoffre gratuite.
- Les demandes ne sont pas facturées pour les échecs d'autorisation et d'authentification.
- Les appels à des méthodes exigeant des clés d'API ne sont pas facturés lorsque des clés d'API sont manquantes ou non valides.

GraphQL et architecture AWS AppSync

> **Note**
>
> Ce guide part du principe que l'utilisateur possède une connaissance pratique du style architectural REST. Nous vous recommandons de consulter cette rubrique ainsi que d'autres rubriques relatives au front-end avant de travailler avec AWS AppSync GraphQL et.

GraphQL est un langage de requête et de manipulation pour les API. GraphQL fournit une syntaxe flexible et intuitive pour décrire les exigences relatives aux données et les interactions. Il permet aux développeurs de demander exactement ce dont ils ont besoin et d'obtenir des résultats prévisibles. Il permet également d'accéder à de nombreuses sources en une seule demande, réduisant ainsi le nombre d'appels réseau et les besoins en bande passante, réduisant ainsi l'autonomie de la batterie et les cycles de processeur consommés par les applications.

Les mises à jour des données sont simplifiées grâce à des mutations, ce qui permet aux développeurs de décrire la manière dont les données doivent changer. GraphQL facilite également la configuration rapide de solutions en temps réel via des abonnements. Toutes ces fonctionnalités combinées, associées à de puissants outils de développement, font de GraphQL un outil essentiel pour gérer les données des applications.

GraphQL est une alternative à REST. L'architecture RESTful est actuellement l'une des solutions les plus populaires pour la communication client-serveur. Il est centré sur le concept selon lequel vos ressources (données) sont exposées par une URL. Ces URL peuvent être utilisées pour accéder aux données et les manipuler par le biais d'opérations CRUD (création, lecture, mise à jour, suppression) sous la forme de méthodes HTTP telles que `GET` `POST`, et. `DELETE` L'avantage de REST est qu'il est relativement simple à apprendre et à mettre en œuvre. Vous pouvez configurer rapidement des API RESTful pour appeler un large éventail de services.

Cependant, la technologie devient de plus en plus complexe. Alors que les applications, les outils et les services commencent à évoluer pour un public mondial, le besoin d'architectures rapides et évolutives est d'une importance capitale. REST présente de nombreuses lacunes lorsqu'il s'agit d'opérations évolutives. Consultez ce cas d'utilisation pour un exemple.

Dans les sections suivantes, nous passerons en revue certains des concepts relatifs aux API RESTful. Nous présenterons ensuite GraphQL et son fonctionnement.

Pour plus d'informations sur GraphQL et les avantages de la migration vers GraphQLAWS, consultez le guide de décision relatif aux implémentations de GraphQL.

Rubriques

Qu'est-ce qu'une API ?

Une interface de programmation d'applications (API) définit les règles que vous devez suivre pour communiquer avec d'autres systèmes logiciels. Les développeurs exposent ou créent des API afin que d'autres applications puissent communiquer avec leurs applications par programmation. Par exemple, l'application de feuille de temps expose une API qui demande le nom complet d'un employé et une plage de dates. Lorsqu'il reçoit ces informations, il traite en interne la feuille de temps de l'employé et renvoie le nombre d'heures travaillées dans cette plage de dates.

Vous pouvez considérer une API Web comme une passerelle entre les clients et les ressources sur le Web.

Clients

Les clients sont des utilisateurs qui souhaitent accéder à des informations depuis le Web. Le client peut être une personne ou un système logiciel utilisant l'API. Par exemple, les développeurs peuvent écrire des programmes qui accèdent aux données météorologiques d'un système météorologique. Vous pouvez également accéder aux mêmes données depuis votre navigateur lorsque vous visitez directement le site Web de la météo.

Ressources

Les ressources sont les informations que les différentes applications fournissent à leurs clients. Les ressources peuvent être des images, des vidéos, du texte, des chiffres ou tout autre type de données. La machine qui fournit la ressource au client est également appelée serveur. Organisations utilisent des API pour partager des ressources et fournir des services Web tout en garantissant la

sécurité, le contrôle et l'authentification. En outre, les API les aident à déterminer quels clients ont accès à des ressources internes spécifiques.

Qu'est-ce que REST ?

À un niveau élevé, le transfert d'état représentatif (REST) est une architecture logicielle qui impose des conditions quant au fonctionnement d'une API. REST a été initialement créé comme ligne directrice pour gérer la communication sur un réseau complexe tel qu'Internet. Vous pouvez utiliser l'architecture basée sur REST pour prendre en charge des communications fiables et performantes à grande échelle. Vous pouvez facilement l'implémenter et le modifier, en apportant de la visibilité et de la portabilité multiplateforme à n'importe quel système d'API.

Les développeurs d'API peuvent concevoir des API à l'aide de différentes architectures. Les API qui suivent le style architectural REST sont appelées API REST. Les services Web qui implémentent l'architecture REST sont appelés services Web RESTful. Le terme API RESTful fait généralement référence aux API Web RESTful. Cependant, vous pouvez utiliser les termes API REST et API RESTful de manière interchangeable.

Voici certains des principes du style architectural REST :

Interface uniforme

L'interface uniforme est essentielle à la conception de tout service Web RESTful. Cela indique que le serveur transfère les informations dans un format standard. La ressource formatée est appelée représentation dans REST. Ce format peut être différent de la représentation interne de la ressource sur l'application serveur. Par exemple, le serveur peut stocker des données sous forme de texte mais les envoyer dans un format de représentation HTML.

L'interface uniforme impose quatre contraintes architecturales :

1. Les demandes doivent identifier les ressources. Pour ce faire, ils utilisent un identifiant de ressource uniforme.
2. Les clients disposent de suffisamment d'informations dans la représentation des ressources pour modifier ou supprimer la ressource s'ils le souhaitent. Le serveur répond à cette condition en envoyant des métadonnées qui décrivent plus en détail la ressource.
3. Les clients reçoivent des informations sur la manière de poursuivre le traitement de la représentation. Le serveur y parvient en envoyant des messages autodescriptifs contenant des métadonnées indiquant comment le client peut les utiliser au mieux.

4. Les clients reçoivent des informations sur toutes les autres ressources connexes dont ils ont besoin pour effectuer une tâche. Le serveur y parvient en envoyant des hyperliens dans la représentation afin que les clients puissent découvrir dynamiquement davantage de ressources.

Apatridie

Dans l'architecture REST, l'apatridie fait référence à une méthode de communication dans laquelle le serveur exécute chaque demande client indépendamment de toutes les demandes précédentes. Les clients peuvent demander des ressources dans n'importe quel ordre, et chaque demande est apatride ou isolée des autres demandes. Cette contrainte de conception de l'API REST implique que le serveur peut parfaitement comprendre et traiter la demande à chaque fois.

Système en couches

Dans une architecture système en couches, le client peut se connecter à d'autres intermédiaires autorisés entre le client et le serveur, et il continuera à recevoir des réponses du serveur. Les serveurs peuvent également transmettre des demandes à d'autres serveurs. Vous pouvez concevoir votre service Web RESTful pour qu'il s'exécute sur plusieurs serveurs dotés de plusieurs niveaux tels que la sécurité, les applications et la logique métier, en travaillant ensemble pour répondre aux demandes des clients. Ces couches restent invisibles pour le client.

Possibilité de mise en cache

Les services Web RESTful prennent en charge la mise en cache, qui consiste à stocker certaines réponses sur le client ou sur un intermédiaire afin d'améliorer le temps de réponse du serveur. Supposons, par exemple, que vous consultiez un site Web dont les images d'en-tête et de pied de page sont communes à chaque page. Chaque fois que vous visitez une nouvelle page Web, le serveur doit renvoyer les mêmes images. Pour éviter cela, le client met en cache ou stocke ces images après la première réponse, puis les utilise directement depuis le cache. Les services Web RESTful contrôlent la mise en cache à l'aide de réponses d'API qui se définissent comme pouvant être mises en cache ou non.

Qu'est-ce qu'une API RESTful ?

L'API RESTful est une interface utilisée par deux systèmes informatiques pour échanger des informations en toute sécurité sur Internet. La plupart des applications métiers doivent communiquer avec d'autres applications internes et tierces pour effectuer diverses tâches. Par exemple, pour générer des fiches de paie mensuelles, votre système de comptes interne doit partager des données

avec le système bancaire de votre client afin d'automatiser la facturation et de communiquer avec une application interne de feuille de temps. Les API RESTful prennent en charge cet échange d'informations car elles respectent des normes de communication logicielles sécurisées, fiables et efficaces.

Comment fonctionnent les API RESTful ?

La fonction de base d'une API RESTful est la même que celle de naviguer sur Internet. Le client contacte le serveur à l'aide de l'API lorsqu'il a besoin d'une ressource. Les développeurs d'API expliquent comment le client doit utiliser l'API REST dans la documentation de l'API de l'application serveur. Voici les étapes générales pour tout appel d'API REST :

1. Le client envoie une demande au serveur. Le client suit la documentation de l'API pour formater la demande d'une manière compréhensible par le serveur.
2. Le serveur authentifie le client et confirme qu'il a le droit de faire cette demande.
3. Le serveur reçoit la demande et la traite en interne.
4. Le serveur renvoie une réponse au client. La réponse contient des informations qui indiquent au client si la demande a été acceptée. La réponse inclut également toute information demandée par le client.

Les détails de la demande et de la réponse à l'API REST varient légèrement en fonction de la manière dont les développeurs d'API conçoivent l'API.

Pourquoi utiliser GraphQL sur REST ?

REST est l'un des styles architecturaux fondamentaux des API Web. Cependant, à mesure que le monde devient de plus en plus interconnecté, la nécessité de développer des applications robustes et évolutives deviendra un problème de plus en plus pressant. Bien que REST soit actuellement la norme du secteur pour la création d'API Web, les implémentations RESTful présentent plusieurs inconvénients récurrents qui ont été identifiés :

1. Demandes de données : à l'aide des API RESTful, vous demandez généralement les données dont vous avez besoin via des points de terminaison. Le problème survient lorsque vous avez des données qui ne sont peut-être pas aussi bien regroupées. Les données dont vous avez besoin se trouvent peut-être derrière plusieurs couches d'abstraction, et le seul moyen de les récupérer est d'utiliser plusieurs points de terminaison, ce qui implique de faire plusieurs demandes pour extraire toutes les données.

2. Extraire et sous-extraction : pour aggraver les problèmes liés aux demandes multiples, les données de chaque point de terminaison sont strictement définies, ce qui signifie que vous retournerez les données définies pour cette API, même si vous ne le vouliez pas techniquement.

   Cela peut entraîner une extraction excessive, ce qui signifie que nos demandes renvoient des données superflues. Supposons, par exemple, que vous demandiez des données sur le personnel de l'entreprise et que vous souhaitiez connaître les noms des employés d'un service donné. Le point de terminaison qui renvoie les données contiendra les noms, mais il peut également contenir d'autres données telles que le titre du poste ou la date de naissance. Comme l'API est fixe, vous ne pouvez pas simplement demander les noms ; le reste des données est fourni avec.

   La situation inverse dans laquelle nous ne renvoyons pas suffisamment de données est appelée sous-extraction. Pour obtenir toutes les données demandées, vous devrez peut-être faire plusieurs demandes au service. Selon la structure des données, vous pourriez être confronté à des requêtes inefficaces entraînant des problèmes tels que le redoutable problème n+1.
3. Des itérations de développement lentes : de nombreux développeurs adaptent leurs API RESTful au flux de leurs applications. Cependant, au fur et à mesure que leurs applications se développent, le front et le backend peuvent nécessiter des modifications importantes. Par conséquent, les API risquent de ne plus s'adapter à la forme des données de manière efficace ou percutante. Cela entraîne des itérations de produit plus lentes en raison de la nécessité de modifier l'API.
4. Performances à grande échelle : en raison de ces problèmes aggravants, l'évolutivité sera affectée dans de nombreux domaines. Les performances du côté de l'application peuvent être affectées car vos demandes renverront trop ou pas assez de données (ce qui se traduira par un plus grand nombre de demandes). Ces deux situations sollicitent inutilement le réseau, ce qui se traduit par de mauvaises performances. Du côté des développeurs, la vitesse de développement peut être réduite car vos API sont fixes et ne correspondent plus aux données qu'ils demandent.

L'argument de vente de GraphQL est de surmonter les inconvénients de REST. Voici quelques-unes des principales solutions proposées par GraphQL aux développeurs :

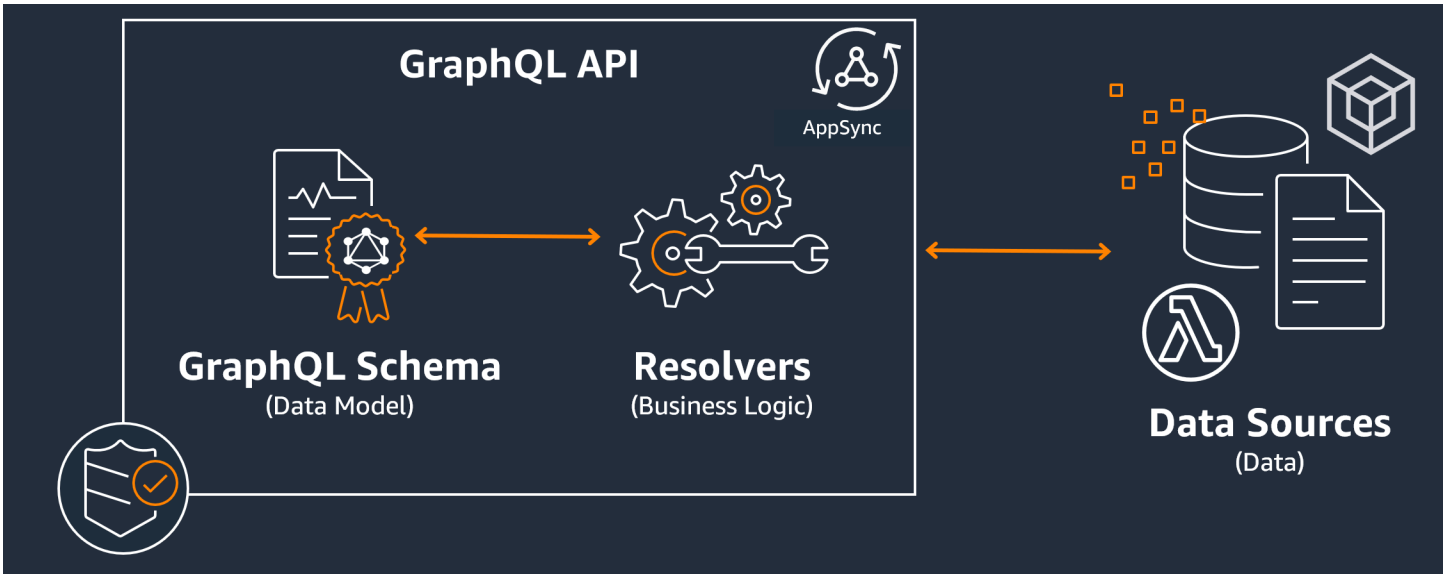
1. Points de terminaison uniques : GraphQL utilise un point de terminaison unique pour interroger les données. Il n'est pas nécessaire de créer plusieurs API pour s'adapter à la forme des données. Cela se traduit par une diminution du nombre de demandes transitant par le réseau.
2. Récupération : GraphQL résout les problèmes récurrents liés à la surextraction et à la sous-extraction en définissant simplement les données dont vous avez besoin. GraphQL vous permet de façonner les données en fonction de vos besoins afin de ne recevoir que ce que vous avez demandé.

3. Abstraction : Les API GraphQL contiennent quelques composants et systèmes qui décrivent les données à l'aide d'une norme indépendante du langage. En d'autres termes, la forme et la structure des données sont normalisées afin que le front-end et le back-end sachent comment elles seront envoyées sur le réseau. Cela permet aux développeurs des deux côtés de travailler avec les systèmes de GraphQL et non de les contourner.
4. Itérations rapides : en raison de la standardisation des données, des modifications peuvent ne pas être nécessaires d'un côté du développement à l'autre. Par exemple, les modifications de présentation du frontend peuvent ne pas entraîner de modifications importantes du backend car GraphQL permet de modifier facilement la spécification des données. Vous pouvez simplement définir ou modifier la forme des données pour répondre aux besoins de l'application au fur et à mesure de sa croissance. Cela se traduit par une réduction des travaux de développement potentiels.

Ce ne sont là que quelques-uns des avantages de GraphQL. Dans les sections suivantes, vous découvrirez comment GraphQL est structuré et quelles sont les propriétés qui en font une alternative unique à REST.

Composants d'une API GraphQL

Une API GraphQL standard est composée d'un schéma unique qui gère la forme des données qui seront interrogées. Votre schéma est lié à une ou plusieurs de vos sources de données, telles qu'une base de données ou une fonction Lambda. Entre les deux se trouvent un ou plusieurs résolveurs qui gèrent la logique métier de vos demandes. Chaque composant joue un rôle important dans votre implémentation de GraphQL. Les sections suivantes présentent ces trois composants et le rôle qu'ils jouent dans le service GraphQL.



Rubriques

Schémas

Le schéma GraphQL est la base d'une API GraphQL. Il sert de modèle qui définit la forme de vos données. Il s'agit également d'un contrat entre votre client et votre serveur qui définit la manière dont vos données seront récupérées et/ou modifiées.

Les schémas GraphQL sont écrits dans le langage SDL (Schema Definition Language). SDL est composé de types et de champs dotés d'une structure établie :

- Types : Les types sont la façon dont GraphQL définit la forme et le comportement des données. GraphQL prend en charge une multitude de types qui seront expliqués plus loin dans cette section. Chaque type défini dans votre schéma contiendra sa propre portée. Le champ d'application comportera un ou plusieurs champs pouvant contenir une valeur ou une logique qui sera utilisée dans votre service GraphQL. Les types remplissent de nombreux rôles différents, les plus courants étant les objets ou les scalaires (types de valeurs primitives).
- Champs : les champs existent dans le cadre d'un type et contiennent la valeur demandée au service GraphQL. Elles sont très similaires aux variables d'autres langages de programmation. La forme des données que vous définissez dans vos champs déterminera la manière dont

les données sont structurées lors d'une opération de demande/réponse. Cela permet aux développeurs de prévoir ce qui sera renvoyé sans savoir comment le backend du service est implémenté.

Pour visualiser à quoi ressemblerait un schéma, examinons le contenu d'un schéma GraphQL simple. Dans le code de production, votre schéma se trouve généralement dans un fichier appelé `schema.graphql` ou`schema.json`. Supposons que nous étudions un projet qui implémente un service GraphQL. Ce projet stocke les données du personnel de l'entreprise, et le `schema.graphql` fichier est utilisé pour récupérer les données sur le personnel et ajouter du nouveau personnel à une base de données. Le code peut ressembler à ceci :

schema.graphql

```
type Person {
   id: ID!
   name: String
   age: Int
}
type Query {
  people: [Person]
}
type Mutation {
  addPerson(id: ID!, name: String, age: Int): Person
}
```

Nous pouvons voir qu'il existe trois types définis dans le schéma :`Person`,`Query`, et`Mutation`. En regardant`Person`, nous pouvons deviner qu'il s'agit du modèle d'une instance d'un employé de l'entreprise, ce qui ferait de ce type un objet. À l'intérieur de son champ d'application`id`, nous voyons`name`, et`age`. Ce sont les champs qui définissent les propriétés d'un`Person`. Cela signifie que notre source de données stocke chacun `Person` d'eux `name` en tant que type `String` scalaire (primitif) et `age` en tant que type `Int` scalaire (primitif). Il `id` agit comme un identifiant spécial et unique pour chacun d'entre eux`Person`. Il s'agit également d'une valeur obligatoire, comme indiqué par le `!` symbole.

Les deux types d'objets suivants se comportent différemment. GraphQL réserve quelques mots clés pour des types d'objets spéciaux qui définissent la manière dont les données seront renseignées dans le schéma. Un `Query` type récupérera les données de la source. Dans notre exemple, notre requête peut récupérer `Person` des objets d'une base de données. Cela peut vous rappeler les `GET`

opérations utilisées dans la terminologie RESTful. A `Mutation` modifiera les données. Dans notre exemple, notre mutation peut ajouter d'autres `Person` objets à la base de données. Cela peut vous rappeler des opérations qui changent d'état, comme `PUT` ou`POST`. Les comportements de tous les types d'objets spéciaux seront expliqués plus loin dans cette section.

Supposons que `Query` dans notre exemple, quelque chose soit extrait de la base de données. Si nous examinons les champs de`Query`, nous voyons un champ appelé`people`. La valeur de son champ est`[Person]`. Cela signifie que nous voulons récupérer une instance de `Person` dans la base de données. Cependant, l'ajout de crochets signifie que nous voulons renvoyer une liste de toutes les `Person` instances et pas seulement une instance spécifique.

Le `Mutation` type est chargé d'effectuer des opérations de changement d'état, telles que la modification des données. Une mutation est chargée d'effectuer une opération de changement d'état sur la source de données. Dans notre exemple, notre mutation contient une opération appelée `addPerson` qui ajoute un nouvel `Person` objet à la base de données. La mutation utilise un `Person` et attend une entrée pour les `age` champs `id`name, et.

À ce stade, vous vous demandez peut-être comment `addPerson` fonctionnent de telles opérations sans implémentation de code, étant donné qu'elles sont censées avoir un certain comportement et ressemblent beaucoup à une fonction avec un nom de fonction et des paramètres. Actuellement, cela ne fonctionne pas car un schéma ne sert que de déclaration. Pour implémenter le comportement de`addPerson`, il faudrait y ajouter un résolveur. Un résolveur est une unité de code exécutée chaque fois que le champ associé (dans ce cas, l'`addPerson`opération) est appelé. Si vous souhaitez utiliser une opération, vous devrez ajouter l'implémentation du résolveur à un moment donné. D'une certaine manière, vous pouvez considérer l'opération du schéma comme la déclaration de fonction et le résolveur comme la définition. Les résolveurs seront expliqués dans une section différente.

Cet exemple montre uniquement les méthodes les plus simples utilisées par un schéma pour manipuler les données. Vous créez des applications complexes, robustes et évolutives en tirant parti des fonctionnalités de GraphQL et. AWS AppSync Dans la section suivante, nous définirons les différents types et comportements de champ que vous pouvez utiliser dans votre schéma.

Types de GraphQL

GraphQL prend en charge de nombreux types différents. Comme vous l'avez vu dans la section précédente, les types définissent la forme ou le comportement de vos données. Ils sont les éléments de base d'un schéma GraphQL.

Les types peuvent être classés en entrées et en sorties. Les entrées sont des types autorisés à être transmis comme argument pour les types d'objets spéciaux (`Query`,, etc.)`Mutation`, tandis que les types de sortie sont strictement utilisés pour stocker et renvoyer des données. Vous trouverez ci-dessous une liste des types et de leurs catégories :

- Objets : un objet contient des champs décrivant une entité. Par exemple, un objet peut être quelque chose comme un `book` avec des champs décrivant ses caractéristiques comme `authorNamepublishingYear`, etc. Ce sont strictement des types de sortie.
- Scalaires : ce sont des types primitifs tels que int, string, etc. Ils sont généralement affectés à des champs. En utilisant le `authorName` champ comme exemple, on pourrait lui attribuer le `String` scalaire pour stocker un nom tel que « John Smith ». Les scalaires peuvent être des types d'entrée et de sortie.
- Entrées : Les entrées vous permettent de transmettre un groupe de champs en tant qu'argument. Leur structure est très similaire à celle des objets, mais ils peuvent être transmis en tant qu'arguments à des objets spéciaux. Les entrées vous permettent de définir des scalaires, des énumérations et d'autres entrées dans son champ d'application. Les entrées ne peuvent être que des types d'entrée.
- Objets spéciaux : les objets spéciaux effectuent des opérations de changement d'état et effectuent l'essentiel du travail. Il existe trois types d'objets spéciaux : requête, mutation et abonnement. Les requêtes récupèrent généralement des données ; les mutations manipulent les données ; les abonnements ouvrent et maintiennent une connexion bidirectionnelle entre les clients et les serveurs pour une communication constante. Les objets spéciaux ne sont ni en entrée ni en sortie étant donné leur fonctionnalité.
- Enums : Les énumérations sont des listes prédéfinies de valeurs légales. Si vous appelez une énumération, ses valeurs ne peuvent être que celles définies dans son champ d'application. Par exemple, si vous aviez une énumération intitulée `trafficLights` représentant une liste de feux de circulation, elle pourrait avoir des valeurs telles que `redLight` et `greenLight` mais non`purpleLight`. Un vrai feu de signalisation n'aura qu'un nombre limité de signaux. Vous pouvez donc utiliser l'énumération pour les définir et les forcer à être les seules valeurs légales lors du référencement`trafficLight`. Les énumérations peuvent être à la fois des types d'entrée et de sortie.
- Unions/interfaces : les syndicats vous permettent de renvoyer un ou plusieurs éléments dans une demande en fonction des données demandées par le client. Par exemple, si vous aviez un `Book` type avec un `title` champ et un `Author` type avec un `name` champ, vous pourriez créer une union entre les deux types. Si votre client souhaitait rechercher dans une base de données l'expression « Jules César », le syndicat pourrait renvoyer Jules César (la pièce de

William Shakespeare) du `Book` `title` et Jules César (l'auteur de Commentarii de Bello Gallico) du. `Author` `name` Les unions ne peuvent être que des types de sortie.

Les interfaces sont des ensembles de champs que les objets doivent implémenter. Cela ressemble un peu aux interfaces des langages de programmation tels que Java où vous devez implémenter les champs définis dans l'interface. Supposons, par exemple, que vous ayez créé une interface appelée `Book` contenant un `title` champ. Supposons que vous ayez créé par la suite un type appelé « `Novel` that implemented »`Book`. Vous `Novel` devrez inclure un `title` champ. Cependant, vous `Novel` pouvez également inclure d'autres champs ne figurant pas dans l'interface, comme `pageCount` par exemple`ISBN`. Les interfaces ne peuvent être que des types de sortie.

Les sections suivantes expliquent le fonctionnement de chaque type dans GraphQL.

Objets

Les objets GraphQL sont le type principal que vous verrez dans le code de production. Dans GraphQL, vous pouvez considérer un objet comme un regroupement de différents champs (similaires aux variables dans d'autres langages), chaque champ étant défini par un type (généralement un scalaire ou un autre objet) pouvant contenir une valeur. Les objets représentent une unité de données qui peut être récupérée/manipulée à partir de l'implémentation de votre service.

Les types d'objets sont déclarés à l'aide du `Type` mot clé. Modifions légèrement notre exemple de schéma :

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}
```

Les types d'objets présentés ici sont `Person` et`Occupation`. Chaque objet possède ses propres champs avec ses propres types. L'une des fonctionnalités de GraphQL est la possibilité de définir d'autres types de champs. Vous pouvez voir que le `occupation` champ `Person` contient un type

d'`Occupation`objet. Nous pouvons établir cette association car GraphQL ne fait que décrire les données et non l'implémentation du service.

Scalaires

Les scalaires sont essentiellement des types primitifs contenant des valeurs. DansAWS AppSync, il existe deux types de scalaires : les scalaires GraphQL par défaut et les scalairesAWS AppSync. Les scalaires sont généralement utilisés pour stocker des valeurs de champs dans des types d'objets. Les types GraphQL par défaut incluent`Int`, `Float` `String``Boolean`, et. `ID` Reprenons l'exemple précédent :

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}
```

En distinguant les `title` champs `name` et, les deux contiennent un `String` scalaire. `Name`pourrait renvoyer une valeur de chaîne comme « `John Smith` » et le titre pourrait renvoyer quelque chose comme « `firefighter` ». Certaines implémentations de GraphQL prennent également en charge les scalaires personnalisés utilisant le `Scalar` mot-clé et implémentant le comportement du type. Cependant, les scalaires personnalisés ne sont AWS AppSync actuellement pas pris en charge. Pour une liste des scalaires, voir Types de scalaires dans. AWS AppSync

Inputs

En raison du concept des types d'entrée et de sortie, certaines restrictions s'appliquent lors de la transmission d'arguments. Les types qui doivent généralement être transmis, en particulier les objets, sont restreints. Vous pouvez utiliser le type de saisie pour contourner cette règle. Les entrées sont des types contenant des scalaires, des énumérations et d'autres types d'entrées.

Les entrées sont définies à l'aide du `input` mot clé :

```
type Person {
  id: ID!
```

```
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}

input personInput {
  id: ID!
  name: String
  age: Int
  occupation: occupationInput
}

input occupationInput {
  title: String
}
```

Comme vous pouvez le constater, nous pouvons avoir des entrées séparées qui imitent le type d'origine. Ces entrées seront souvent utilisées dans le cadre de vos opérations sur le terrain comme suit :

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}

input occupationInput {
  title: String
}

type Mutation {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person
}
```

Notez que nous sommes toujours en train de passer `occupationInput` à la place de `Occupation` pour créer un`Person`.

Il ne s'agit là que d'un des scénarios pour les entrées. Ils n'ont pas nécessairement besoin de copier les objets 1:1, et dans le code de production, vous ne les utiliserez probablement pas de cette manière. Il est recommandé de tirer parti des schémas GraphQL en définissant uniquement ce que vous devez saisir en tant qu'arguments.

De plus, les mêmes entrées peuvent être utilisées dans plusieurs opérations, mais nous vous déconseillons de le faire. Chaque opération doit idéalement contenir sa propre copie unique des entrées au cas où les exigences du schéma changeraient.

Objets spéciaux

GraphQL réserve quelques mots clés à des objets spéciaux qui définissent une partie de la logique métier régissant la manière dont votre schéma récupérera et manipulera les données. Il peut tout au plus y avoir un seul de ces mots clés dans un schéma. Ils servent de points d'entrée pour toutes les données demandées que vos clients exécutent avec votre service GraphQL.

Les objets spéciaux sont également définis à l'aide du `type` mot-clé. Bien qu'ils soient utilisés différemment des types d'objets classiques, leur implémentation est très similaire.

Queries

Les requêtes sont très similaires aux `GET` opérations dans la mesure où elles effectuent une extraction en lecture seule pour obtenir des données de votre source. Dans GraphQL, `Query` définit tous les points d'entrée pour les clients effectuant des requêtes sur votre serveur. Il y en aura toujours un `Query` dans votre implémentation GraphQL.

Voici les types `Query` d'objets modifiés que nous avons utilisés dans notre précédent exemple de schéma :

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}
type Occupation {
  title: String
}
```

```
type Query {
  people: [Person]
}
```

Notre `Query` contient un champ appelé `people` qui renvoie une liste d'`Person`instances à partir de la source de données. Supposons que nous devions modifier le comportement de notre application et que nous devions maintenant renvoyer une liste contenant uniquement les `Occupation` instances dans un but distinct. Nous pourrions simplement l'ajouter à la requête :

```
type Query {
  people: [Person]
  occupations: [Occupation]
}
```

Dans GraphQL, nous pouvons traiter notre requête comme une source unique de requêtes. Comme vous pouvez le constater, cela est potentiellement beaucoup plus simple que les implémentations RESTful qui peuvent utiliser différents points de terminaison pour obtenir la même chose (`.../api/1/peopleet`). `.../api/1/occupations`

En supposant que nous ayons une implémentation de résolveur pour cette requête, nous pouvons maintenant effectuer une requête réelle. Tant que le `Query` type existe, nous devons l'appeler explicitement pour qu'il s'exécute dans le code de l'application. Cela peut être fait à l'aide du `query` mot clé :

```
query getItems {
   people {
      name
   }
   occupations {
      title
   }
}
```

Comme vous pouvez le voir, cette requête est appelée `getItems` et renvoie `people` (une liste d'`Person`objets) et `occupations` (une liste d'`Occupation`objets). Dans`people`, nous renvoyons uniquement le `name` champ de chacun`Person`, tandis que nous renvoyons le `title` champ de chacun`Occupation`. La réponse peut ressembler à ceci :

```
{
  "data": {
```

```
    "people": [
      {
        "name": "John Smith"
      },
      {
        "name": "Andrew Miller"
      },
      .
      .
      .
    ],
    "occupations": [
      {
        "title": "Firefighter"
      },
      {
        "title": "Bookkeeper"
      },
      .
      .
      .
    ]
  }
}
```

L'exemple de réponse montre comment les données suivent la forme de la requête. Chaque entrée récupérée est répertoriée dans le champ d'application du champ. `peopleet` et `occupations` renvoient les éléments sous forme de listes séparées. Bien que cela soit utile, il peut être plus pratique de modifier la requête pour renvoyer une liste des noms et professions des personnes :

```
query getItems {
   people {
      name
      occupation {
        title
      }
}
```

Il s'agit d'une modification légale car notre `Person` type contient un `occupation` champ de type`Occupation`. Une fois répertorié dans le champ de portée de`people`, nous `Person` renvoyons chacun `name` avec le nom associé `Occupation` à`title`. La réponse peut ressembler à ceci :

```
}
  "data": {
    "people": [
      {
        "name": "John Smith",
        "occupation": {
          "title": "Firefighter"
        }
      },
      {
        "name": "Andrew Miller",
        "occupation": {
          "title": "Bookkeeper"
        }
      },
      .
      .
      .
    ]
  }
}
```

Mutations

Les mutations sont similaires aux opérations de changement d'état telles que `PUT` ou`POST`. Ils exécutent une opération d'écriture pour modifier les données de la source, puis récupèrent la réponse. Ils définissent vos points d'entrée pour les demandes de modification de données. Contrairement aux requêtes, une mutation peut être incluse ou non dans le schéma en fonction des besoins du projet. Voici la mutation tirée de l'exemple de schéma :

```
type Mutation {
  addPerson(id: ID!, name: String, age: Int): Person
}
```

Le `addPerson` champ représente un point d'entrée qui ajoute un `Person` à la source de données. `addPerson`est le nom du champ ;`id`,`name`, et `age` sont les paramètres ; et `Person` est le type de retour. Rétrospectivement, le `Person` type :

```
type Person {
  id: ID!
  name: String
```

```
  age: Int
  occupation: Occupation
}
```

Nous avons ajouté le `occupation` champ. Cependant, nous ne pouvons pas définir ce champ sur `Occupation` directement car les objets ne peuvent pas être transmis en tant qu'arguments ; il s'agit uniquement de types de sortie. Nous devrions plutôt transmettre une entrée avec les mêmes champs en tant qu'argument :

```
input occupationInput {
  title: String
}
```

Nous pouvons également facilement mettre à jour notre `addPerson` pour l'inclure en tant que paramètre lors de la création de nouvelles `Person` instances :

```
type Mutation {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person
}
```

Voici le schéma mis à jour :

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}

input occupationInput {
  title: String
}

type Mutation {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person
}
```

Notez que le `title` champ `occupation` sera transmis `occupationInput` pour terminer la création de l'objet au `Person` lieu de l'`Occupation`objet d'origine. En supposant que nous ayons une implémentation de résolveur pour`addPerson`, nous pouvons maintenant effectuer une véritable mutation. Tant que le `Mutation` type existe, nous devons l'appeler explicitement pour qu'il s'exécute dans le code de l'application. Cela peut être fait à l'aide du `mutation` mot clé :

```
mutation createPerson {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput) {
    name
    age
    occupation {
      title
    }
  }
}
```

Cette mutation est appelée`createPerson`, et `addPerson` c'est l'opération. Pour en créer un nouveau`Person`, nous pouvons saisir les arguments pour `id``name`,`age`, et`occupation`. Dans le cadre de`addPerson`, nous pouvons également voir d'autres domaines tels que `name``age`, etc. Voici votre réponse ; ce sont les champs qui seront renvoyés une fois l'`addPerson`opération terminée. Voici la dernière partie de l'exemple :

```
mutation createPerson {
  addPerson(id: "1", name: "Steve Powers", age: "50", occupation: "Miner") {
    id
    name
    age
    occupation {
      title
    }
  }
}
```

En utilisant cette mutation, le résultat pourrait ressembler à ceci :

```
{
  "data": {
    "addPerson": {
      "id": "1",
      "name": "Steve Powers",
      "age": "50",
```

```
      "occupation": {
        "title": "Miner"
      }
    }
  }
}
```

Comme vous pouvez le constater, la réponse a renvoyé les valeurs que nous avions demandées dans le même format que celui défini dans notre mutation. Il est recommandé de renvoyer toutes les valeurs modifiées afin de réduire la confusion et d'éviter d'avoir à effectuer d'autres requêtes à l'avenir. Les mutations vous permettent d'inclure plusieurs opérations dans son champ d'application. Ils seront exécutés séquentiellement dans l'ordre indiqué dans la mutation. Par exemple, si nous créons une autre opération appelée `add0ccupation` qui ajoute des titres de poste à la source de données, nous pouvons l'appeler dans la mutation suivante`addPerson`. `addPerson`sera traité en premier, suivi de`add0ccupation`.

Subscriptions

Les abonnements WebSocketspermettent d'établir une connexion bidirectionnelle durable entre le serveur et ses clients. Généralement, un client s'abonne ou écoute le serveur. Chaque fois que le serveur effectue une modification côté serveur ou exécute un événement, le client abonné reçoit les mises à jour. Ce type de protocole est utile lorsque plusieurs clients sont abonnés et doivent être informés des modifications apportées au serveur ou à d'autres clients. Par exemple, les abonnements peuvent être utilisés pour mettre à jour les flux de réseaux sociaux. Il peut y avoir deux utilisateurs, l'utilisateur A et l'utilisateur B, qui sont tous deux abonnés aux mises à jour automatiques des notifications chaque fois qu'ils reçoivent des messages directs. L'utilisateur A sur le client A pourrait envoyer un message direct à l'utilisateur B sur le client B. Le client de l'utilisateur A enverrait le message direct, qui serait traité par le serveur. Le serveur enverrait ensuite le message direct au compte de l'utilisateur B tout en envoyant une notification automatique au client B.

Voici un exemple `Subscription` que nous pourrions ajouter à l'exemple de schéma :

```
type Subscription {
  personAdded: Person
}
```

Le `personAdded` champ envoie un message aux clients abonnés chaque fois qu'un nouveau message `Person` est ajouté à la source de données. En supposant que nous ayons une implémentation de résolveur pour`personAdded`, nous pouvons désormais utiliser l'abonnement.

Tant que le `Subscription` type existe, nous devons l'appeler explicitement pour qu'il s'exécute dans le code de l'application. Cela peut être fait à l'aide du `subscription` mot clé :

```
subscription personAddedOperation {
  personAdded {
    id
    name
  }
}
```

L'abonnement est appelé`personAddedOperation`, et l'opération l'est`personAdded`. `personAdded`renverra les `name` champs `id` et des nouvelles `Person` instances. En regardant l'exemple de mutation, nous avons ajouté une `Person` en utilisant cette opération :

```
addPerson(id: "1", name: "Steve Powers", age: "50", occupation: "Miner")
```

Si nos clients étaient abonnés aux mises à jour des nouveautés`Person`, ils pourraient le voir après les `addPerson` essais :

```
{
  "data": {
    "personAdded": {
      "id": "1",
      "name": "Steve Powers"
    }
  }
}
```

Vous trouverez ci-dessous un résumé de ce que proposent les abonnements :

Les abonnements sont des canaux bidirectionnels qui permettent au client et au serveur de recevoir des mises à jour rapides mais régulières. Ils utilisent généralement le WebSocket protocole, qui crée des connexions standardisées et sécurisées.

Les abonnements sont souples dans la mesure où ils réduisent les frais de configuration des connexions. Une fois abonné, un client peut simplement continuer à utiliser cet abonnement pendant de longues périodes. Ils utilisent généralement les ressources informatiques de manière efficace en permettant aux développeurs d'adapter la durée de vie de l'abonnement et de configurer les informations qui seront demandées.

En général, les abonnements permettent au client de souscrire plusieurs abonnements à la fois. En ce qui concerneAWS AppSync, les abonnements ne sont utilisés que pour recevoir des mises à jour en temps réel du AWS AppSync service. Ils ne peuvent pas être utilisés pour effectuer des requêtes ou des mutations.

La principale alternative aux abonnements est le sondage, qui envoie des requêtes à intervalles réguliers pour demander des données. Ce processus est généralement moins efficace que les abonnements et met beaucoup de pression à la fois sur le client et sur le backend.

Une chose qui n'a pas été mentionnée dans notre exemple de schéma est le fait que vos types d'objets spéciaux doivent également être définis dans une `schema` racine. Ainsi, lorsque vous exportez un schéma au AWS AppSync format, il peut ressembler à ceci :

schema.graphql

```
schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}

.
.
.

type Query {
  # code goes here
}
type Mutation {
  # code goes here
}
type Subscription {
  # code goes here
}
```

Énumération

Les énumérations, ou énumérations, sont des scalaires spéciaux qui limitent les arguments juridiques qu'un type ou un champ peut avoir. Cela signifie que chaque fois qu'une énumération est définie dans le schéma, son type ou champ associé sera limité aux valeurs de l'énumération. Les

énumérations sont sérialisées sous forme de scalaires de chaînes. Notez que différents langages de programmation peuvent gérer les énumérations GraphQL différemment. Par exemple, n' JavaScript a pas de support d'énumération natif, de sorte que les valeurs d'énumération peuvent être mappées à des valeurs int à la place.

Les énumérations sont définies à l'aide du `enum` mot-clé. Voici un exemple :

```
enum trafficSignals {
  solidRed
  solidYellow
  solidGreen
  greenArrowLeft
  ...
}
```

Lors de l'appel de l'`trafficLights`énumération, le ou les arguments ne peuvent être que `solidRed``solidYellow`,`solidGreen`, etc. Il est courant d'utiliser des énumérations pour décrire des éléments qui offrent un nombre de choix distinct mais limité.

Unions/Interfaces

Voir Interfaces et unions dans GraphQL.

Champs GraphQL

Les champs existent dans le cadre d'un type et contiennent la valeur demandée au service GraphQL. Elles sont très similaires aux variables d'autres langages de programmation. Par exemple, voici un type d'`Person`objet :

```
type Person {
   name: String
   age: Int
}
```

Dans ce cas, les champs sont `name` `age` et contiennent respectivement une `Int` valeur `String` et. Les champs d'objet tels que ceux présentés ci-dessus peuvent être utilisés comme entrées dans les champs (opérations) de vos requêtes et mutations. Par exemple, consultez ce qui `Query` suit :

```
type Query {
  people: [Person]
```

```
}
```

Le `people` champ demande toutes les instances de `Person` depuis la source de données. Lorsque vous ajoutez ou extrayez un fichier `Person` dans votre serveur GraphQL, vous pouvez vous attendre à ce que les données suivent le format de vos types et de vos champs, c'est-à-dire que la structure de vos données dans le schéma détermine la manière dont elles seront structurées dans votre réponse :

```
}
  "data": {
    "people": [
      {
        "name": "John Smith",
        "age": "50"
      },
      {
        "name": "Andrew Miller",
        "age": "60"
      },
      .
      .
      .
    ]
  }
}
```

Les champs jouent un rôle important dans la structuration des données. Quelques propriétés supplémentaires expliquées ci-dessous peuvent être appliquées aux champs pour une personnalisation accrue.

Listes

Les listes renvoient tous les éléments d'un type spécifié. Une liste peut être ajoutée au type d'un champ à l'aide de crochets `[]` :

```
type Person {
  name: String
  age: Int
}
type Query {
  people: [Person]
```

```
}
```

Dans`Query`, les crochets qui l'entourent `Person` indiquent que vous souhaitez renvoyer toutes les instances `Person` de la source de données sous forme de tableau. Dans la réponse, les `age` valeurs `name` et de chacune `Person` seront renvoyées sous forme de liste unique et délimitée :

```
}
  "data": {
    "people": [
      {
        "name": "John Smith",         # Data of Person 1
        "age": "50"
      },
      {
        "name": "Andrew Miller",      # Data of Person 2
        "age": "60"
      },
      .                               # Data of Person N
      .
      .
    ]
  }
}
```

Vous n'êtes pas limité à des types d'objets spéciaux. Vous pouvez également utiliser des listes dans les champs des types d'objets classiques.

Non nuls

Les valeurs non nulles indiquent un champ qui ne peut pas être nul dans la réponse. Vous pouvez attribuer à un champ une valeur non nulle en utilisant le ! symbole :

```
type Person {
  name: String!
  age: Int
}
type Query {
  people: [Person]
}
```

Le `name` champ ne peut pas être explicitement nul. Si vous interrogiez la source de données et que vous fournissiez une entrée nulle pour ce champ, une erreur serait générée.

Vous pouvez combiner des listes et des valeurs non nulles. Comparez les requêtes suivantes :

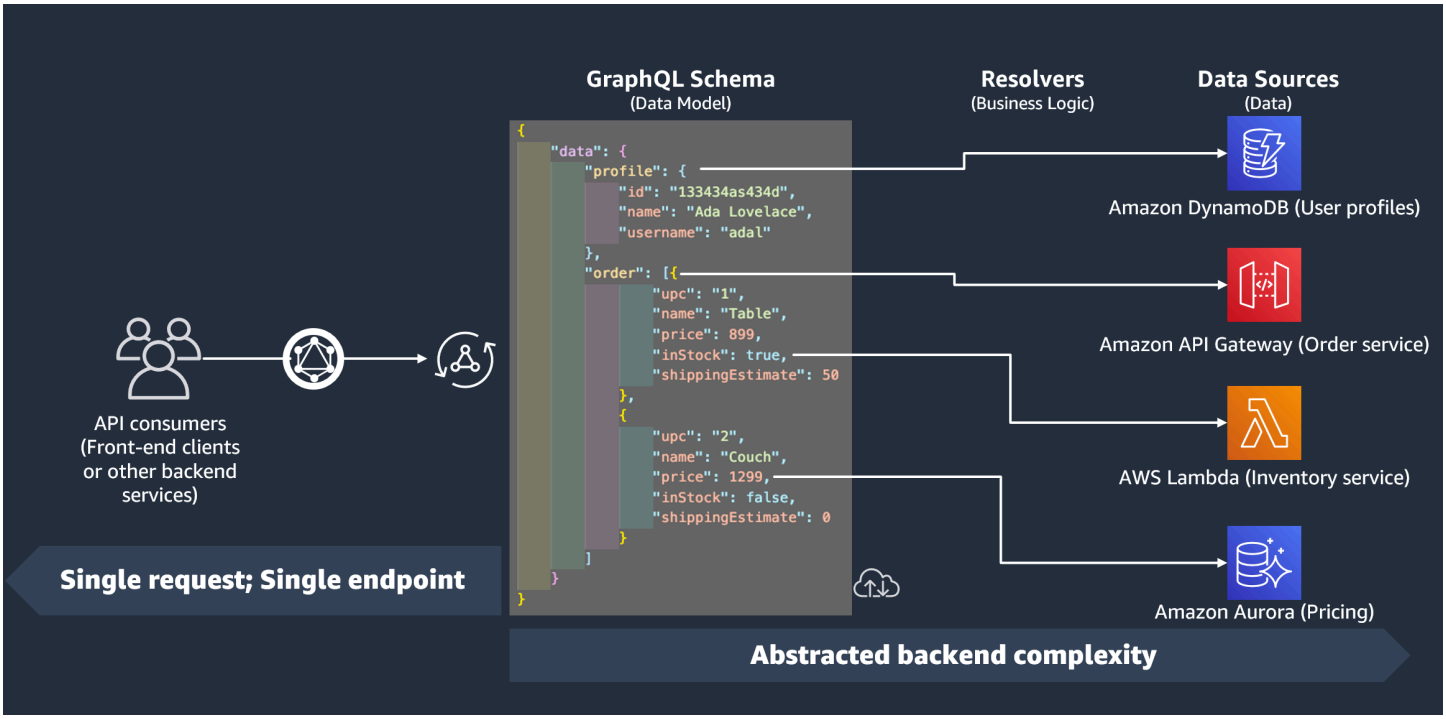
```
type Query {
  people: [Person!]      # Use case 1
}

.
.
.

type Query {
  people: [Person]!      # Use case 2
}

.
.
.

type Query {
  people: [Person!]!     # Use case 3
}
```

Dans le cas d'utilisation 1, la liste ne peut pas contenir d'éléments nuls. Dans le cas d'utilisation 2, la liste elle-même ne peut pas être définie sur null. Dans le cas d'utilisation 3, la liste et ses éléments ne peuvent pas être nuls. Cependant, dans tous les cas, vous pouvez toujours renvoyer des listes vides.

Comme vous pouvez le constater, GraphQL comporte de nombreux composants mobiles. Dans cette section, nous avons présenté la structure d'un schéma simple ainsi que les différents types et champs qu'un schéma prend en charge. Dans la section suivante, vous découvrirez les autres composants d'une API GraphQL et leur fonctionnement avec le schéma.

Sources de données

Dans la section précédente, nous avons appris qu'un schéma définit la forme de vos données. Cependant, nous n'avons jamais expliqué d'où venaient ces données. Dans les projets réels, votre schéma est comme une passerelle qui gère toutes les demandes adressées au serveur. Lorsqu'une demande est faite, le schéma agit comme le point de terminaison unique qui interagit avec le client. Le schéma accèdera aux données de la source de données, les traitera et les transmettra au client. Consultez l'infographie ci-dessous :



AWS AppSyncet GraphQL implémentent superbement des solutions Backend For Frontend (BFF). Ils travaillent en tandem pour réduire la complexité à grande échelle en faisant abstraction du backend. Si votre service utilise différentes sources de données et/ou microservices, vous pouvez essentiellement éliminer une partie de la complexité en définissant la forme des données de chaque source (sous-graphe) dans un schéma unique (supergraphe). Cela signifie que votre API GraphQL n'est pas limitée à l'utilisation d'une seule source de données. Vous pouvez associer autant de sources de données que vous le souhaitez à votre API GraphQL et spécifier dans votre code la manière dont elles interagiront avec le service.

Comme vous pouvez le voir dans l'infographie, le schéma GraphQL contient toutes les informations dont les clients ont besoin pour demander des données. Cela signifie que tout peut être traité en une seule demande plutôt que plusieurs requêtes comme c'est le cas avec REST. Ces demandes passent par le schéma, qui est le seul point de terminaison du service. Lorsque les demandes sont traitées, un résolveur (expliqué dans la section suivante) exécute son code pour traiter les données de la source de données correspondante. Lorsque la réponse est renvoyée, le sous-graphe lié à la source de données est rempli avec les données du schéma.

AWS AppSyncprend en charge de nombreux types de sources de données différents. Dans le tableau ci-dessous, nous allons décrire chaque type, énumérer certains de ses avantages et fournir des liens utiles pour plus de contexte.

Source de données	Description	Avantages	Informations supplémentaires

Amazon DynamoDB	« Amazon DynamoDB est un service de base de données NoSQL entièrement géré qui fournit des performances rapides et prévisibles avec une évolutivité sans faille. DynamoDB vous libère des charges administratives liées à l'exploitation et à la mise à l'échelle d'une base de données distribuée de façon que vous n'ayez pas à vous soucier de divers aspects tels que l'approvisionnement, le paramétrage, la configuration, la réplication, le matériel, les correctifs logiciels ou la mise à l'échelle de cluster. DynamoDB propose également le chiffrement au repos, ce qui élimine la charge opérationnelle et la complexité liées	• Performances à grande échelle : DynamoDB est conçu pour garantir des performances constantes à n'importe quelle échelle. Cela est possible grâce à l'utilisation de partitions. DynamoDB partitionnera automatiquement vos tables en plusieurs allocations qui seront stockées sur plusieurs SSD répartis sur plusieurs nœuds. Cela permettra généralement d'augmenter le débit du réseau et de réduire la latence.
		• Capacité à grande échelle : DynamoDB surveille votre trafic et vous permet d'ajuster automatiquement votre
			• Documentation officielle de DynamoDB
			  • Partitions
			  • Auto scaling (Mise à l'échelle automatique)
			  • Tolérance aux pannes
			  • Surveillance
			  • Sécurité
			• GraphQL et DynamoDB
			• Opérations de résolution pour DynamoDB
			• Modèle de tarification

Source de données	Description	Avantages	Informations supplémentaires
	à la protection des données sensibles. »	débit si le réseau reste surchargé pendant de longues périodes. • Disponibilité et tolérance aux pannes : DynamoDB est pris en charge par plusieurs régions physiquement isolées, chacune contenant plusieurs zones de disponibilité physiquement isolées. DynamoDB bascule automatiquement vers une zone de sauvegarde en cas d'interruption de service. Vous pouvez également sauvegarder et répliquer vos données manuellement pour garantir la sécurité des données. • Journalisation et surveillance : DynamoDB fournit plusieurs outils	

Source de données	Description	Avantages	Informations supplémentaires
		d'analyse pour vos tables. Vous pouvez surveiller les performances de votre table et créer des alarmes pour vous informer des modificat ions important es apportées au service. • Sécurité : DynamoDB suit des protocoles stricts pour garantir que vos données sont conformes aux exigences de sécurité de votre entreprise. • Intégration avec AWS AppSync : DynamoDB s'intègre parfaitem ent à notre service. Vous pouvez créer de nouvelles tables DynamoDB et générer automatiq uement un schéma à partir de celles-ci afin de rationali ser votre processus	

Source de données	Description	Avantages	Informations supplémentaires
		de développement. Nous proposons également un ensemble complet d'opérations pour demander facilement des données à partir de tables DynamoDB existantes dans votre compte dans votre résolveur.	

Source de données	Description	Avantages	Informations supplémentaires

AWS Lambda	« AWS Lambda est un service de calcul qui vous permet d'exécuter du code sans provisionner ni gérer de serveurs.
	
	Lambda exécute le code sur une infrastructure informatique à haute disponibilité et effectue toute l'administration des ressources informatiques, y compris la maintenance des serveurs et du système d'exploitation, l'allocation et la mise à l'échelle automatique des capacités, ainsi que la mise à l'échelle automatique et la journalisation. Avec Lambda, il vous suffit de fournir votre code dans l'un des environnements d'exécution de langage pris en charge par Lambda. »
		• ay-as-you-use Modèle P : Lambda ne vous facture que lorsque vous utilisez ses ressources. Ils vous permettent également d'adapter la quantité de ressources utilisées aux besoins de votre application.
		• Mise à l'échelle automatique : votre application peut parfois nécessiter une puissance de calcul supplémentaire pour un processus particulier. Lambda vous permet de dimensionner automatiquement les ressources informatiques en fonction des besoins de votre application.
		• Délais de déploiement accélérés : vous
			• Documentation officielle
			  • Dimensionnement
			  • déploiement
			  • runtimes
			• Tutoriel sur le résolveur Lambda
			• Modèle de tarification

Source de données	Description	Avantages	Informations supplémentaires
		pouvez rationaliser votre processus de développement grâce à un package de déploiement. Utilisez un package pour télécharg er votre code de fonction vers le service Lambda. Vous pouvez ensuite utiliser leurs environnements d'exécution pour tester et exécuter vos fonctions. • Polyvalence : Lambda peut être utilisé dans une multitude de cas d'utilisation. Vous pouvez intégrer Lambda en toute simplicité à des services tiers ou à des AWS services. Les pipelines CI/CD et les services de publipostage en sont quelques exemples. • Intégration avec AWS AppSync :	

Source de données	Description	Avantages	Informations supplémentaires
		Vous pouvez facilement invoquer vos fonctions Lambda dans votre résolveur pour traiter les demandes. Notre service fournit une opération de demande rationalisée pour effectuer des appels Lambda. Nous autorisons les appels uniques et groupés.	

Source de données	Description	Avantages	Informations supplémentaires
OpenSearch	« Amazon OpenSearch Service est un service géré qui facilite le déploiement, l'exploitation et le dimensionnement de OpenSearch clusters dans le AWS cloud. Amazon OpenSearch Service prend en charge OpenSearch les anciens logiciels Elasticsearch OSS (jusqu'à la version 7.10, dernière version open source du logiciel). Lorsque vous créez un cluster, vous avez la possibilité de choisir le moteur de recherche que vous voulez utiliser. OpenSearchest un moteur de recherche et d'analyse entièrement open source pour des cas d'utilisation tels que l'analyse des journaux, la surveillance des applications en temps réel et l'analyse du flux	• Mise à l'échelle : vous pouvez facilement adapter le service à vos besoins grâce à OpenSearch Serverless. • Ingestion de données : vous pouvez utiliser OpenSearch l'ingestion pour importer, traiter et analyser des données. Il existe de nombreuses applications pour l'ingestion de données, que vous pouvez trouver ici. • Sécurité : OpenSearch peut gérer votre configuration AWS de sécurité, y compris l'IAM CloudTrail, les VPC, l'authentification, etc. • Disponibilité : prend OpenSearch également en charge différentes	• Documentation officielle • Serverless (Sans serveur) • Modèle de tarification

Source de données	Description	Avantages	Informations supplémentaires
	de clics. Pour en savoir plus, consultez la documentation OpenSearch. Amazon OpenSearch Service fournit toutes les ressources pour votre OpenSearch cluster et le lance. Il détecte et remplace également automatiq uement les nœuds de OpenSearch service défaillants, réduisant ainsi les frais associés aux infrastructures autogérées. Vous pouvez faire évoluer votre cluster en un seul appel d'API ou en quelques clics dans la console. »	régions et zones de disponibilité dans son service. • Intégration avec AWS AppSync : Dans AWS AppSync, vous pouvez utiliser les API GraphQL pour stocker et récupérer des données à partir de domaines de OpenSearch service existants dans votre compte.	

Source de données	Description	Avantages	Informations supplémentaires
Points de terminaison HTTP	Vous pouvez utiliser des points de terminaison HTTP comme sources de données. AWS AppSync peut envoyer des demandes aux points de terminaison avec les informations pertinentes telles que les paramètres et la charge utile. La réponse HTTP sera exposée au résolveur , qui renverra la réponse finale une fois ses opérations terminées.	• Utile pour les applications simples qui ne sont pas aussi intégrées à des services tels que Lambda.	• Référence du résolveur

Source de données	Description	Avantages	Informations supplémentaires

Amazon EventBridge	« EventBridge est un service sans serveur qui utilise des événements pour connecter les composants de l'application entre eux, ce qui vous permet de créer plus facilement des applications évolutives pilotées par des événements. Utilisez-le pour acheminer des événements provenant de sources telles que des applications locales, des AWS services et des logiciels tiers vers des applications grand public au sein de votre entreprise. EventBridge fournit un moyen simple et cohérent d'ingérer, de filtrer, de transformer et de diffuser des événements afin que vous puissiez créer rapidement de nouvelles applications. »	• Architecture pilotée par les événements : vous pouvez tirer parti de l'architecture pilotée par les événements.
		• Planification : vous pouvez utiliser le EventBridge planificateur pour automatiser vos tâches et vos règles à l'aide d'expressions cron ou définir des intervalles de temps comme alternative aux modèles d'événements.
		• Canaux : à l'EventBridge aide de canaux, vous pouvez remplacer le bus d'événements par un canal qui inclut des modèles d'événements de filtrage supplémentaires et un enrichissement par le biais de transformations de données
			• Documentation officielle
			  • Tuyaux
			  • Planificateur
			• Référence du résolveur
			• Modèle de tarification

Source de données	Description	Avantages	Informations supplémentaires
		avant d'envoyer l'événement à la cible. • Intégration avec AWS AppSync : vous AWS AppSync permet d'envoyer des événements aux bus d'événements à l'aide de votre résolveur.	

Source de données	Description	Avantages	Informations supplémentaires

Bases de données relationnelles	« Amazon Relational Database Service (Amazon RDS) est un service Web qui facilite la configuration, l'exploitation et le dimensionnement d'une base de données relationnelle dans le cloud. AWS Il fournit une capacité redimensionnable et rentable pour une base de données relationnelle standard et gère les tâches d'administration de base de données courantes. »	• Gestion simplifiée : RDS effectue régulièrement la maintenance de ses ressources. La maintenance implique le plus souvent des mises à jour du matériel sous-jacent, du système d'exploitation sous-jacent ou de la version du moteur de base de données de l'instance de base de données. Dans des circonstances normales, vous pouvez décider à quel moment effectuer les mises à jour (les exceptions incluent les correctifs de sécurité).
		• Recommandations : La fonction de recommandation de RDS fournit des suggestions automatisées pour résoudre les
			• Documentation officielle
			  • Fonctions
			  • Maintenance
			  • Recommandations
			  • Options de stockage
			  • Disponibilité
			  • Sécurité
			• Modèle de tarification

Source de données	Description	Avantages	Informations supplémentaires
		problèmes potentiels de votre instance. • Disponibilité : RDS est disponible dans différentes régions physiques du monde entier. Vous pouvez facilement répartir vos besoins en matière de base de données sur différents nœuds afin de fournir un meilleur service à vos clients. • Personnalisation : RDS est conçu pour répondre aux exigences des grandes entreprises. RDS propose diverses options pour le calcul, le déploiement rapide, l'évolutivité et le stockage. • Sécurité : RDS est intégré à plusieurs outils et services pour maintenir la sécurité des bases de données au niveau de l'utilisa	

Source de données	Description	Avantages	Informations supplémentaires
		teur, de la base de données et du réseau. • Intégration avec AWS AppSync : Si vous recherche z une solution de backend mature, elle vous AWS AppSync permet d'envoyer, de traiter, de stocker et de renvoyer des données en utilisant votre instance comme source de données.	

Source de données	Description	Avantages	Informations supplémentaires
Aucune source de données	Si vous n'avez pas l'intention d'utiliser un service de source de données, vous pouvez le configurer surnone. Une source de none données, bien qu'elle soit toujours explicite ment classée comme source de données, n'est pas un support de stockage. Malgré cela, il est toujours utile dans certains cas pour la manipulat ion et le transfert de données.	• Potentiellement utile pour des choses comme la conversion de données • Utile pour résoudre un problème localement	• Référence du résolveur

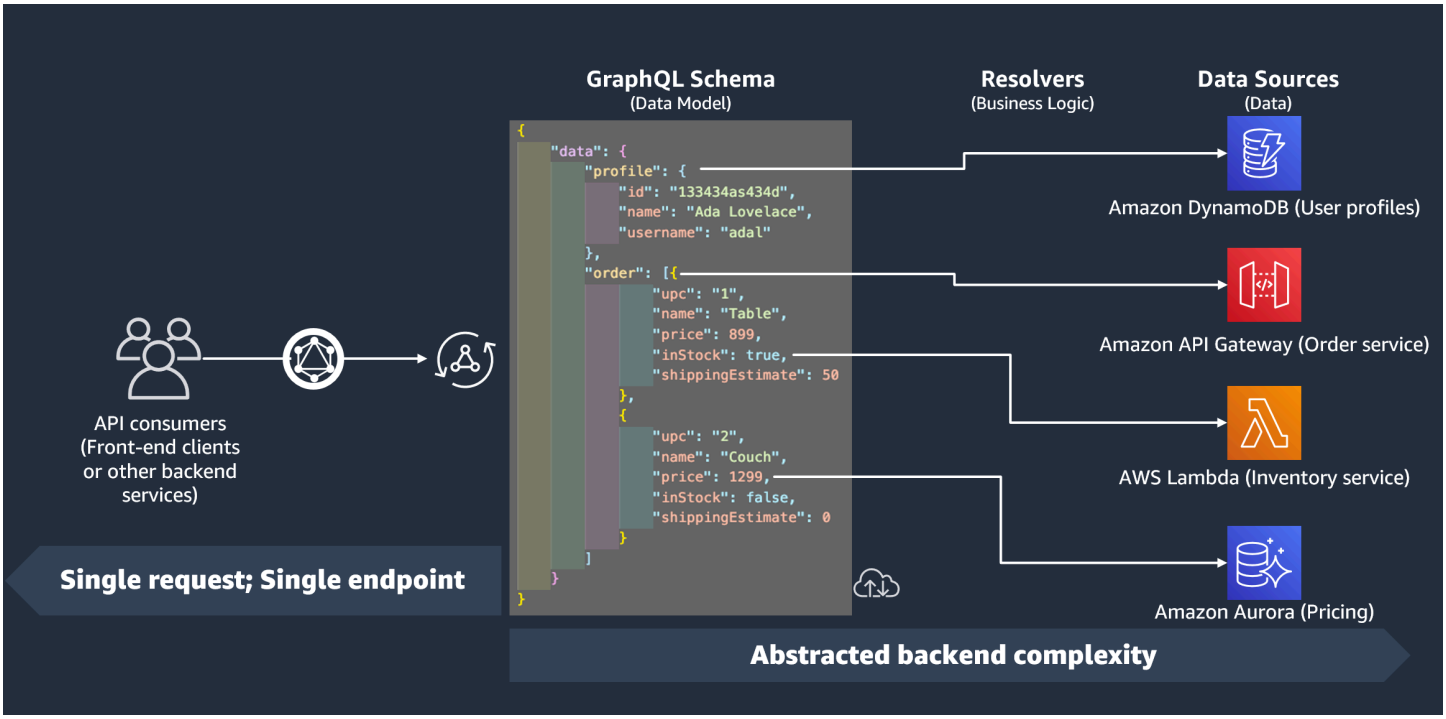
> **Tip**
>
> Pour plus d'informations sur la façon dont les sources de données interagissent avec ellesAWS AppSync, voir Joindre une source de données.

Résolveurs

Dans les sections précédentes, vous avez découvert les composants du schéma et de la source de données. Nous devons maintenant examiner la manière dont le schéma et les sources de données interagissent. Tout commence par le résolveur.

Un résolveur est une unité de code qui gère la manière dont les données de ce champ seront résolues lorsqu'une demande est envoyée au service. Les résolveurs sont attachés à des

champs spécifiques au sein de vos types dans votre schéma. Ils sont le plus souvent utilisés pour implémenter les opérations de changement d'état pour vos opérations de terrain de requête, de mutation et d'abonnement. Le résolveur traitera la demande d'un client, puis renverra le résultat, qui peut être un groupe de types de sortie tels que des objets ou des scalaires :



Temps d'exécution du résolveur

DansAWS AppSync, vous devez d'abord spécifier un environnement d'exécution pour votre résolveur. Un environnement d'exécution d'un résolveur indique l'environnement dans lequel un résolveur est exécuté. Il dicte également la langue dans laquelle vos résolveurs seront écrits. AWS AppSyncsupporte actuellement APPSYNC_JS pour JavaScript et Velocity Template Language (VTL). Consultez les fonctionnalités JavaScript d'exécution pour les résolveurs et les fonctions JavaScript ou la référence de l'utilitaire de modèle de mappage Resolver pour VTL.

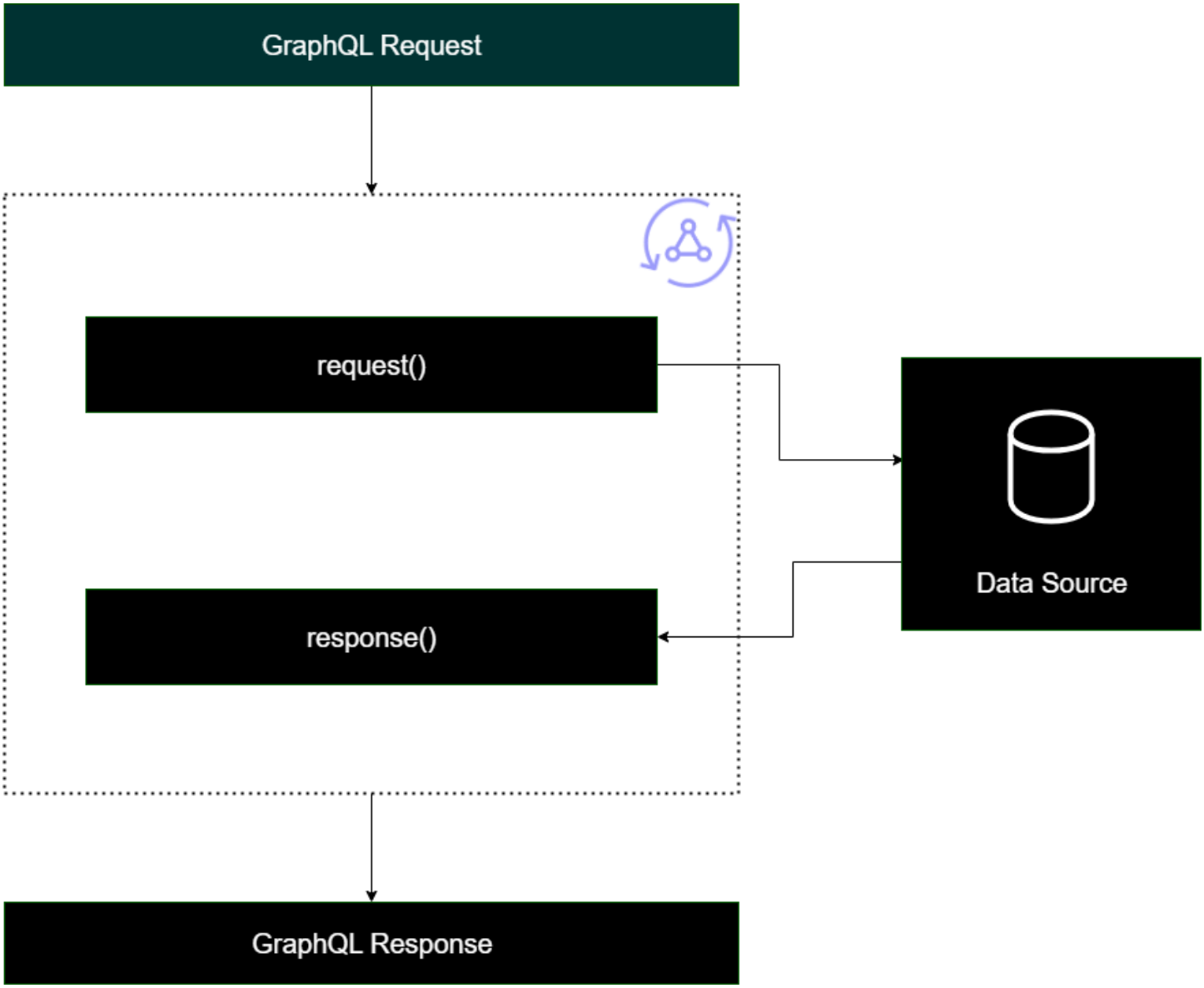
Structure du résolveur

Du point de vue du code, les résolveurs peuvent être structurés de plusieurs manières. Il existe des résolveurs d'unités et de pipelines.

Résolveurs d'unités

Un résolveur d'unités est composé d'un code qui définit un seul gestionnaire de demandes et de réponses exécuté sur une source de données. Le gestionnaire de demandes prend un objet de contexte comme argument et renvoie la charge utile de la demande utilisée pour appeler votre source

de données. Le gestionnaire de réponses reçoit une charge utile en retour de la source de données avec le résultat de la demande exécutée. Le gestionnaire de réponse transforme la charge utile en réponse GraphQL pour résoudre le champ GraphQL.

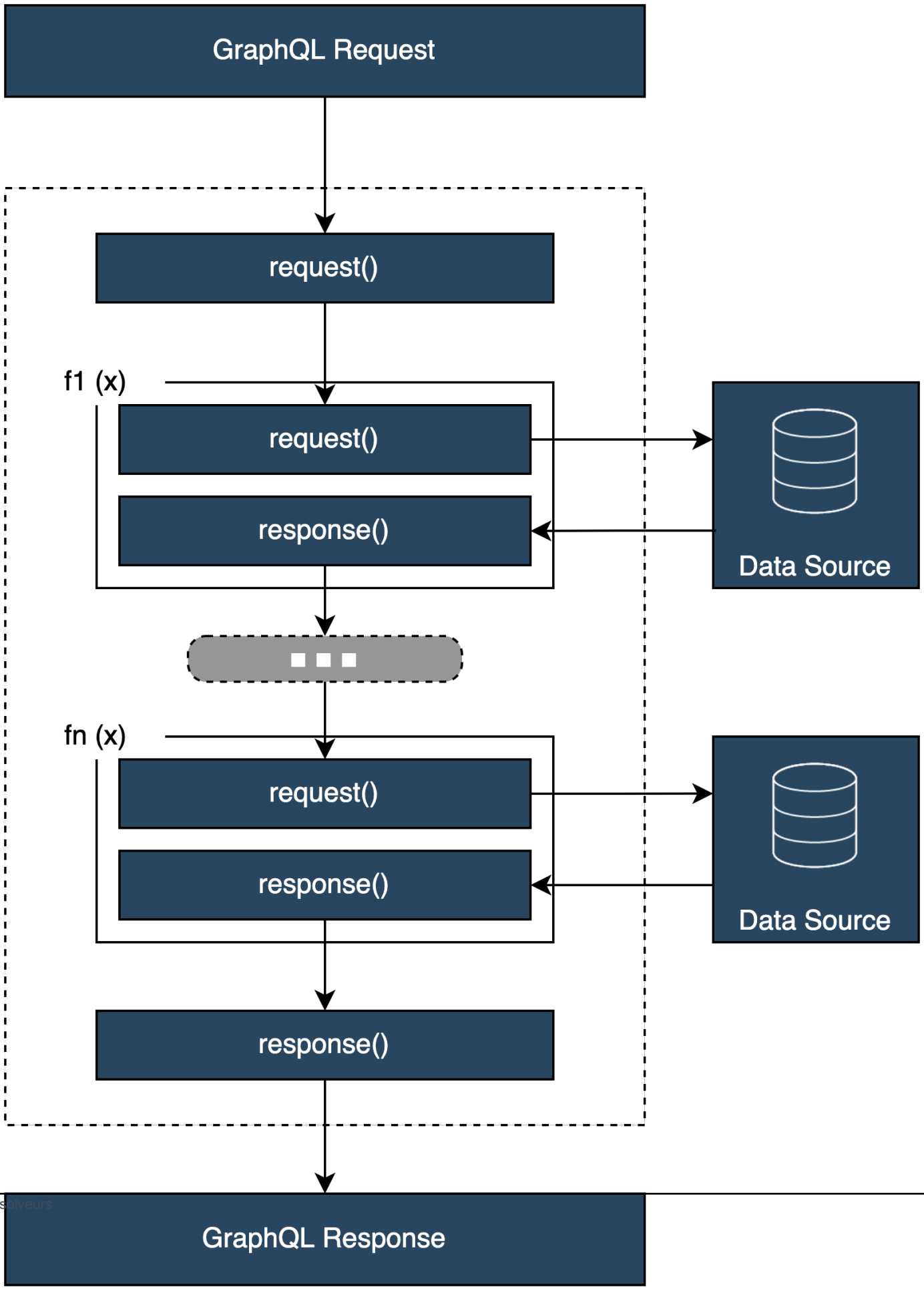


Résolveurs de pipelines

Lors de la mise en œuvre de résolveurs de pipeline, ils suivent une structure générale :

- Avant l'étape : lorsqu'une demande est faite par le client, les données de la demande sont transmises aux résolveurs des champs de schéma utilisés (généralement vos requêtes, mutations, abonnements). Le résolveur commencera à traiter les données de la demande à l'aide d'un gestionnaire avant étape, ce qui permet d'effectuer certaines opérations de prétraitement avant que les données ne passent par le résolveur.

- Fonction (s) : Une fois l'étape précédente exécutée, la demande est transmise à la liste des fonctions. La première fonction de la liste s'exécutera sur la source de données. Une fonction est un sous-ensemble du code de votre résolveur contenant son propre gestionnaire de requêtes et de réponses. Un gestionnaire de demandes prendra les données de la demande et effectuera des opérations sur la source de données. Le gestionnaire de réponses traitera la réponse de la source de données avant de la renvoyer à la liste. S'il existe plusieurs fonctions, les données de la demande seront envoyées à la fonction suivante de la liste à exécuter. Les fonctions de la liste seront exécutées en série dans l'ordre défini par le développeur. Une fois que toutes les fonctions ont été exécutées, le résultat final est transmis à l'étape suivante.
- Étape suivante : L'étape suivante est une fonction de gestion qui vous permet d'effectuer certaines opérations finales sur la réponse de la fonction finale avant de la transmettre à la réponse GraphQL.


GraphQL Request
request()
f1 (x)
request()
response()
Data Source
fn (x)
request()
response()
Data Source
response()
GraphQL Response

Structure du gestionnaire du résolveur

Les gestionnaires sont généralement des fonctions appelées `Request` et `Response` :

```
export function request(ctx) {
    // Code goes here
}

export function response(ctx) {
    // Code goes here
}
```

Dans un résolveur d'unités, il n'y aura qu'un seul ensemble de ces fonctions. Dans un résolveur de pipeline, il y en aura un ensemble pour les étapes avant et après et un ensemble supplémentaire par fonction. Pour visualiser à quoi cela pourrait ressembler, examinons un `Query` type simple :

```
type Query {
 helloWorld: String!
}
```

Il s'agit d'une requête simple avec un champ appelé `helloWorld` type`String`. Supposons que nous voulions toujours que ce champ renvoie la chaîne « Hello World ». Pour implémenter ce comportement, nous devons ajouter le résolveur dans ce champ. Dans un résolveur d'unités, nous pourrions ajouter quelque chose comme ceci :

```
export function request(ctx) {
    return {}
}

export function response(ctx) {
    return "Hello World"
}
```

`request`Vous pouvez simplement laisser ce champ vide car nous ne demandons ni ne traitons de données. Nous pouvons également supposer que notre source de données l'est`None`, ce qui indique que ce code n'a pas besoin d'effectuer d'invocations. La réponse renvoie simplement « Hello World ». Pour tester ce résolveur, nous devons faire une demande en utilisant le type de requête :

```
query helloWorldTest {
```

```
  helloWorld
}
```

Il s'agit d'une requête appelée `helloWorldTest` qui renvoie le `helloWorld` champ. Lorsqu'il est exécuté, le résolveur de `helloWorld` champs exécute et renvoie également la réponse :

```
{
  "data": {
    "helloWorld": "Hello World"
  }
}
```

Renvoyer des constantes comme celle-ci est la chose la plus simple que vous puissiez faire. En réalité, vous allez renvoyer des entrées, des listes, etc. Voici un exemple plus complexe :

```
type Book {
  id: ID!
  title: String
}

type Query {
  getBooks: [Book]
}
```

Nous renvoyons ici une liste de`Books`. Supposons que nous utilisions une table DynamoDB pour stocker les données d'un livre. Nos gestionnaires peuvent ressembler à ceci :

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

Notre demande a utilisé une opération de numérisation intégrée pour rechercher toutes les entrées de la table, a stocké les résultats dans le contexte, puis les a transmis à la réponse. La réponse a pris les éléments du résultat et les a renvoyés dans la réponse :

```
{
  "data": {
    "getBooks": {
      "items": [
        {
          "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
          "title": "book1"
        },
        {
          "id": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
          "title": "book2"
        },

        ...

      ]
    }
  }
}
```

Contexte du résolveur

Dans un résolveur, chaque étape de la chaîne de gestionnaires doit connaître l'état des données des étapes précédentes. Le résultat d'un gestionnaire peut être stocké et transmis à un autre en tant qu'argument. GraphQL définit quatre arguments de base du résolveur :

Arguments de base du résolveur	Description
`obj`, `root`, `parent`, etc.	Le résultat du parent.
`args`	Les arguments fournis au champ dans la requête GraphQL.
`context`	Une valeur qui est fournie à chaque résolveur et contient des informations contextuelles

Arguments de base du résolveur	Description
	importantes telles que l'utilisateur actuellement connecté ou l'accès à une base de données.
`info`	Une valeur qui contient des informations spécifiques au champ pertinentes pour la requête en cours ainsi que les détails du schéma.

DansAWS AppSync, l'argument `context` (ctx) peut contenir toutes les données mentionnées ci-dessus. Il s'agit d'un objet créé par demande et qui contient des données telles que les informations d'identification d'autorisation, les données de résultats, les erreurs, les métadonnées des demandes, etc. Le contexte est un moyen facile pour les programmeurs de manipuler les données provenant d'autres parties de la requête. Reprenez cet extrait :

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

La requête reçoit le contexte (ctx) comme argument ; il s'agit de l'état de la demande. Il effectue une analyse de tous les éléments d'un tableau, puis stocke le résultat dans le contexte dans`result`. Le contexte est ensuite transmis à l'argument de réponse, qui accède au `result` et renvoie son contenu.

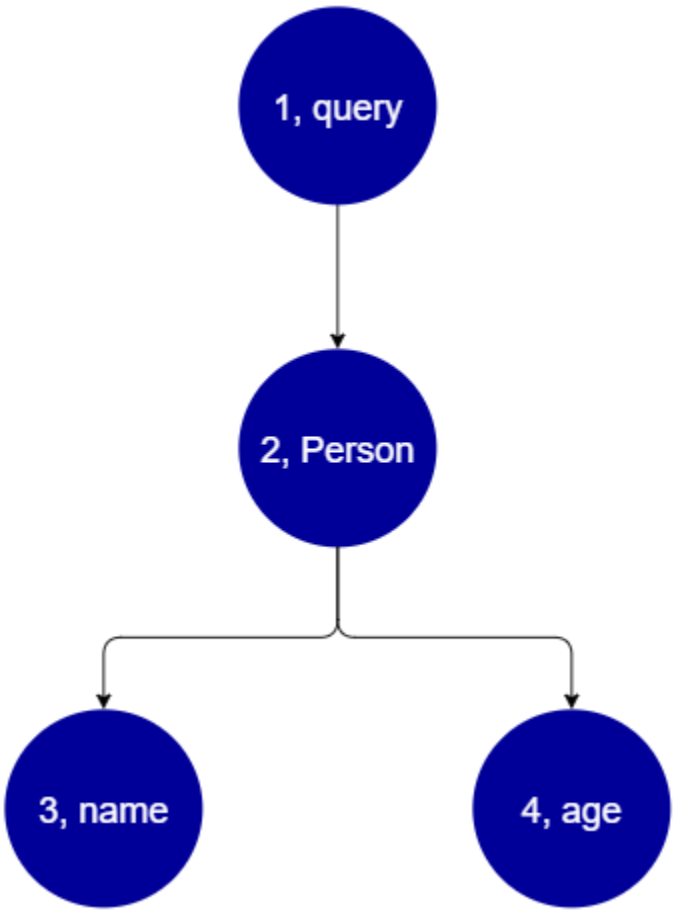
Requêtes et analyse syntaxique

Lorsque vous envoyez une requête à votre service GraphQL, celle-ci doit passer par un processus d'analyse et de validation avant d'être exécutée. Votre demande sera analysée et traduite dans un

arbre syntaxique abstrait. Le contenu de l'arborescence est validé en exécutant plusieurs algorithmes de validation par rapport à votre schéma. Après l'étape de validation, les nœuds de l'arbre sont parcourus et traités. Les résolveurs sont appelés, les résultats sont stockés dans le contexte et la réponse est renvoyée. Prenons, par exemple, cette requête :

```
query {
  Person {  //object type
    name  //scalar
    age   //scalar
  }
}
```

Nous revenons `Person` avec un `name` et des `age` champs. Lors de l'exécution de cette requête, l'arborescence ressemblera à ceci :



D'après l'arborescence, il apparaît que cette demande recherchera la racine `Query` dans le schéma. À l'intérieur de la requête, le `Person` champ sera résolu. D'après les exemples précédents, nous savons qu'il peut s'agir d'une entrée de l'utilisateur, d'une liste de valeurs, etc. Elle `Person` est très probablement liée à un type d'objet contenant les champs dont nous avons besoin (`name`et`age`).

Une fois que ces deux champs enfants sont trouvés, ils sont résolus dans l'ordre indiqué (`name`suivi de`age`). Une fois que l'arbre est complètement résolu, la demande est terminée et sera renvoyée au client.

Propriétés supplémentaires de GraphQL

GraphQL repose sur plusieurs principes de conception visant à maintenir la simplicité et la robustesse à grande échelle.

Déclaratif

GraphQL est déclaratif, ce qui signifie que l'utilisateur décrira (façonnera) les données en déclarant uniquement les champs qu'il souhaite interroger. La réponse renverra uniquement les données relatives à ces propriétés. *Par exemple, voici une opération qui récupère un Book objet dans une table DynamoDB dont la valeur id ISBN 13 est 9780199536061 :*

```
{
  getBook(id: "9780199536061") {
    name
    year
    author
  }
}
```

La réponse renverra les champs de la charge utile (`name`,`year`, et`author`) et rien d'autre :

```
{
  "data": {
    "getBook": {
      "name": "Anna Karenina",
      "year": "1878",
      "author": "Leo Tolstoy",
    }
  }
}
```

Grâce à ce principe de conception, GraphQL élimine les problèmes récurrents de surextraction et de sous-extraction auxquels sont confrontées les API REST dans les systèmes complexes. Cela se traduit par une collecte de données plus efficace et une amélioration des performances du réseau.

Hiérarchique

GraphQL est flexible dans la mesure où les données demandées peuvent être façonnées par l'utilisateur pour répondre aux besoins de l'application. Les données demandées suivent toujours les types et la syntaxe des propriétés définies dans votre API GraphQL. *Par exemple, l'extrait suivant montre l'getBookopération avec une nouvelle portée de champ appelée quotes qui renvoie toutes les chaînes de guillemets stockées et les pages liées au 9780199536061 : Book*

```
{
  getBook(id: "9780199536061") {
    name
    year
    author
    quotes {
      description
      page
    }
  }
}
```

L'exécution de cette requête renvoie le résultat suivant :

```
{
  "data": {
    "getBook": {
      "name": "Anna Karenina",
      "year": "1878",
      "author": "Leo Tolstoy",
      "quotes": [
         {
            "description": "The highest Petersburg society is essentially one: in it
 everyone knows everyone else, everyone even visits everyone else.",
            "page": 135
         },
         {
            "description": "Happy families are all alike; every unhappy family is
 unhappy in its own way.",
            "page": 1
         },
         {
```

```
            "description": "To Konstantin, the peasant was simply the chief partner in
 their common labor.",
            "page": 251
          }
      ]
    }
  }
}
```

Comme vous pouvez le constater, les `quotes` champs liés au livre demandé ont été renvoyés sous forme de tableau dans le même format que celui décrit dans notre requête. Bien qu'il n'ait pas été présenté ici, GraphQL présente l'avantage supplémentaire de ne pas être précis quant à l'emplacement des données qu'il récupère. `Bookset` `quotes` pourraient être stockés séparément, mais GraphQL récupérera toujours les informations tant que l'association existe. Cela signifie que votre requête peut récupérer une multitude de données autonomes en une seule demande.

Introspectif

GraphQL est autodocumenté, c'est-à-dire introspectif. Il prend en charge plusieurs opérations intégrées qui permettent aux utilisateurs de visualiser les types et les champs sous-jacents du schéma. Par exemple, voici un `Foo` type avec un `description` champ `date` et :

```
type Foo {
 date: String
 description: String
}
```

Nous pourrions utiliser l'`_type`opération pour trouver les métadonnées de saisie sous le schéma :

```
{
  __type(name: "Foo") {
    name                    # returns the name of the type
    fields {                # returns all fields in the type
      name                  # returns the name of each field
      type {                # returns all types for each field
        name                # returns the scalar type
      }
    }
  }
}
```

Cela renverra une réponse :

```
{
  "__type": {
    "name": "Foo",                    # The type name
    "fields": [
      {
        "name": "date",               # The date field
        "type": { "name": "String" }   # The date's type
      },
      {
        "name": "description",        # The description field
        "type": { "name": "String" }   # The description's type
      },
    ]
  }
}
```

Cette fonctionnalité peut être utilisée pour déterminer les types et les champs pris en charge par un schéma GraphQL particulier. GraphQL prend en charge une grande variété de ces opérations introspectives. Pour plus d'informations, consultez Introspection.

Typographie forte

GraphQL prend en charge la saisie forte grâce à son système de types et de champs. Lorsque vous définissez un élément dans votre schéma, il doit avoir un type qui peut être validé avant l'exécution. Il doit également suivre les spécifications de syntaxe de GraphQL. Ce concept n'est pas différent de la programmation dans d'autres langages. Par exemple, voici le `Foo` type précédent :

```
type Foo {
 date: String
 description: String
}
```

Nous pouvons voir que `Foo` c'est l'objet qui sera créé. À l'intérieur d'une instance de`Foo`, il y aura un `description` champ `date` et, tous deux de type `String` primitif (scalaire). Syntaxiquement, nous voyons que cela `Foo` a été déclaré et que ses champs existent dans son champ d'application. Cette combinaison de vérification de type et de syntaxe logique garantit que votre API GraphQL est concise et évidente. Les spécifications de typage et de syntaxe de GraphQL se trouvent ici.

Mise en route : création de votre première API GraphQL

Vous pouvez utiliserAWS AppSyncconsole pour configurer et lancer une API GraphQL. Les API GraphQL nécessitent généralement trois composants :

1. Schéma GraphQL- Votre schéma GraphQL est le modèle de l'API. Il définit les types et les champs que vous pouvez demander lorsqu'une opération est exécutée. Pour remplir le schéma avec des données, vous devez connecter les sources de données à l'API GraphQL. Dans ce guide de démarrage rapide, nous allons créer un schéma à l'aide d'un modèle prédéfini.
2. Sources de données- Il s'agit des ressources qui contiennent les données permettant de remplir votre API GraphQL. Il peut s'agir d'une table DynamoDB, d'une fonction Lambda, etc.AWS AppSyncprend en charge une multitude de sources de données pour créer des API GraphQL robustes et évolutives. Les sources de données sont liées aux champs du schéma. Chaque fois qu'une demande est effectuée sur un champ, les données de la source renseignent le champ. Ce mécanisme est contrôlé par le résolveur. Dans ce guide de démarrage rapide, nous allons créer une source de données à l'aide d'un modèle prédéfini associé au schéma.
3. Résolveurs- Les résolveurs sont chargés de lier le champ de schéma à la source de données. Ils récupèrent les données de la source, puis renvoient le résultat en fonction de ce qui a été défini par le champ.AWS AppSyncprend en charge les deuxJavaScriptet VTL pour écrire des résolveurs pour vos API GraphQL. Dans ce guide de démarrage rapide, les résolveurs seront automatiquement générés en fonction du schéma et de la source de données. Nous n'allons pas approfondir cette question dans cette section.

AWS AppSyncprend en charge la création et la configuration de tous les composants GraphQL. Lorsque vous ouvrez la console, vous pouvez utiliser les méthodes suivantes pour créer votre API :

1. Conception d'une API GraphQL personnalisée en la générant via un modèle prédéfini et en configurant une nouvelle table DynamoDB (source de données) pour la prendre en charge.
2. Conception d'une API GraphQL avec un schéma vide, sans sources de données ni résolveurs.
3. Utilisation d'une table DynamoDB pour importer des données et générer les types et les champs de votre schéma.
4. En utilisantAWS AppSyncc'estWebSocketcapacités et architecture Pub/Sub pour développer des API en temps réel.
5. Utilisation des API GraphQL existantes (API source) pour établir un lien vers une API fusionnée.

> **Note**
>
> Nous vous recommandons de consulter leConception d'un schémasection avant de travailler avec des outils plus avancés. Ces guides expliqueront des exemples plus simples que vous pouvez utiliser de manière conceptuelle pour créer des applications plus complexes dansAWS AppSync.

AWS AppSyncprend également en charge plusieurs options hors console pour créer des API GraphQL. Il s'agit des licences suivantes :

1. AWS Amplify
2. AWS SAM
3. AWS CloudFormation
4. Le CDK

L'exemple suivant vous montre comment créer les composants de base d'une API GraphQL à l'aide de modèles prédéfinis et de DynamoDB.

Rubriques

Étape 1 : Lancer un schéma

Dans cet exemple, vous allez créer unTodoAPI qui permet aux utilisateurs de créerTododes articles pour les rappels de corvées quotidiennes, tels que`Terminer la tâche`ou`Récupérez les produits d'épicerie`. Cette API montre comment utiliser les opérations GraphQL lorsque l'état persiste dans une table DynamoDB.

Conceptuellement, la création de votre première API GraphQL comporte trois étapes principales. Vous devez définir le schéma (types et champs), associer vos sources de données à vos champs,

puis écrire le résolveur qui gère la logique métier. Toutefois, l'expérience de la console modifie l'ordre des choses. Nous allons commencer par définir la manière dont nous voulons que notre source de données interagisse avec notre schéma, puis définirons le schéma et le résolveur ultérieurement.

Pour créer votre API GraphQL

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
2. Dans le Tableau de bord, choisissez Créer une API.
3. Tandis queAPI GraphQLest sélectionné, choisissezDesign à partir de zéro. Ensuite, choisissez Next (Suivant).
4. PourNom de l'API, remplacez le nom prérempli par**`Todo API`**, puis choisissezSuivant.

   > **Note**
   >
   > D'autres options sont également présentes ici, mais nous ne les utiliserons pas dans cet exemple.

5. Dans leSpécifier les ressources GraphQLsection, procédez comme suit :

   a. ChoisissezCréez du type basé sur une table DynamoDB dès maintenant.

      > **Note**
      >
      > Cela signifie que nous allons créer une nouvelle table DynamoDB à joindre en tant que source de données.

   b. Dans leNom du modèlechamp, entrez**`Todo`**.

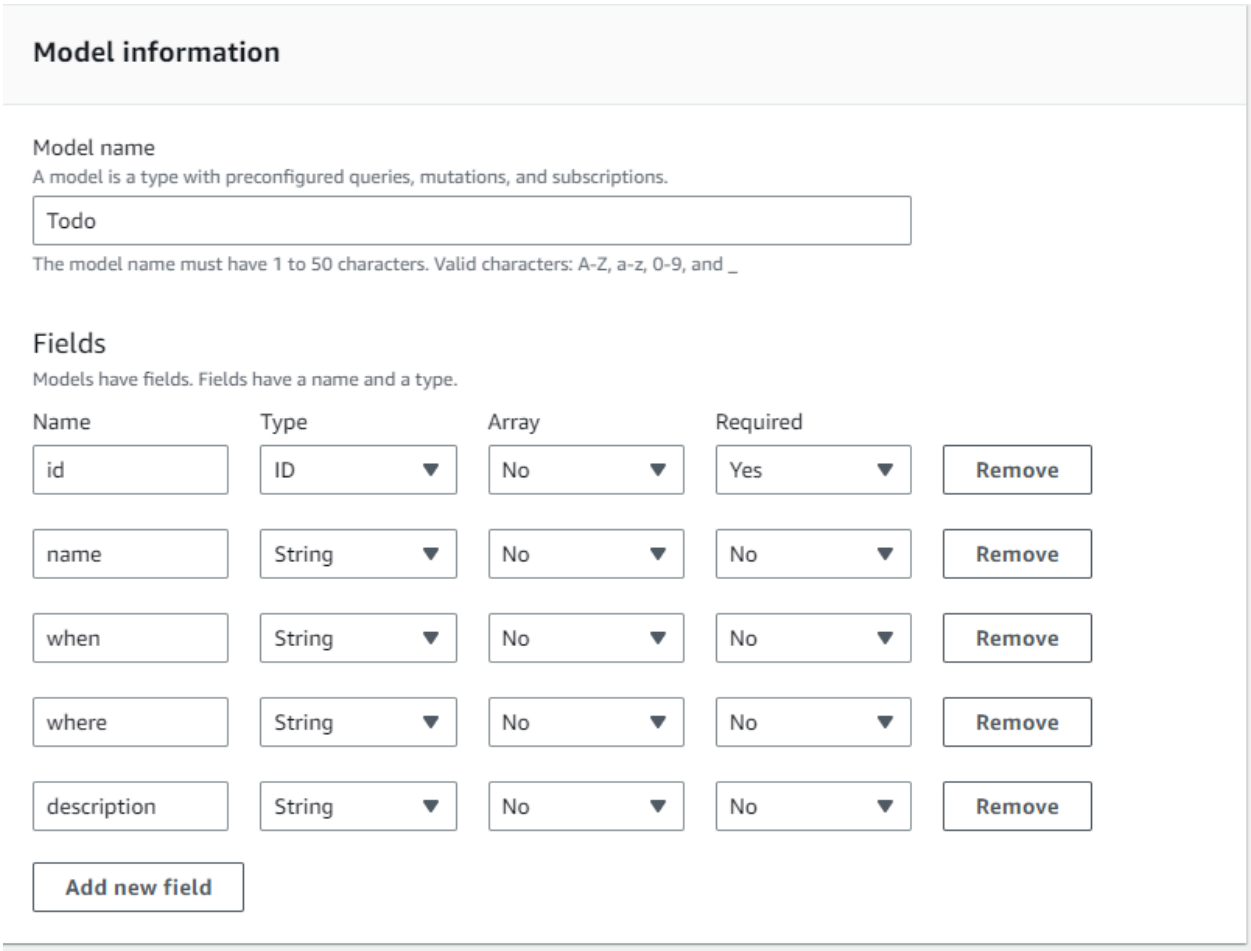
      > **Note**
      >
      > Notre première exigence est de définir notre schéma. CeNom du modèlesera le nom du type, donc ce que vous faites réellement est de créer un`type`appelé`Todo`qui existera dans le schéma :
      >
      > ```
      > type Todo {}
      > ```

   c. En dessousChamps, procédez comme suit :

i. Créez un champ nommé**`id`**, avec le type`ID`, et doit être défini sur`Yes`.

> **Note**
>
> Ce sont les champs qui existeront dans le cadre de votre`Todo`type. Le nom de votre champ ici sera appelé`id`avec un type de`ID!`:
>
> ```
> type Todo {
>  id: ID!
> }
> ```
>
> AWS AppSyncprend en charge plusieurs valeurs scalaires pour différents cas d'utilisation.

ii. En utilisantAjouter un nouveau champ, créez quatre champs supplémentaires avec`Name`valeurs définies sur**`name`**,**`when`**,**`where`**, et**`description`**. Leur`Type`les valeurs seront`String`, et le`Array`et`Required`les valeurs seront toutes deux définies surNo. Il doit ressembler à ce qui suit :



Note

Le type complet et ses champs ressembleront à ceci :

```
type Todo {
 id: ID!
 name: String
 when: String
 where: String
 description: String
}
```

Comme nous créons un schéma à l'aide de ce modèle prédéfini, il sera également rempli de plusieurs mutations standard basées sur le type, telles que`create`,`delete`, et`update`pour vous aider à remplir facilement votre source de données.

d. En dessousconfigurer la table des modèles, entrez un nom de table, tel que**`TodoAPITable`**. Réglez leClé primairepour`id`.

> **Note**
>
> Nous créons essentiellement une nouvelle table DynamoDB appelée*`Todo Apitable`*qui sera attachée à l'API en tant que source de données principale. Notre clé primaire est définie sur le paramètre requis`id`champ que nous avons défini auparavant. Notez que cette nouvelle table est vide et ne contient rien d'autre que la clé de partition.

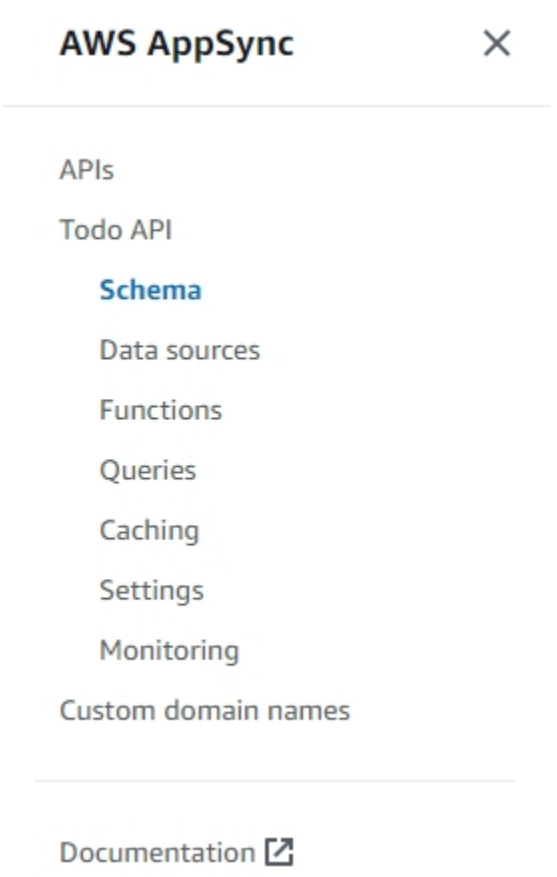
e. Choisissez Suivant.

6. Passez en revue vos modifications et choisissezCréer une API. Attendez un moment pour laisserAWS AppSyncle service termine la création de votre API.

Vous avez créé avec succès une API GraphQL avec son schéma et sa source de données DynamoDB. Pour résumer les étapes ci-dessus, nous avons choisi de créer une toute nouvelle API GraphQL. Nous avons défini le nom de l'API, puis ajouté notre définition de schéma en ajoutant notre premier type. Nous avons défini le type et ses champs, puis choisi d'associer une source de données à l'un des champs en créant une nouvelle table DynamoDB ne contenant aucune donnée.

Étape 2 : visite guidée de la console

Avant d'ajouter des données à notre table DynamoDB, nous devons passer en revue les fonctionnalités de base duAWS AppSyncexpérience sur console. LeAWS AppSyncl'onglet console sur le côté gauche de la page permet aux utilisateurs de naviguer facilement vers l'un des principaux composants ou options de configuration quiAWS AppSyncfournit :



Concepteur de schémas

ChoisissezSchémapour afficher le schéma que vous venez de créer. Si vous examinez le contenu du schéma, vous remarquerez qu'il a déjà été chargé avec de nombreuses opérations d'assistance afin de rationaliser le processus de développement. Dans leSchémaéditeur, si vous parcourez le code, vous finirez par atteindre le modèle que vous avez défini dans la section précédente :

```
type Todo {
 id: ID!
 name: String
 when: String
 where: String
 description: String
}
```

Votre modèle est devenu le type de base utilisé dans l'ensemble de votre schéma. Nous allons commencer à ajouter des données à notre source de données à l'aide de mutations générées automatiquement à partir de ce type.

Voici quelques conseils et informations supplémentaires sur leSchémaéditeur :

1. L'éditeur de code possède des fonctionnalités de linting et de vérification des erreurs que vous pouvez utiliser lorsque vous écrivez vos propres applications.
2. Le côté droit de la console affiche les types GraphQL qui ont été créés, ainsi que des résolveurs sur différents types de niveau supérieur, comme les requêtes.
3. Lorsque vous ajoutez de nouveaux types à un schéma (par exemple,`type User {...}`), vous pouvez avoirAWS AppSyncmettre à votre disposition des ressources DynamoDB. Celles-ci incluent la clé primaire, la clé de tri et la conception d'index appropriées pour correspondre au mieux à votre modèle d'accès aux données GraphQL. Si vous choisissez Créer des ressources en haut, puis l'un des types définis par l'utilisateur dans le menu, vous pouvez choisir différentes options de champ dans la conception de schéma. Nous aborderons ce sujet dans leconcevoir un schémasection.

Configuration du résolveur

Dans le concepteur de schéma,RésolveursCette section contient tous les types et champs de votre schéma. Si vous parcourez la liste des champs, vous remarquerez que vous pouvez associer des résolveurs à certains champs en choisissantJoindre. Cela ouvrira un éditeur de code dans lequel vous pourrez écrire votre code de résolution.AWS AppSyncprend en charge à la fois VTL etJavaScriptruntimes, qui peuvent être modifiés en haut de la page en choisissantActions, puisMettre à jour Runtime. Au bas de la page, vous pouvez également créer des fonctions qui exécuteront plusieurs opérations en séquence. Cependant, les résolveurs sont un sujet avancé, et nous n'en parlerons pas dans cette section.

Sources de données

ChoisissezSources de donnéespour afficher votre table DynamoDB. En choisissant le`Resource`option (si disponible), vous pouvez consulter la configuration de votre source de données. Dans notre exemple, cela mène à la console DynamoDB. À partir de là, vous pouvez modifier vos données. Vous pouvez également modifier directement certaines données en choisissant la source de données, puis en choisissantModifier. Si vous devez supprimer votre source de données, vous pouvez choisir votre source de données, puis sélectionnerSupprimer. Enfin, vous pouvez créer de nouvelles sources de données en choisissantCréation d'une source de données, puis en configurant le nom et le type. Notez que cette option permet de lierAWS AppSyncservice à une ressource existante. Vous devez tout de même créer la ressource dans votre compte en utilisant le service approprié avantAWS AppSyncle reconnaît.

Requêtes

ChoisissezRequêtespour afficher vos requêtes et vos mutations. Lorsque nous avons créé notre API GraphQL à l'aide de notre modèle,AWS AppSynca généré automatiquement des mutations et des requêtes auxiliaires à des fins de test. Dans l'éditeur de requêtes, le côté gauche contientExplorateur. Il s'agit d'une liste répertoriant toutes vos mutations et requêtes. Vous pouvez facilement activer les opérations et les champs que vous souhaitez utiliser ici en cliquant sur leurs valeurs nominales. Cela fera apparaître automatiquement le code dans la partie centrale de l'éditeur. Ici, vous pouvez modifier vos mutations et requêtes en modifiant les valeurs. Au bas de l'éditeur, vous avezVariable de requêteéditeur qui vous permet de saisir les valeurs des champs pour les variables d'entrée de vos opérations. ChoisirCouriren haut de l'éditeur, une liste déroulante s'affichera pour sélectionner la requête/mutation à exécuter. Le résultat de cette exécution apparaîtra sur le côté droit de la page. De retour dans leExplorateurdans la section supérieure, vous pouvez choisir une opération (requête, mutation, abonnement), puis choisir le+symbole pour ajouter une nouvelle instance de cette opération particulière. En haut de la page, vous trouverez une autre liste déroulante contenant le mode d'autorisation pour les exécutions de vos requêtes. Toutefois, nous n'aborderons pas cette fonctionnalité dans cette section (pour plus d'informations, voirSécurité.).

Paramètres

ChoisissezRéglagespour afficher certaines options de configuration pour votre API GraphQL. Ici, vous pouvez activer certaines options telles que la journalisation, le suivi et les fonctionnalités de pare-feu des applications Web. Vous pouvez également ajouter de nouveaux modes d'autorisation pour protéger vos données contre les fuites indésirables au public. Toutefois, ces options sont plus avancées et ne seront pas abordées dans cette section.

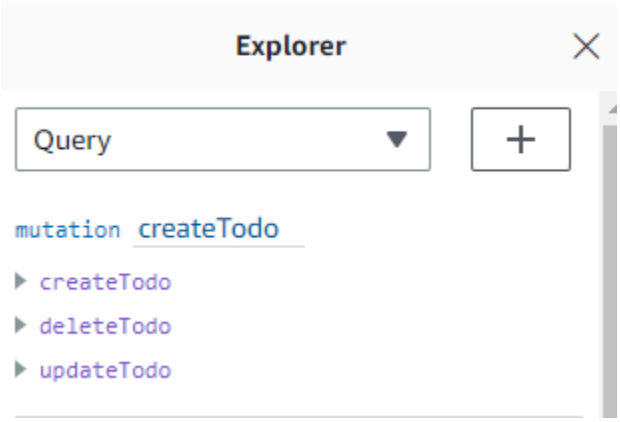
> **Note**
>
> Le mode d'autorisation par défaut,`API_KEY`, utilise une clé d'API pour tester l'application. Il s'agit de l'autorisation de base accordée à toutes les API GraphQL nouvellement créées. Nous vous recommandons d'utiliser une autre méthode de production. Pour les besoins de l'exemple présenté dans cette section, nous n'utiliserons que la clé API. Pour plus d'informations sur les méthodes d'autorisation prises en charge, voirSécurité.

Étape 3 : Ajouter des données avec une mutation GraphQL

L'étape suivante consiste à ajouter des données à votre table DynamoDB vide à l'aide d'une mutation GraphQL. Les mutations sont l'un des types d'opérations fondamentaux de GraphQL. Ils sont définis dans le schéma et vous permettent de manipuler les données de votre source de données. En termes d'API REST, elles sont très similaires à des opérations telles que`PUT`ou`POST`.

Pour ajouter des données à votre source de données

1. Si ce n'est pas déjà fait, connectez-vous auAWS Management Consoleet ouvrez leAppSyncconsole.
2. Choisissez votre API dans le tableau.
3. Dans l'onglet de gauche, choisissezRequêtes.
4. Dans leExplorateurdans l'onglet situé à gauche du tableau, vous pouvez voir plusieurs mutations et requêtes déjà définies dans l'éditeur de requêtes :



> **Note**
>
> Cette mutation est en fait inscrite dans votre schéma en tant que`Mutation`type. Il contient le code :

```
type Mutation {
 createTodo(input: CreateTodoInput!): Todo
 updateTodo(input: UpdateTodoInput!): Todo
 deleteTodo(input: DeleteTodoInput!): Todo
}
```

Comme vous pouvez le constater, les opérations ici sont similaires à celles de l'éditeur de requêtes.

AWS AppSyncles a automatiquement générés à partir du modèle que nous avons défini précédemment. Cet exemple utilisera le`createTodo`mutation pour ajouter des entrées à notre*`Todo Apitable`*table.

5. Choisissez le`createTodo`opération en l'étendant dans le cadre du`createTodo`mutation :

Cochez les cases pour tous les champs, comme sur l'image ci-dessus.

> **Note**
>
> Les attributs que vous voyez ici sont les différents éléments modifiables de la mutation. Votre`input`peut être considéré comme le paramètre de`createTodo`. Les différentes options avec des cases à cocher sont les champs qui seront renvoyés dans la réponse une fois l'opération effectuée.

6. Dans l'éditeur de code au centre de l'écran, vous remarquerez que l'opération apparaît sous le`createTodo`mutation :

```
mutation createTodo($createtodoinput: CreateTodoInput!) {
  createTodo(input: $createtodoinput) {
    where
    when
    name
```

```
    id
    description
  }
}
```

> **Note**
>
> Pour expliquer correctement cet extrait, nous devons également examiner le code du schéma. La déclaration`mutation createTodo($createtodoinput: CreateTodoInput!){}`est la mutation avec l'une de ses opérations,`createTodo`. La mutation complète se trouve dans le schéma :
>
> ```
> type Mutation {
>  createTodo(input: CreateTodoInput!): Todo
>  updateTodo(input: UpdateTodoInput!): Todo
>  deleteTodo(input: DeleteTodoInput!): Todo
> }
> ```
>
> Pour en revenir à la déclaration de mutation de l'éditeur, le paramètre est un objet appelé`$createtodoinput`avec un type d'entrée obligatoire`CreateTodoInput`. Notez que`CreateTodoInput`(et toutes les entrées de la mutation) sont également définies dans le schéma. Par exemple, voici le code standard pour`CreateTodoInput`:
>
> ```
> input CreateTodoInput {
>  name: String
>  when: String
>  where: String
>  description: String
> }
> ```
>
> Il contient les champs que nous avons définis dans notre modèle, à savoir`name`,`when`,`where`, et`description`.
> Pour en revenir au code de l'éditeur, dans`createTodo(input: $createtodoinput) {}`, nous déclarons l'entrée comme`$createtodoinput`, qui a également été utilisé dans la déclaration de mutation. Nous le faisons parce que cela permet à GraphQL de valider nos entrées par rapport aux types fournis et de s'assurer qu'elles sont utilisées avec les entrées correctes.
> La dernière partie du code de l'éditeur indique les champs qui seront renvoyés dans la réponse après l'exécution d'une opération :

```
{
    where
    when
    name
    id
    description
  }
```

Dans leVariables de requêteonglet en dessous de cet éditeur, il y aura un générique`createtodoinput`objet susceptible de contenir les données suivantes :

```
{
  "createtodoinput": {
    "name": "Hello, world!",
    "when": "Hello, world!",
    "where": "Hello, world!",
    "description": "Hello, world!"
  }
}
```

> **Note**
>
> C'est ici que nous allouons les valeurs pour l'entrée mentionnée précédemment :
>
> ```
> input CreateTodoInput {
>  name: String
>  when: String
>  where: String
>  description: String
> }
> ```

Changez le`createtodoinput`en ajoutant les informations que nous voulons mettre dans notre table DynamoDB. Dans ce cas, nous avons voulu créerTodoarticles sous forme de rappels :

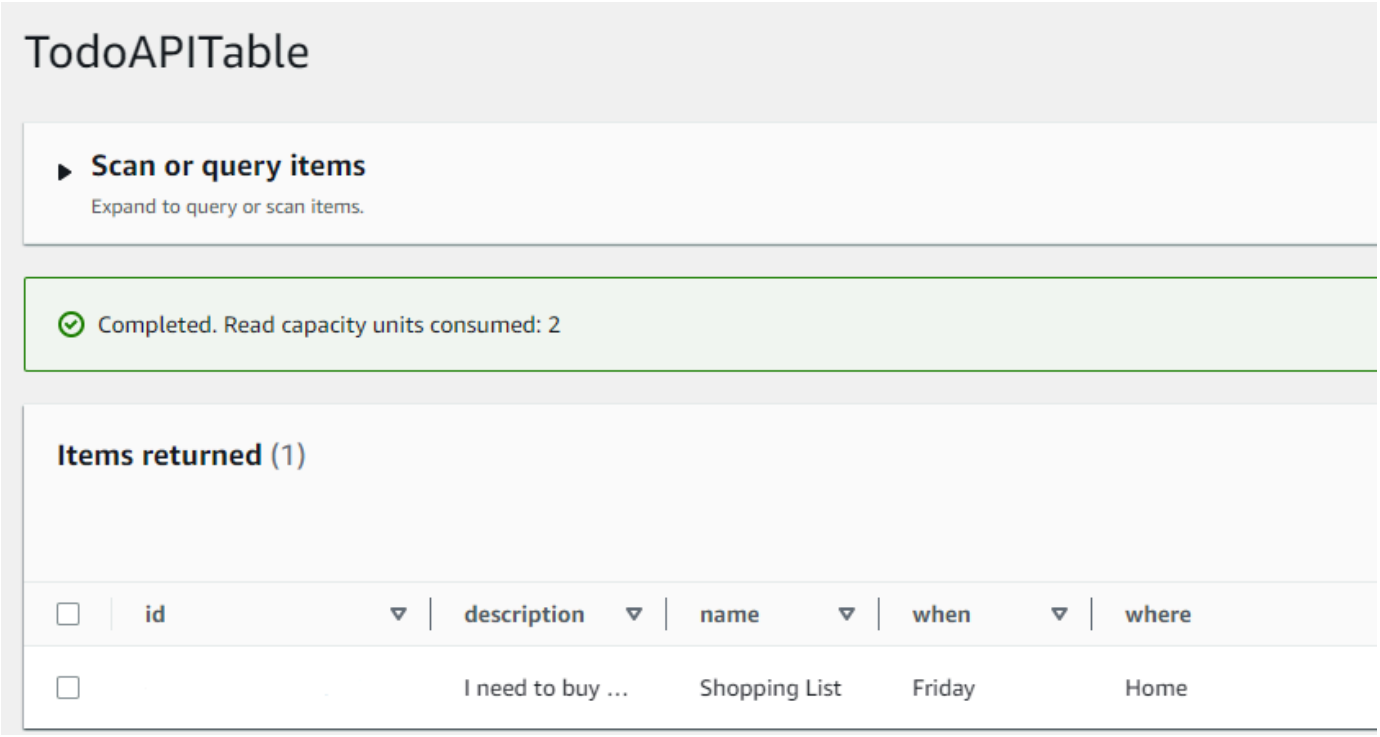
```
{
  "createtodoinput": {
```

```
    "name": "Shopping List",
    "when": "Friday",
    "where": "Home",
    "description": "I need to buy eggs"
  }
}
```

7. ChoisissezCouriren haut de l'éditeur. ChoisissezCréer Tododans la liste déroulante. Sur le côté droit de l'éditeur, vous devriez voir la réponse. Cela peut ressembler à ce qui suit :

```
{
  "data": {
    "createTodo": {
      "where": "Home",
      "when": "Friday",
      "name": "Shopping List",
      "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
      "description": "I need to buy eggs"
    }
  }
}
```

Si vous accédez au service DynamoDB, vous verrez désormais une entrée dans votre source de données contenant les informations suivantes :



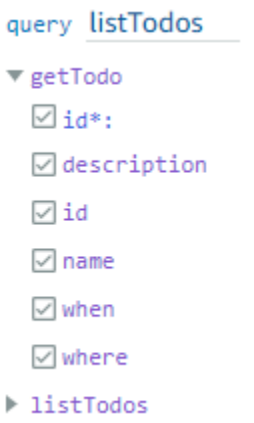
Pour résumer l'opération, le moteur GraphQL a analysé l'enregistrement et un résolveur l'a inséré dans votre table Amazon DynamoDB. Encore une fois, vous pouvez le vérifier dans la console DynamoDB. Notez que vous n'avez pas besoin de transmettre un`id`valeur. Un`id`est généré et renvoyé dans les résultats. Cela est dû au fait que l'exemple utilisait un`autoId()`fonction dans un résolveur GraphQL pour la clé de partition définie sur vos ressources DynamoDB. Nous verrons comment créer des résolveurs dans une autre section. Prenez note du retour`id`valeur ; vous l'utiliserez dans la section suivante pour récupérer des données avec une requête GraphQL.

Étape 4 : récupérer des données avec une requête GraphQL

Maintenant qu'un enregistrement existe dans votre base de données, vous obtiendrez des résultats lorsque vous exécuterez une requête. Une requête est l'une des autres opérations fondamentales de GraphQL. Il est utilisé pour analyser et récupérer des informations à partir de votre source de données. En termes d'API REST, cela est similaire à`GET`opération. Le principal avantage des requêtes GraphQL est la possibilité de spécifier les exigences exactes de votre application en matière de données afin que vous puissiez récupérer les données pertinentes au bon moment.

Pour interroger votre source de données

1. Si ce n'est pas déjà fait, connectez-vous auAWS Management Consoleet ouvrez leAppSyncconsole.
2. Choisissez votre API dans le tableau.
3. Dans l'onglet de gauche, choisissezRequêtes.
4. Dans leExplorateuronglet à gauche du tableau, sous`query listTodos`, élargissez le`getTodo`opération :



5. Dans l'éditeur de code, vous devriez voir le code d'opération :

```
query listTodos {
  getTodo(id: "") {
    description
    id
    name
    when
    where
  }
```

Dans`(id:"")`, renseignez la valeur que vous avez enregistrée dans le résultat de l'opération de mutation. Dans notre exemple, ce serait :

```
query listTodos {
  getTodo(id: "abcdefgh-1234-1234-1234-abcdefghijkl") {
    description
    id
    name
    when
    where
  }
```

6. ChoisissezCourir, puisListe Todos. Le résultat apparaîtra à droite de l'éditeur. Notre exemple ressemblait à ceci :

```
{
  "data": {
    "getTodo": {
      "description": "I need to buy eggs",
      "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
      "name": "Shopping List",
      "when": "Friday",
      "where": "Home"
    }
  }
}
```

> **Note**
>
> Les requêtes renvoient uniquement les champs que vous spécifiez. Vous pouvez désélectionner les champs dont vous n'avez pas besoin en les supprimant du champ de retour :
>
> ```
> {
>     description
>     id
>     name
>     when
>     where
>   }
> ```
>
> Vous pouvez également décocher la case dansExplorateuronglet à côté du champ que vous souhaitez supprimer.

7. Vous pouvez également essayer`listTodos`opération en répétant les étapes pour créer une entrée dans votre source de données, puis en répétant les étapes de requête avec`listTodos`opération. Voici un exemple dans lequel nous avons ajouté une deuxième tâche :

```
{
  "createtodoinput": {
    "name": "Second Task",
    "when": "Monday",
    "where": "Home",
    "description": "I need to mow the lawn"
  }
}
```

En appelant le`listTodos`opération, il a renvoyé à la fois les anciennes et les nouvelles entrées :

```
{
  "data": {
    "listTodos": {
      "items": [
        {
```

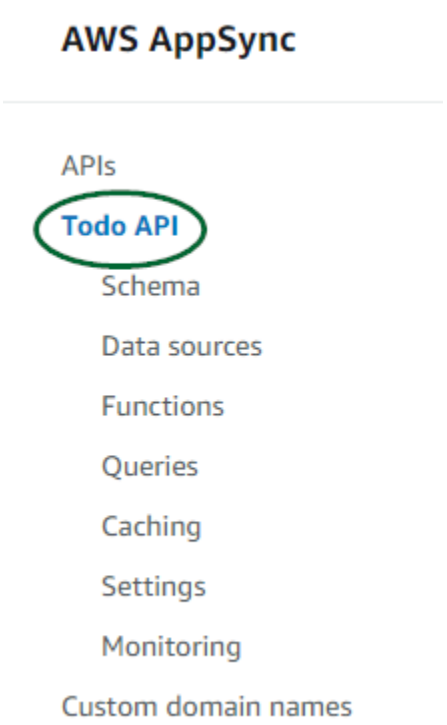
```
          "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
          "name": "Shopping List",
          "when": "Friday",
          "where": "Home",
          "description": "I need to buy eggs"
        },
        {
          "id": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
          "name": "Second Task",
          "when": "Monday",
          "where": "Home",
          "description": "I need to mow the lawn"
        }
      ]
    }
  }
}
```

Sections supplémentaires

Ces sections sont une référence pour les personnes plus avancéesAWS AppSyncsujets. Nous vous recommandons de suivre leLectures supplémentairessection avant de faire quoi que ce soit d'autre.

Integration

Dans l'onglet console, si vous choisissez le nom de votre API,Intégrationla page apparaît :



Il résume les étapes de configuration de votre API et décrit les prochaines étapes de création d'une application client. LeIntégrez à votre applicationcette section fournit des informations détaillées sur l'utilisation duAWSAmplifier la chaîne d'outilspour automatiser le processus de connexion de votre API à iOS, Android etJavaScriptapplications via la configuration et la génération de code. La chaîne d'outils Amplify fournit un support complet pour la création de projets à partir de votre station de travail locale, y compris le provisionnement GraphQL et les flux de travail pour CI/CD.

LeExemples de clientscette section répertorie également des exemples d'applications clientes (par exemple,JavaScript, iOS, Android) pour tester une expérience de bout en bout. Vous pouvez cloner et télécharger ces exemples, et le fichier de configuration contient les informations nécessaires (telles que l'URL de votre point de terminaison) dont vous avez besoin pour démarrer. Suivez les instructions figurant sur leAWS Amplifychaîne d'outilspage pour exécuter votre application.

Lectures supplémentaires

- Conception d'API GraphQL- Il s'agit d'un guide complet pour créer votre GraphQL à l'aide d'un schéma vide sans sources de données ni résolveurs.

Conception d'API GraphQL

AWS AppSyncvous permet de créer des API GraphQL à l'aide de l'expérience de la console. Vous en avez eu un aperçu dans la section Lancement d'un exemple de schéma. Cependant, ce guide ne présentait pas le catalogue complet des options et des configurations que vous pouviez exploiterAWS AppSync.

Lorsque vous choisissez de créer une API GraphQL dans la console, plusieurs options s'offrent à vous. Si vous avez suivi notre guide de lancement d'un exemple de schéma, nous vous avons montré comment créer une API à partir d'un modèle prédéfini. Dans les sections suivantes, nous vous expliquerons les autres options et configurations permettant de créer des API GraphQL dans. AWS AppSync

Dans cette section, vous allez passer en revue les concepts suivants :

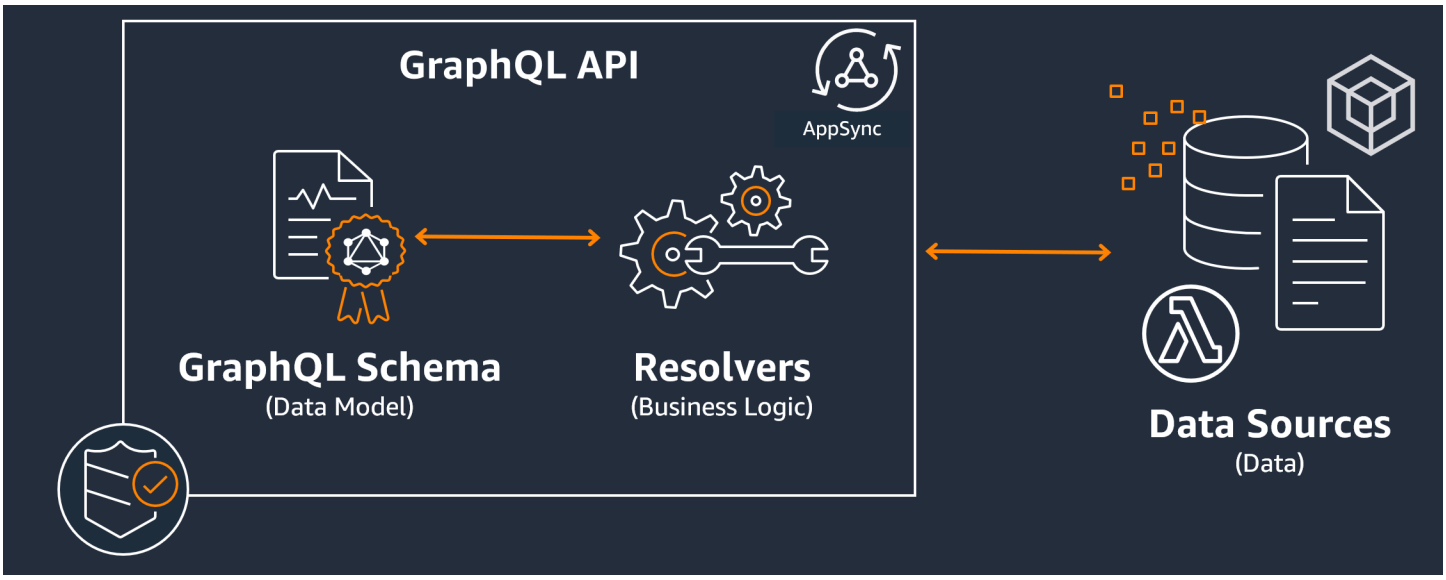
1. Blank APIs or imports: Ce guide décrit l'intégralité du processus de création d'une API GraphQL. Vous apprendrez à créer un GraphQL à partir d'un modèle vierge sans modèle, à configurer des sources de données pour votre schéma et à ajouter votre premier résolveur à un champ.
2. Real-time data: Ce guide vous montrera les options potentielles pour créer une API à l'aide AWS AppSync du WebSocket moteur.
3. Merged APIs: Ce guide explique comment créer de nouvelles API GraphQL en associant et en fusionnant des données provenant de plusieurs API GraphQL existantes.
4. the section called “Introspection RDS”: Ce guide explique comment intégrer vos tables Amazon RDS à l'aide d'une API de données.

Structuration d'une API GraphQL (API vides ou importées)

Avant de créer votre API GraphQL à partir d'un modèle vierge, il serait utile de passer en revue les concepts qui entourent GraphQL. Une API GraphQL comporte trois composants fondamentaux :

1. Leschémaest le fichier contenant la forme et la définition de vos données. Lorsqu'une demande est envoyée par un client à votre service GraphQL, les données renvoyées suivent les spécifications du schéma. Pour plus d'informations, veuillez consulter Schémas.
2. Lesource de donnéesest attaché à votre schéma. Lorsqu'une demande est faite, c'est ici que les données sont récupérées et modifiées. Pour plus d'informations, veuillez consulter Data sources.

3. Lerésolveurse situe entre le schéma et la source de données. Lorsqu'une demande est faite, le résolveur effectue l'opération sur les données de la source, puis renvoie le résultat sous forme de réponse. Pour plus d'informations, veuillez consulter Resolvers.



AWS AppSyncgère vos API en vous permettant de créer, de modifier et de stocker le code de vos schémas et résolveurs. Vos sources de données proviendront de référentiels externes tels que des bases de données, des tables DynamoDB et des fonctions Lambda. Si vous utilisez unAWSservice pour stocker vos données ou planifiez de le faire,AWS AppSyncfournit une expérience quasi fluide lors de l'association de données provenant de votreAWScomptes vers vos API GraphQL.

Dans la section suivante, vous allez apprendre à créer chacun de ces composants à l'aide duAWS AppSyncservice.

Rubriques

- Étape 1 : Conception de votre schéma
- Étape 2 : Joindre une source de données
- Étape 3 : Configuration des résolveurs
- Étape 4 : Utilisation d'une API : exemple de CDK

Étape 1 : Conception de votre schéma

Le schéma GraphQL est la base de toute implémentation de serveur GraphQL. Chaque API GraphQL est définie par unseulschéma qui contient des types et des champs décrivant la manière

dont les données des demandes seront renseignées. Les données qui transitent par votre API et les opérations effectuées doivent être validées par rapport au schéma.

En général, lesystème de type GraphQLdécrit les fonctionnalités d'un serveur GraphQL et est utilisé pour déterminer si une requête est valide. Le système de types d'un serveur est souvent appelé schéma de ce serveur et peut être composé de différents types d'objets, de types scalaires, de types d'entrée, etc. GraphQL est à la fois déclaratif et fortement typé, ce qui signifie que les types seront bien définis lors de l'exécution et ne renverront que ce qui a été spécifié.

AWS AppSyncvous permet de définir et de configurer des schémas GraphQL. La section suivante décrit comment créer des schémas GraphQL à partir de zéro en utilisantAWS AppSyncles services de.

Structuration d'un schéma GraphQL

> **Tip**
>
> Nous vous recommandons de consulter leSchémassection avant de continuer.

GraphQL est un outil puissant pour implémenter des services d'API. SelonLe site web de GraphQL, GraphQL est le suivant :

«GraphQL est un langage de requête pour les API et un environnement d'exécution permettant de répondre à ces requêtes avec vos données existantes. GraphQL fournit une description complète et compréhensible des données de votre API, donne aux clients le pouvoir de demander exactement ce dont ils ont besoin, rien de plus, facilite l'évolution des API au fil du temps et fournit de puissants outils de développement.«

Cette section couvre la toute première partie de votre implémentation GraphQL, le schéma. En utilisant la citation ci-dessus, un schéma joue le rôle de « fournir une description complète et compréhensible des données de votre API ». En d'autres termes, un schéma GraphQL est une représentation textuelle des données, des opérations et des relations entre les données de votre service. Le schéma est considéré comme le point d'entrée principal pour l'implémentation de votre service GraphQL. Comme on pouvait s'y attendre, c'est souvent l'une des premières choses que vous réalisez dans votre projet. Nous vous recommandons de consulter leSchémassection avant de continuer.

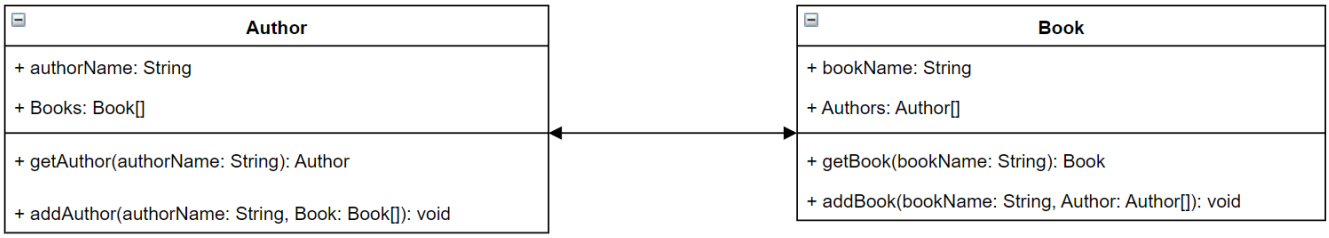
Pour citer leSchémassection, les schémas GraphQL sont écrits dansLangage de définition du schéma(SDL). SDL est composé de types et de champs dotés d'une structure établie :

- Les types: Les types sont la façon dont GraphQL définit la forme et le comportement des données. GraphQL prend en charge une multitude de types qui seront expliqués plus loin dans cette section. Chaque type défini dans votre schéma contiendra sa propre portée. Le champ d'application comportera un ou plusieurs champs pouvant contenir une valeur ou une logique qui sera utilisée dans votre service GraphQL. Les types remplissent de nombreux rôles différents, les plus courants étant les objets ou les scalaires (types de valeurs primitives).
- Champs: Les champs existent dans le cadre d'un type et contiennent la valeur demandée au service GraphQL. Elles sont très similaires aux variables d'autres langages de programmation. La forme des données que vous définissez dans vos champs déterminera la manière dont les données sont structurées lors d'une opération de demande/réponse. Cela permet aux développeurs de prévoir ce qui sera renvoyé sans savoir comment le backend du service est implémenté.

Les schémas les plus simples contiendront trois catégories de données différentes :

1. Racines du schéma: Les racines définissent les points d'entrée de votre schéma. Il indique les champs qui effectueront certaines opérations sur les données, telles que l'ajout, la suppression ou la modification de quelque chose.
2. Les types: il s'agit de types de base utilisés pour représenter la forme des données. Vous pouvez presque les considérer comme des objets ou des représentations abstraites de quelque chose avec des caractéristiques définies. Par exemple, vous pouvez créer un`Person`objet qui représente une personne dans une base de données. Les caractéristiques de chaque personne seront définies dans le`Person`sous forme de champs. Ils peuvent être n'importe quoi comme le nom, l'âge, le travail, l'adresse de la personne, etc.
3. Types d'objets spéciaux: ce sont les types qui définissent le comportement des opérations dans votre schéma. Chaque type d'objet spécial est défini une fois par schéma. Ils sont d'abord placés dans la racine du schéma, puis définis dans le corps du schéma. Chaque champ d'un type d'objet spécial définit une opération unique à implémenter par votre résolveur.

Pour mettre les choses en perspective, imaginez que vous créez un service qui stocke les auteurs et les livres qu'ils ont écrits. Chaque auteur a un nom et une liste de livres qu'il a écrits. Chaque livre a un nom et une liste d'auteurs associés. Nous voulons également pouvoir ajouter ou récupérer des livres et des auteurs. Une représentation UML simple de cette relation peut ressembler à ceci :



Dans GraphQL, les entitésAuthoretBookreprésentez deux types d'objets différents dans votre schéma :

```
type Author {
}

type Book {
}
```

AuthorcontientauthorNameetBooks, tandis queBookcontientbookNameetAuthors. Ceux-ci peuvent être représentés sous forme de champs correspondant à vos types :

```
type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}
```

Comme vous pouvez le constater, les représentations de type sont très proches du diagramme. Cependant, c'est dans les méthodes que cela devient un peu plus délicat. Ils seront placés dans l'un des rares types d'objets spéciaux sous forme de champ. La catégorisation des objets spéciaux dépend de leur comportement. GraphQL contient trois types d'objets spéciaux fondamentaux : les requêtes, les mutations et les abonnements. Pour plus d'informations, voirObjets spéciaux.

Parce quegetAuthoretgetBookdemandent tous deux des données, ils seront placés dans unQuerytype d'objet spécial :

```
type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}

type Query {
  getAuthor(authorName: String): Author
  getBook(bookName: String): Book
}
```

Les opérations sont liées à la requête, elle-même liée au schéma. L'ajout d'une racine de schéma définira le type d'objet spécial (`Query`dans ce cas) comme l'un de vos points d'entrée. Cela peut être fait à l'aide du`schema`mot clé :

```
schema {
  query: Query
}

type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}

type Query {
  getAuthor(authorName: String): Author
  getBook(bookName: String): Book
}
```

En ce qui concerne les deux dernières méthodes,`addAuthor`et`addBook`ajoutent des données à votre base de données, elles seront donc définies dans un`Mutation`type d'objet spécial. Toutefois, à partir duLes typespage, nous savons également que les entrées faisant directement référence à

des objets ne sont pas autorisées car ce sont strictement des types de sortie. Dans ce cas, nous ne pouvons pas utiliser`Author`ou`Book`, nous devons donc créer un type d'entrée avec les mêmes champs. Dans cet exemple, nous avons ajouté`AuthorInput`et`BookInput`, qui acceptent tous deux les mêmes champs de leur type respectif. Ensuite, nous créons notre mutation en utilisant les entrées comme paramètres :

```
schema {
  query: Query
  mutation: Mutation
}

type Author {
  authorName: String
  Books: [Book]
}

input AuthorInput {
  authorName: String
  Books: [BookInput]
}

type Book {
  bookName: String
  Authors: [Author]
}

input BookInput {
  bookName: String
  Authors: [AuthorInput]
}

type Query {
  getAuthor(authorName: String): Author
  getBook(bookName: String): Book
}

type Mutation {
  addAuthor(input: [BookInput]): Author
  addBook(input: [AuthorInput]): Book
}
```

Passons en revue ce que nous venons de faire :

1. Nous avons créé un schéma avec`Booket``Author`types pour représenter nos entités.
2. Nous avons ajouté les champs contenant les caractéristiques de nos entités.
3. Nous avons ajouté une requête pour récupérer ces informations dans la base de données.
4. Nous avons ajouté une mutation pour manipuler les données de la base de données.
5. Nous avons ajouté des types d'entrée pour remplacer les paramètres de nos objets dans la mutation afin de respecter les règles de GraphQL.
6. Nous avons ajouté la requête et la mutation à notre schéma racine afin que l'implémentation de GraphQL comprenne l'emplacement du type racine.

Comme vous pouvez le constater, le processus de création d'un schéma repose sur de nombreux concepts issus de la modélisation des données (en particulier de la modélisation de base de données) en général. Vous pouvez considérer le schéma comme s'adaptant à la forme des données provenant de la source. Il sert également de modèle que le résolveur implémentera. Dans les sections suivantes, vous apprendrez à créer un schéma à l'aide de différentsAWS-outils et services soutenus.

> **Note**
>
> Les exemples présentés dans les sections suivantes ne sont pas destinés à être exécutés dans une application réelle. Ils ne sont là que pour présenter les commandes afin que vous puissiez créer vos propres applications.

Création de schémas

Votre schéma se trouvera dans un fichier appelé`schema.graphql`.AWS AppSyncpermet aux utilisateurs de créer de nouveaux schémas pour leurs API GraphQL à l'aide de différentes méthodes. Dans cet exemple, nous allons créer une API vide avec un schéma vide.

Console

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
   a. Dans le Tableau de bord, choisissez Créer une API.
   b. En dessousOptions d'API, choisissezAPI GraphQL,Design à partir de zéro, puisSuivant.

i. PourNom de l'API, modifiez le nom prérempli en fonction des besoins de votre application.

ii. Pourcoordonnées, vous pouvez saisir un point de contact pour identifier le responsable de l'API. Il s'agit d'un champ facultatif.

iii. En dessousConfiguration de l'API privée, vous pouvez activer les fonctionnalités d'API privées. Une API privée n'est accessible qu'à partir d'un point de terminaison VPC configuré (VPCE). Pour plus d'informations, voirAPI privées.

Nous ne recommandons pas d'activer cette fonctionnalité pour cet exemple. ChoisissezSuivantaprès avoir examiné vos contributions.

c. En dessousCréation d'un type GraphQL, vous pouvez choisir de créer une table DynamoDB à utiliser comme source de données ou de l'ignorer et de le faire plus tard.

Pour cet exemple, choisissezCréez des ressources GraphQL ultérieurement. Nous allons créer une ressource dans une section séparée.

d. Passez en revue vos entrées, puis choisissezCréer une API.

2. Vous serez dans le tableau de bord de votre API spécifique. Vous pouvez le savoir, car le nom de l'API figurera en haut du tableau de bord. Si ce n'est pas le cas, vous pouvez sélectionnerAPIsdans leBarre latérale, puis choisissez votre API dansTableau de bord des API.

- Dans leBarre latéralesous le nom de votre API, choisissezSchéma.

3. Dans leEditeur de schéma, vous pouvez configurer votre`schema.graphql`fichier. Il peut être vide ou rempli de types générés à partir d'un modèle. Sur la droite, vous avezRésolveurssection pour joindre des résolveurs aux champs de votre schéma. Nous n'examinerons pas les résolveurs dans cette section.

CLI

> **Note**
>
> Lorsque vous utilisez l'interface de ligne de commande, assurez-vous de disposer des autorisations appropriées pour accéder au service et en créer des ressources. Vous souhaiterez peut-être définirmoindre privilègepolitiques pour les utilisateurs non administrateurs qui ont besoin d'accéder au service. Pour plus d'informations surAWS AppSyncpolitiques, voirGestion des identités et des accès pourAWS AppSync.

En outre, nous vous recommandons de lire d'abord la version console si ce n'est pas déjà fait.

1. Si ce n'est pas déjà fait,installerleAWSCLI, puis ajoutez votreconfiguration.
2. Créez un objet d'API GraphQL en exécutant le`create-graphql-api`commande.

   Vous devez saisir deux paramètres pour cette commande particulière :

   1. Le`name`de votre API.
   2. Le`authentication-type`, ou le type d'identifiant utilisé pour accéder à l'API (IAM, OIDC, etc.).

   > **Note**
   >
   > D'autres paramètres tels que`Region`doit être configuré mais utilise généralement par défaut les valeurs de configuration de votre CLI.

   Voici un exemple de commande :

   ```
   aws appsync create-graphql-api --name testAPI123 --authentication-type API_KEY
   ```

   Une sortie sera renvoyée dans la CLI. Voici un exemple:

   ```
   {
       "graphqlApi": {
           "xrayEnabled": false,
           "name": "testAPI123",
           "authenticationType": "API_KEY",
           "tags": {},
           "apiId": "abcdefghijklmnopqrstuvwxyz",
           "uris": {
               "GRAPHQL": "https://zyxwvutsrqponmlkjihgfedcba.appsync-api.us-
   west-2.amazonaws.com/graphql",
               "REALTIME": "wss://zyxwvutsrqponmlkjihgfedcba.appsync-realtime-
   api.us-west-2.amazonaws.com/graphql"
           },
   ```

```
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz"
    }
}
```

3.

> **Note**
>
> Il s'agit d'une commande facultative qui prend un schéma existant et le télécharge dansAWS AppSyncservice utilisant un blob en base 64. Nous n'utiliserons pas cette commande pour les besoins de cet exemple.

Exécutez la commande `start-schema-creation`.

Vous devez saisir deux paramètres pour cette commande particulière :

1. Votre`api-id`à partir de l'étape précédente.
2. Le schéma`definition`est un blob binaire codé en base 64.

Voici un exemple de commande :

```
 aws appsync start-schema-creation --api-id abcdefghijklmnopqrstuvwxyz --
definition "aa1111aa-123b-2bb2-c321-12hgg76cc33v"
```

Une sortie sera renvoyée :

```
{
    "status": "PROCESSING"
}
```

Cette commande ne renverra pas le résultat final après le traitement. Vous devez utiliser une commande séparée,`get-schema-creation-status`, pour voir le résultat. Notez que ces deux commandes sont asynchrones. Vous pouvez donc vérifier l'état de sortie même lorsque le schéma est encore en cours de création.

CDK

> **Tip**
>
> Avant d'utiliser le CDK, nous vous recommandons de consulter les CDKdocumentation officielleainsi queAWS AppSyncc'estRéférence CDK.
> Les étapes répertoriées ci-dessous ne montreront qu'un exemple général de l'extrait de code utilisé pour ajouter une ressource particulière. C'estpascensé être une solution fonctionnelle dans votre code de production. Nous partons également du principe que vous disposez déjà d'une application fonctionnelle.

1. Le point de départ du CDK est un peu différent. Idéalement, votre`schema.graphql`le fichier devrait déjà être créé. Il vous suffit de créer un nouveau fichier avec`.graphql`extension de fichier. Il peut s'agir d'un fichier vide.
2. En général, vous devrez peut-être ajouter la directive d'importation au service que vous utilisez. Par exemple, il peut suivre les formes suivantes :

   ```
   import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'
   import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
   ```

   Pour ajouter une API GraphQL, votre fichier de pile doit importer leAWS AppSyncservice :

   ```
   import * as appsync from 'aws-cdk-lib/aws-appsync';
   ```

   > **Note**
   >
   > Cela signifie que nous importons l'ensemble du service sous le`appsync`mot clé. Pour l'utiliser dans votre application, votreAWS AppSyncles constructions utiliseront le format`appsync.construct_name`. Par exemple, si nous voulions créer une API GraphQL, nous dirions`new appsync.GraphqlApi(args_go_here)`. L'étape suivante illustre cela.

3. L'API GraphQL la plus basique inclura un`name`pour l'API et le`schema`chemin.

   ```
   const add_api = new appsync.GraphqlApi(this, 'API_ID', {
     name: 'name_of_API_in_console',
   ```

```
  schema: appsync.SchemaFile.fromAsset(path.join(__dirname,
 'schema_name.graphql')),
});
```

> **Note**
>
> Voyons ce que fait cet extrait. Dans le cadre de`api`, nous créons une nouvelle API GraphQL en appelant`appsync.GraphqlApi(scope: Construct, id: string, props: GraphqlApiProps)`. Le champ d'application est`this`, qui fait référence à l'objet actuel. L'identifiant est*`API_ID`*, qui sera le nom de la ressource de votre API GraphQL dansAWS CloudFormationquand il est créé. Le`GraphqlApiProps`contient le`name`de votre API GraphQL et du`schema`. Le`schema`générera un schéma (`SchemaFile.fromAsset`) en recherchant le chemin absolu (`__dirname`) pour le`.graphql`fichier (*`nom_schéma.graphql`*). Dans un scénario réel, votre fichier de schéma se trouvera probablement dans l'application CDK.
>
> Pour utiliser les modifications apportées à votre API GraphQL, vous devez redéployer l'application.

Ajouter des types aux schémas

Maintenant que vous avez ajouté votre schéma, vous pouvez commencer à ajouter vos types d'entrée et de sortie. Notez que les types présentés ici ne doivent pas être utilisés dans le code réel ; ce ne sont que des exemples destinés à vous aider à comprendre le processus.

Nous allons d'abord créer un type d'objet. Dans le code réel, il n'est pas nécessaire de commencer par ces types. Vous pouvez créer le type que vous voulez à tout moment, à condition de respecter les règles et la syntaxe de GraphQL.

> **Note**
>
> Les prochaines sections utiliseront leéditeur de schéma, alors gardez-le ouvert.

Console

- Vous pouvez créer un type d'objet à l'aide du`type`mot-clé avec le nom du type :

```
type Type_Name_Goes_Here {}
```

Dans le champ d'application du type, vous pouvez ajouter des champs qui représentent les caractéristiques de l'objet :

```
type Type_Name_Goes_Here {
  # Add fields here
}
```

Voici un exemple:

```
type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}
```

> **Note**
>
> Dans cette étape, nous avons ajouté un type d'objet générique avec un `id` champ stocké sous `ID`, un `title` champ stocké sous forme de `String`, et un `date` champ stocké sous forme de `AWSDateTime`. Pour consulter la liste des types et des champs et leur fonction, voir Schémas. Pour consulter la liste des scalaires et leur fonction, consultez le Référence de type.

CLI

> **Note**
>
> Nous vous recommandons de lire d'abord la version console si ce n'est pas déjà fait.

- Vous pouvez créer un type d'objet en exécutant `create-type` commande.

  Vous devez entrer quelques paramètres pour cette commande particulière :

  1. Le `api-id` de votre API.

2. Le`definition`, ou le contenu de votre type. Dans l'exemple de console, c'était :

```
type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}
```

3. Le`format`de votre contribution. Dans cet exemple, nous utilisonsSDL.

Voici un exemple de commande :

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type
 Obj_Type_1{id: ID! title: String date: AWSDateTime}" --format SDL
```

Une sortie sera renvoyée dans la CLI. Voici un exemple:

```
{
    "type": {
        "definition": "type Obj_Type_1{id: ID! title: String date:
 AWSDateTime}",
        "name": "Obj_Type_1",
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Obj_Type_1",
        "format": "SDL"
    }
}
```

> **Note**
>
> Dans cette étape, nous avons ajouté un type d'objet générique avec un`id`champ stocké sous`ID`, un`title`champ stocké sous forme de`String`, et un`date`champ stocké sous forme de`AWSDateTime`. Pour consulter la liste des types et des champs et leur fonction, voirSchémas. Pour consulter la liste des scalaires et leur fonction, voirRéférence de type.
>
> Par ailleurs, vous vous êtes peut-être rendu compte que la saisie de la définition fonctionne directement pour les types plus petits, mais qu'elle est impossible pour

l'ajout de types plus grands ou multiples. Vous pouvez choisir de tout ajouter dans un `.graphql`fichier, puispassez-le en entrée.

CDK

> **Tip**
>
> Avant d'utiliser le CDK, nous vous recommandons de consulter les CDKdocumentation officielleainsi queAWS AppSyncc'estRéférence CDK.
> Les étapes répertoriées ci-dessous ne montreront qu'un exemple général de l'extrait de code utilisé pour ajouter une ressource particulière. C'estpascensé être une solution fonctionnelle dans votre code de production. Nous partons également du principe que vous disposez déjà d'une application fonctionnelle.

Pour ajouter un type, vous devez l'ajouter à votre `.graphql`fichier. Par exemple, l'exemple de console était le suivant :

```
type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}
```

Vous pouvez ajouter vos types directement au schéma comme n'importe quel autre fichier.

> **Note**
>
> Pour utiliser les modifications apportées à votre API GraphQL, vous devez redéployer l'application.

Letype d'objetcomporte des champs qui sonttypes scalairestels que des chaînes et des entiers.AWS AppSyncvous permet également d'utiliser des types scalaires améliorés tels que`AWSDateTime`en plus des scalaires GraphQL de base. En outre, tout champ se terminant par un point d'exclamation est obligatoire.

LeIDle type scalaire en particulier est un identifiant unique qui peut être soitStringouInt. Vous pouvez les contrôler dans le code de votre résolveur pour une attribution automatique.

Il existe des similitudes entre les types d'objets spéciaux tels queQueryet des types d'objets « normaux » comme dans l'exemple ci-dessus, dans la mesure où ils utilisent tous les deux letypemots clés et sont considérés comme des objets. Toutefois, pour les types d'objets spéciaux (Query,Mutation, etSubscription), leur comportement est très différent car ils sont exposés en tant que points d'entrée de votre API. Ils visent également à façonner les opérations plutôt que les données. Pour plus d'informations, voirLes types de requêtes et de mutations.

En ce qui concerne les types d'objets spéciaux, l'étape suivante pourrait consister à en ajouter un ou plusieurs pour effectuer des opérations sur les données mises en forme. Dans un scénario réel, chaque schéma GraphQL doit au moins avoir un type de requête racine pour demander des données. Vous pouvez considérer la requête comme l'un des points d'entrée (ou points de terminaison) de votre serveur GraphQL. Ajoutons une requête à titre d'exemple.

Console

- Pour créer une requête, vous pouvez simplement l'ajouter au fichier de schéma comme n'importe quel autre type. Une requête nécessiteraitQuerytapez et une entrée dans la racine comme ceci :

```
schema {
  query: Name_of_Query
}

type Name_of_Query {
  # Add field operation here
}
```

Notez que*Nom de la requête*dans un environnement de production sera simplement appeléQuerydans la plupart des cas. Nous vous recommandons de le maintenir à cette valeur. Dans le type de requête, vous pouvez ajouter des champs. Chaque champ effectuera une opération dans la demande. Par conséquent, la plupart de ces champs, sinon tous, seront attachés à un résolveur. Cependant, cela ne nous intéresse pas dans cette section. En ce qui concerne le format de l'opération sur le terrain, cela pourrait ressembler à ceci :

```
Name_of_Query(params): Return_Type # version with params
Name_of_Query: Return_Type # version without params
```

Voici un exemple:

```
schema {
  query: Query
}

type Query {
  getObj: [Obj_Type_1]
}

type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}
```

> **Note**
>
> Au cours de cette étape, nous avons ajouté un`Query`saisissez-le et définissez-le dans notre`schema`racine. Notre`Query`type défini a`getObj`champ qui renvoie une liste de`Obj_Type_1`objets. Notez que`Obj_Type_1`est l'objet de l'étape précédente. Dans le code de production, vos opérations sur le terrain travailleront normalement avec des données façonnées par des objets tels que`Obj_Type_1`. En outre, des champs tels que`getObj`aura normalement un résolveur pour exécuter la logique métier. Cela sera traité dans une autre section.
>
> À titre de note supplémentaire,AWS AppSyncajoute automatiquement une racine de schéma lors des exportations. Techniquement, vous n'avez donc pas besoin de l'ajouter directement au schéma. Notre service traitera automatiquement les schémas dupliqués. Nous l'ajoutons ici en tant que meilleure pratique.

CLI

> **Note**
>
> Nous vous recommandons de lire d'abord la version console si ce n'est pas déjà fait.

1. Créez un `schema` racine avec un `query` définition en exécutant le `create-type` commande.

   Vous devez entrer quelques paramètres pour cette commande particulière :

   1. Le `api-id` de votre API.
   2. Le `definition`, ou le contenu de votre type. Dans l'exemple de console, c'était :

      ```
      schema {
        query: Query
      }
      ```

   3. Le `format` de votre contribution. Dans cet exemple, nous utilisons `SDL`.

   Voici un exemple de commande :

   ```
   aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "schema
    {query: Query}" --format SDL
   ```

   Une sortie sera renvoyée dans la CLI. Voici un exemple:

   ```
   {
       "type": {
           "definition": "schema {query: Query}",
           "name": "schema",
           "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
   abcdefghijklmnopqrstuvwxyz/types/schema",
           "format": "SDL"
       }
   }
   ```

   > **Note**
   >
   > Notez que si vous n'avez pas saisi correctement quelque chose dans le `create-type`, vous pouvez mettre à jour la racine de votre schéma (ou n'importe quel type du schéma) en exécutant `update-type` commande. Dans cet exemple, nous allons modifier temporairement la racine du schéma pour qu'elle contienne un `subscription` définition.
   > Vous devez entrer quelques paramètres pour cette commande particulière :

1. Le`api-id`de votre API.
2. Le`type-name`de votre type. Dans l'exemple de console, c'était`schema`.
3. Le`definition`, ou le contenu de votre type. Dans l'exemple de console, c'était :

```
schema {
  query: Query
}
```

Le schéma après l'ajout d'un`subscription`ressemblera à ceci :

```
schema {
  query: Query
  subscription: Subscription
}
```

4. Le`format`de votre contribution. Dans cet exemple, nous utilisons`SDL`.

Voici un exemple de commande :

```
aws appsync update-type --api-id abcdefghijklmnopqrstuvwxyz --type-name
 schema --definition "schema {query: Query subscription: Subscription}"
 --format SDL
```

Une sortie sera renvoyée dans la CLI. Voici un exemple:

```
{
    "type": {
        "definition": "schema {query: Query subscription: Subscription}",
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/schema",
        "format": "SDL"
    }
}
```

L'ajout de fichiers préformatés fonctionnera toujours dans cet exemple.

2. Créez un`Query`tapez en exécutant`create-type`commande.

Vous devez entrer quelques paramètres pour cette commande particulière :

1. Le`api-id`de votre API.
2. Le`definition`, ou le contenu de votre type. Dans l'exemple de console, c'était :

```
type Query {
  getObj: [Obj_Type_1]
}
```

3. Le`format`de votre contribution. Dans cet exemple, nous utilisonsSDL.

Voici un exemple de commande :

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type
 Query {getObj: [Obj_Type_1]}" --format SDL
```

Une sortie sera renvoyée dans la CLI. Voici un exemple:

```
{
    "type": {
        "definition": "Query {getObj: [Obj_Type_1]}",
        "name": "Query",
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Query",
        "format": "SDL"
    }
}
```

> **Note**
>
> Au cours de cette étape, nous avons ajouté un`Query`saisissez-le et définissez-le dans votre`schema`racine. Notre`Query`type défini a`getObj`champ qui a renvoyé une liste de`Obj_Type_1`objets.
> Dans le`schema`code racine`query: Query`, le`query:`la partie indique qu'une requête a été définie dans votre schéma, tandis que le`Query`part indique le nom réel de l'objet spécial.

CDK

> **Tip**
>
> Avant d'utiliser le CDK, nous vous recommandons de consulter les CDKdocumentation officielleainsi queAWS AppSyncc'estRéférence CDK.
> Les étapes répertoriées ci-dessous ne montreront qu'un exemple général de l'extrait de code utilisé pour ajouter une ressource particulière. C'estpascensé être une solution fonctionnelle dans votre code de production. Nous partons également du principe que vous disposez déjà d'une application fonctionnelle.

Vous devez ajouter votre requête et la racine du schéma dans `.graphql`fichier. Notre exemple ressemble à l'exemple ci-dessous, mais vous devez le remplacer par votre code de schéma actuel :

```
schema {
  query: Query
}

type Query {
  getObj: [Obj_Type_1]
}

type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}
```

Vous pouvez ajouter vos types directement au schéma comme n'importe quel autre fichier.

> **Note**
>
> La mise à jour de la racine du schéma est facultative. Nous l'avons ajoutée à cet exemple en tant que meilleure pratique.
> Pour utiliser les modifications apportées à votre API GraphQL, vous devez redéployer l'application.

Vous avez maintenant vu un exemple de création d'objets et d'objets spéciaux (requêtes). Vous avez également vu comment ils peuvent être interconnectés pour décrire les données et les opérations. Vous pouvez avoir des schémas contenant uniquement la description des données et une ou plusieurs requêtes. Cependant, nous aimerions ajouter une autre opération pour ajouter des données à la source de données. Nous allons ajouter un autre type d'objet spécial appelé`Mutation`qui modifie les données.

Console

- Une mutation sera appelée`Mutation`. Comme`Query`, les opérations sur le terrain à l'intérieur`Mutation`décrira une opération et sera attaché à un résolveur. Notez également que nous devons le définir dans`schema`root car il s'agit d'un type d'objet spécial. Voici un exemple de mutation :

  ```
  schema {
    mutation: Name_of_Mutation
  }

  type Name_of_Mutation {
    # Add field operation here
  }
  ```

  Une mutation typique sera répertoriée à la racine comme une requête. La mutation est définie à l'aide du`type`mot clé avec le nom. *`Nom de la mutation`*sera généralement appelé`Mutation`, nous vous recommandons donc de continuer ainsi. Chaque champ effectuera également une opération. En ce qui concerne le format de l'opération sur le terrain, cela pourrait ressembler à ceci :

  ```
  Name_of_Mutation(params): Return_Type # version with params
  Name_of_Mutation: Return_Type # version without params
  ```

  Voici un exemple:

  ```
  schema {
    query: Query
    mutation: Mutation
  }

  type Obj_Type_1 {
  ```

```
  id: ID!
  title: String
  date: AWSDateTime
}

type Query {
  getObj: [Obj_Type_1]
}

type Mutation {
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
}
```

Note

Au cours de cette étape, nous avons ajouté un`Mutation`ontapez avec un`addObj`champ. Résumons le rôle de ce champ :

```
addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
```

`addObj`utilise le`Obj_Type_1`objet pour effectuer une opération. Cela est évident en raison des champs, mais la syntaxe le prouve dans le`: Obj_Type_1`type de retour. À l'intérieur`addObj`, il accepte le`id`,`title`, et`date`champs du`Obj_Type_1`objet en tant que paramètres. Comme vous pouvez le constater, cela ressemble beaucoup à une déclaration de méthode. Cependant, nous n'avons pas encore décrit le comportement de notre méthode. Comme indiqué précédemment, le schéma est uniquement là pour définir quelles seront les données et les opérations, et non leur mode de fonctionnement. La mise en œuvre de la véritable logique métier interviendra plus tard lorsque nous créerons nos premiers résolveurs.
Une fois que vous avez terminé avec votre schéma, il est possible de l'exporter en tant que`schema.graphql`fichier. Dans leEditeur de schéma, vous pouvez choisirSchéma d'exportationpour télécharger le fichier dans un format compatible.
À titre de note supplémentaire,AWS AppSyncajoute automatiquement une racine de schéma lors des exportations. Techniquement, vous n'avez donc pas besoin de l'ajouter directement au schéma. Notre service traitera automatiquement les schémas dupliqués. Nous l'ajoutons ici en tant que meilleure pratique.

CLI

> **Note**
>
> Nous vous recommandons de lire d'abord la version console si ce n'est pas déjà fait.

1. Mettez à jour votre schéma racine en exécutant le`update-type`commande.

   Vous devez entrer quelques paramètres pour cette commande particulière :

   1. Le`api-id`de votre API.
   2. Le`type-name`de votre type. Dans l'exemple de console, c'était`schema`.
   3. Le`definition`, ou le contenu de votre type. Dans l'exemple de console, c'était :

      ```
      schema {
        query: Query
        mutation: Mutation
      }
      ```

   4. Le`format`de votre contribution. Dans cet exemple, nous utilisons`SDL`.

   Voici un exemple de commande :

   ```
   aws appsync update-type --api-id abcdefghijklmnopqrstuvwxyz --type-name schema
    --definition "schema {query: Query mutation: Mutation}" --format SDL
   ```

   Une sortie sera renvoyée dans la CLI. Voici un exemple:

   ```
   {
       "type": {
           "definition": "schema {query: Query mutation: Mutation}",
           "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
   abcdefghijklmnopqrstuvwxyz/types/schema",
           "format": "SDL"
       }
   }
   ```

2. Créez un`Mutation`tapez en exécutant`create-type`commande.

Vous devez entrer quelques paramètres pour cette commande particulière :

1. Le`api-id`de votre API.
2. Le`definition`, ou le contenu de votre type. Dans l'exemple de console, c'était

```
type Mutation {
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
}
```

3. Le`format`de votre contribution. Dans cet exemple, nous utilisonsSDL.

Voici un exemple de commande :

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type
 Mutation {addObj(id: ID! title: String date: AWSDateTime): Obj_Type_1}" --
format SDL
```

Une sortie sera renvoyée dans la CLI. Voici un exemple:

```
{
    "type": {
        "definition": "type Mutation {addObj(id: ID! title: String date:
 AWSDateTime): Obj_Type_1}",
        "name": "Mutation",
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Mutation",
        "format": "SDL"
    }
}
```

CDK

> **Tip**
>
> Avant d'utiliser le CDK, nous vous recommandons de consulter les CDKdocumentation officielleainsi queAWS AppSyncc'estRéférence CDK.
> Les étapes répertoriées ci-dessous ne montreront qu'un exemple général de l'extrait de code utilisé pour ajouter une ressource particulière. C'estpascensé être une solution

fonctionnelle dans votre code de production. Nous partons également du principe que vous disposez déjà d'une application fonctionnelle.

Vous devez ajouter votre requête et la racine du schéma dans `.graphql`fichier. Notre exemple ressemble à l'exemple ci-dessous, mais vous devez le remplacer par votre code de schéma actuel :

```
schema {
  query: Query
  mutation: Mutation
}

type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}

type Query {
  getObj: [Obj_Type_1]
}

type Mutation {
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
}
```

Note

La mise à jour de la racine du schéma est facultative. Nous l'avons ajoutée à cet exemple en tant que meilleure pratique.
Pour utiliser les modifications apportées à votre API GraphQL, vous devez redéployer l'application.

Considérations facultatives - Utilisation des énumérations comme statuts

À ce stade, vous savez comment créer un schéma de base. Cependant, vous pouvez ajouter de nombreux éléments pour améliorer les fonctionnalités du schéma. Une chose courante dans les applications est l'utilisation d'énumérations comme statuts. Vous pouvez utiliser une énumération

pour forcer le choix d'une valeur spécifique d'un ensemble de valeurs lors de l'appel. C'est bon pour des choses dont vous savez qu'elles ne changeront pas radicalement sur de longues périodes. Hypothétiquement parlant, nous pourrions ajouter une énumération qui renvoie le code d'état ou une chaîne dans la réponse.

Par exemple, supposons que nous créons une application de réseau social qui stocke les données de publication d'un utilisateur dans le backend. Notre schéma contient un`Post`type qui représente les données d'une publication individuelle :

```
type Post {
  id: ID!
  title: String
  date: AWSDateTime
  poststatus: PostStatus
}
```

Notre`Post`contiendra un`id`, poste`title`,`date`de publication, et une énumération appelée`PostStatus`qui représente l'état de la publication telle qu'elle est traitée par l'application. Pour nos opérations, nous aurons une requête qui renverra toutes les données de publication :

```
type Query {
  getPosts: [Post]
}
```

Nous aurons également une mutation qui ajoutera des publications à la source de données :

```
type Mutation {
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post
}
```

En regardant notre schéma, le`PostStatus`enum peut avoir plusieurs statuts. Nous voudrions peut-être que les trois états de base soient appelés`success`(message traité avec succès),`pending`(poste en cours de traitement), et`error`(le message n'a pas pu être traité). Pour ajouter l'énumération, nous pourrions faire ceci :

```
enum PostStatus {
  success
  pending
  error
```

```
}
```

Le schéma complet pourrait ressembler à ceci :

```
schema {
  query: Query
  mutation: Mutation
}

type Post {
  id: ID!
  title: String
  date: AWSDateTime
  poststatus: PostStatus
}

type Mutation {
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post
}

type Query {
  getPosts: [Post]
}

enum PostStatus {
  success
  pending
  error
}
```

Si un utilisateur ajoute un`Post`dans l'application, le`addPost`une opération sera appelée pour traiter ces données. En tant que résolveur attaché à`addPost`traite les données, il mettra continuellement à jour le`poststatus`avec le statut de l'opération. Lorsqu'on le lui demande, le`Post`contiendra le statut final des données. N'oubliez pas que nous ne décrivons que la manière dont nous voulons que les données fonctionnent dans le schéma. Nous avons beaucoup d'hypothèses quant à la mise en œuvre de nos résolveurs, qui mettront en œuvre la logique métier réelle pour traiter les données afin de répondre à la demande.

Considérations facultatives - Abonnements

Abonnements en AWS AppSync sont invoqués en réponse à une mutation. Vous configurez cela avec un type `Subscription` et une directive `@aws_subscribe()` du schéma pour signifier quelles

mutations appellent un ou plusieurs abonnements. Pour plus d'informations sur la configuration des abonnements, voirDonnées en temps réel.

Considérations facultatives - Relations et pagination

Supposons que vous en ayez un million`Posts`stockées dans une table DynamoDB, et vous vouliez renvoyer certaines de ces données. Cependant, l'exemple de requête donné ci-dessus ne renvoie que tous les messages. Vous ne voudriez pas les récupérer tous à chaque fois que vous faites une demande. Au lieu de cela, vous voudriezpaginerà travers eux. Apportez les modifications suivantes à votre schéma :

- Dans le`getPosts`champ, ajoutez deux arguments d'entrée :`nextToken`(itérateur) et`limit`(limite d'itération).
- Ajouter un nouveau`PostIterator`type contenant`Posts`(récupère la liste des`Post`objets) et`nextToken`champs (itérateur).
- Changez`getPosts`pour qu'il revienne`PostIterator`et non une liste de`Post`objets.

```
schema {
  query: Query
  mutation: Mutation
}

type Post {
  id: ID!
  title: String
  date: AWSDateTime
  poststatus: PostStatus
}

type Mutation {
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post
}

type Query {
  getPosts(limit: Int, nextToken: String): PostIterator
}

enum PostStatus {
  success
  pending
```

```
  error
}

type PostIterator {
  posts: [Post]
  nextToken: String
}
```

Le`PostIterator`type vous permet de renvoyer une partie de la liste des`Post`des objets et un`nextToken`pour obtenir la portion suivante. À l'intérieur`PostIterator`, il existe une liste de`Post`objets (`[Post]`) qui est renvoyé avec un jeton de pagination (`nextToken`). DansAWS AppSync, celui-ci serait connecté à Amazon DynamoDB via un résolveur et généré automatiquement sous forme de jeton chiffré. Cela convertir la valeur de l'argument `limit` en paramètre `maxResults` et l'argument `nextToken` en paramètre `exclusiveStartKey`. Pour obtenir des exemples et des exemples de modèles intégrés dans leAWS AppSyncconsole, voirRéférence du résolveur (JavaScript).

Étape 2 : Joindre une source de données

Les sources de données sont des ressources de votreAWScompte avec lequel les API GraphQL peuvent interagir.AWS AppSyncprend en charge une multitude de sources de données telles queAWS Lambda, Amazon DynamoDB, bases de données relationnelles (Amazon Aurora Serverless), AmazonOpenSearchService et points de terminaison HTTP. UnAWS AppSyncL'API peut être configurée pour interagir avec plusieurs sources de données, ce qui vous permet d'agréger les données en un seul endroit.AWS AppSyncpeut utiliser les produits existantsAWSdes ressources de votre compte ou provisionnez des tables DynamoDB en votre nom à partir d'une définition de schéma.

La section suivante explique comment associer une source de données à votre API GraphQL.

Types de sources de données

Maintenant que vous avez créé un schéma dans leAWS AppSyncconsole, vous pouvez y associer une source de données. Lorsque vous créez une API pour la première fois, il est possible de configurer une table Amazon DynamoDB lors de la création du schéma prédéfini. Cependant, nous n'aborderons pas cette option dans cette section. Vous pouvez en voir un exemple dans leLancement d'un schémasection.

Nous examinerons plutôt toutes les sources de donnéesAWS AppSyncsoutient. De nombreux facteurs entrent en ligne de compte pour choisir la bonne solution pour votre application. Les sections

ci-dessous fournissent un contexte supplémentaire pour chaque source de données. Pour des informations générales sur les sources de données, voirSources de données.

Amazon DynamoDB

Amazon DynamoDB est l'un desAWS'principales solutions de stockage pour les applications évolutives. Le composant principal de DynamoDB est letable, qui est simplement un recueil de données. Vous créerez généralement des tables basées sur des entités telles que`Book`ou`Author`. Les informations d'entrée dans le tableau sont stockées sous la formearticles, qui sont des groupes de champs uniques à chaque entrée. Un élément complet représente une ligne/un enregistrement dans la base de données. Par exemple, un article pour`Book`l'entrée peut inclure`title`et`author`ainsi que leurs valeurs. Les champs individuels tels que le`title`et`author`sont appelésattributs, qui sont similaires aux valeurs de colonne dans les bases de données relationnelles.

Comme vous pouvez le deviner, les tables seront utilisées pour stocker les données de votre application.AWS AppSyncvous permet de connecter vos tables DynamoDB à votre API GraphQL pour manipuler les données. Prends çacas d'utilisationà partir duBlog web et mobile frontal. Cette application permet aux utilisateurs de s'inscrire à une application de réseau social. Les utilisateurs peuvent rejoindre des groupes et télécharger des publications qui sont diffusées aux autres utilisateurs abonnés au groupe. Leur application stocke les informations relatives aux utilisateurs, aux publications et aux groupes d'utilisateurs dans DynamoDB. L'API GraphQL (gérée parAWS AppSync) s'interface avec la table DynamoDB. Lorsqu'un utilisateur apporte une modification au système qui sera répercutée sur le front-end, l'API GraphQL récupère ces modifications et les diffuse aux autres utilisateurs en temps réel.

AWS Lambda

Lambda est un service piloté par les événements qui crée automatiquement les ressources nécessaires pour exécuter le code en réponse à un événement. Lambda utilisefonctions, qui sont des instructions de groupe contenant le code, les dépendances et les configurations nécessaires à l'exécution d'une ressource. Les fonctions s'exécutent automatiquement lorsqu'elles détectent undéclencheur, un groupe d'activités qui invoque votre fonction. Un déclencheur peut être quelque chose comme une application effectuant un appel d'API, unAWSservice sur votre compte permettant de créer une ressource, etc. Lorsqu'elles sont déclenchées, les fonctions seront traitéesévénements, qui sont des documents JSON contenant les données à modifier.

Lambda est idéal pour exécuter du code sans avoir à fournir les ressources nécessaires à son exécution. Prends çacas d'utilisationà partir duBlog web et mobile frontal. Ce cas d'utilisation est un peu similaire à celui présenté dans la section DynamoDB. Dans cette application, l'API

GraphQL est chargée de définir les opérations pour des choses telles que l'ajout de publications (mutations) et la récupération de ces données (requêtes). Pour mettre en œuvre les fonctionnalités de leurs opérations (par exemple,`getPost ( id: String ! ) : Post`,`getPostsByAuthor ( author: String ! ) : [ Post ]`), ils utilisent les fonctions Lambda pour traiter les demandes entrantes. En dessousOption 2 :AWS AppSyncavec résolveur Lambda, ils utilisentAWS AppSyncservice pour gérer leur schéma et lier une source de données Lambda à l'une des opérations. Lorsque l'opération est appelée, Lambda s'interface avec le proxy Amazon RDS pour exécuter la logique métier sur la base de données.

Amazon RDS

Amazon RDS vous permet de créer et de configurer rapidement des bases de données relationnelles. Dans Amazon RDS, vous allez créer un génériqueinstance de base de donnéesqui servira d'environnement de base de données isolé dans le cloud. Dans ce cas, vous allez utiliser unmoteur de base de données, qui est le véritable logiciel RDBMS (PostgreSQL, MySQL, etc.). Le service décharge une grande partie du travail de backend en fournissant une évolutivité à l'aide deAWS» infrastructure, services de sécurité tels que les correctifs et le chiffrement, et réduction des coûts administratifs liés aux déploiements.

Prends le mêmecas d'utilisationdepuis la section Lambda. En dessousOption 3 :AWS AppSyncavec le résolveur Amazon RDS, une autre option présentée consiste à lier l'API GraphQL dansAWS AppSyncdirectement à Amazon RDS. À l'aide d'unAPI de données, ils associent la base de données à l'API GraphQL. Un résolveur est attaché à un champ (généralement une requête, une mutation ou un abonnement) et implémente les instructions SQL nécessaires pour accéder à la base de données. Lorsqu'une demande appelant le champ est faite par le client, le résolveur exécute les instructions et renvoie la réponse.

Amazon EventBridge

DansEventBridge, vous allez créerbus événementiels, qui sont des pipelines qui reçoivent des événements provenant de services ou d'applications que vous associez (lesource de l'événement) et les traiter selon un ensemble de règles. Unévénementest un changement d'état dans un environnement d'exécution, tandis qu'unrègleest un ensemble de filtres pour les événements. Une règle suit unmodèle d'événement, ou les métadonnées du changement d'état d'un événement (identifiant, région, numéro de compte, ARN, etc.). Lorsqu'un événement correspond au modèle d'événement,EventBridgeenverra l'événement à travers le pipeline jusqu'au service de destination (cible) et déclenchez l'action spécifiée dans la règle.

EventBridgeest utile pour acheminer les opérations de changement d'état vers un autre service. Prends çacas d'utilisationà partir duBlog web et mobile frontal. L'exemple illustre une solution de commerce électronique dans laquelle plusieurs équipes gèrent différents services. L'un de ces services fournit des mises à jour des commandes au client à chaque étape de la livraison (commande passée, en cours, expédiée, livrée, etc.) sur le front-end. Cependant, l'équipe frontale qui gère ce service n'a pas un accès direct aux données du système de commande, car celles-ci sont gérées par une équipe principale distincte. Le système de commande de l'équipe principale est également décrit comme une boîte noire. Il est donc difficile de glaner des informations sur la manière dont l'équipe structure ses données. Cependant, l'équipe principale a mis en place un système qui publiait les données des commandes via un bus d'événements géré parEventBridge. Pour accéder aux données provenant du bus d'événements et les acheminer vers le front-end, l'équipe du front-end a créé une nouvelle cible pointant vers son API GraphQL située dansAWS AppSync. Ils ont également créé une règle pour envoyer uniquement les données relatives à la mise à jour de la commande. Lorsqu'une mise à jour est effectuée, les données du bus d'événements sont envoyées à l'API GraphQL. Le schéma de l'API traite les données, puis les transmet au front-end.

Aucune source de données

Si vous ne prévoyez pas d'utiliser une source de données, vous pouvez la configurer sur`none`. UN`none`une source de données, bien qu'elle soit toujours explicitement classée comme source de données, n'est pas un support de stockage. Généralement, un résolveur invoque une ou plusieurs sources de données à un moment donné pour traiter la demande. Il existe toutefois des situations dans lesquelles il n'est pas nécessaire de manipuler une source de données. Configuration de la source de données sur`none`exécutera la demande, sautera l'étape d'invocation des données, puis exécutera la réponse.

Prends le mêmecas d'utilisationà partir duEventBridgesection. Dans le schéma, la mutation traite la mise à jour du statut, puis l'envoie aux abonnés. Pour rappeler le fonctionnement des résolveurs, il y a généralement au moins un appel de source de données. Cependant, les données de ce scénario ont déjà été envoyées automatiquement par le bus d'événements. Cela signifie que la mutation n'est pas nécessaire pour effectuer un appel de source de données ; le statut de la commande peut simplement être géré localement. La mutation est définie sur`none`, qui agit comme une valeur directe sans appel de source de données. Le schéma est ensuite renseigné avec les données, qui sont envoyées aux abonnés.

OpenSearch

AmazonOpenSearchLe service est une suite d'outils permettant de mettre en œuvre la recherche en texte intégral, la visualisation des données et la journalisation. Vous pouvez utiliser ce service pour interroger les données structurées que vous avez téléchargées.

Dans ce service, vous allez créer des instances deOpenSearch. Ils sont appelésnœuds. Dans un nœud, vous allez ajouter au moins unindex. D'un point de vue conceptuel, les indices sont un peu comme des tables dans des bases de données relationnelles. (Cependant,OpenSearchn'est pas conforme à l'ACID, il ne doit donc pas être utilisé de cette façon). Vous allez remplir votre index avec les données que vous téléchargerez dans leOpenSearchservice. Lorsque vos données sont téléchargées, elles sont indexées dans une ou plusieurs partitions présentes dans l'index. UNtessonest comme une partition de votre index qui contient certaines de vos données et qui peut être interrogée séparément des autres partitions. Une fois chargées, vos données seront structurées sous forme de fichiers JSON appelésdocuments. Vous pouvez ensuite interroger le nœud pour obtenir des données dans le document.

Points de terminaison HTTP

Vous pouvez utiliser des points de terminaison HTTP comme sources de données.AWS AppSyncpeut envoyer des demandes aux points de terminaison avec les informations pertinentes telles que les paramètres et la charge utile. La réponse HTTP sera exposée au résolveur, qui renverra la réponse finale une fois ses opérations terminées.

Ajouter une source de données

Si vous avez créé une source de données, vous pouvez la lier auAWS AppSyncservice et, plus précisément, l'API.

Console

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
    a. Choisissez votre API dansTableau de bord.
    b. Dans leBarre latérale, choisissezSources de données.
2. Choisissez Create data source.
    a. Donnez un nom à votre source de données. Vous pouvez également lui donner une description, mais c'est facultatif.
    b. Choisissez votreType de source de données.

c. Pour DynamoDB, vous devez choisir votre région, puis le tableau dans la région. Vous pouvez dicter des règles d'interaction avec votre table en choisissant de créer un nouveau rôle de table générique ou en important un rôle existant pour la table. Vous pouvez activergestion des versions, qui permet de créer automatiquement des versions des données pour chaque demande lorsque plusieurs clients tentent de mettre à jour les données en même temps. Le versionnement est utilisé pour conserver et gérer plusieurs variantes de données à des fins de détection et de résolution de conflits. Vous pouvez également activer la génération automatique de schémas, qui prend votre source de données et génère une partie du CRUD,`List`, et`Query`opérations nécessaires pour y accéder dans votre schéma.

   PourOpenSearch, vous devrez choisir votre région, puis le domaine (cluster) de la région. Vous pouvez dicter des règles d'interaction avec votre domaine en choisissant de créer un nouveau rôle de table générique ou en important un rôle existant pour la table.

   Pour Lambda, vous devrez choisir votre région, puis l'ARN de la fonction Lambda dans la région. Vous pouvez dicter des règles d'interaction avec votre fonction Lambda en choisissant de créer un nouveau rôle de table générique ou en important un rôle existant pour la table.

   Pour le protocole HTTP, vous devez saisir votre point de terminaison HTTP.

   PourEventBridge, vous devrez choisir votre région, puis le bus événementiel de la région. Vous pouvez dicter des règles d'interaction avec votre bus d'événements en choisissant de créer un nouveau rôle de table générique ou en important un rôle existant pour la table.

   Pour RDS, vous devrez choisir votre région, puis le magasin secret (nom d'utilisateur et mot de passe), le nom de la base de données et le schéma.

   Dans aucun cas, vous ajouterez une source de données sans source de données réelle. Cela permet de gérer les résolveurs localement plutôt que par le biais d'une source de données réelle.

   > **Note**
   >
   > Si vous importez des rôles existants, ils ont besoin d'une politique de confiance. Pour plus d'informations, consultez lePolitique de confiance IAM.

3. Sélectionnez Create (Créer).

> **Note**
>
> Si vous créez une source de données DynamoDB, vous pouvez également accéder auSchémapage dans la console, choisissezCréation de ressourcesen haut de la page, puis remplissez un modèle prédéfini à convertir en tableau. Dans cette option, vous allez remplir ou importer le type de base, configurer les données de base de la table, y compris la clé de partition, et passer en revue les modifications du schéma.

CLI

- Créez votre source de données en exécutant le`create-data-source`commande.

  Vous devez entrer quelques paramètres pour cette commande particulière :

  1. Le`api-id`de votre API.
  2. Le`name`de votre table.
  3. Le`type`de la source de données. Selon le type de source de données que vous choisissez, vous devrez peut-être saisir un`service-role-arn`et un`-config`étiquette.

  Voici un exemple de commande :

```
aws appsync create-data-source --api-id abcdefghijklmnopqrstuvwxyz
--name data_source_name --type data_source_type --service-role-arn
arn:aws:iam::107289374856:role/role_name --[data_source_type]-config {params}
```

CDK

> **Tip**
>
> Avant d'utiliser le CDK, nous vous recommandons de consulter les CDKdocumentation officielleainsi queAWS AppSyncc'estRéférence CDK.
> Les étapes répertoriées ci-dessous ne montreront qu'un exemple général de l'extrait de code utilisé pour ajouter une ressource particulière. C'estpascensé être une solution

fonctionnelle dans votre code de production. Nous partons également du principe que vous disposez déjà d'une application fonctionnelle.

Pour ajouter votre source de données spécifique, vous devez ajouter la construction à votre fichier de pile. Vous trouverez une liste des types de sources de données ici :

- DynamoDbDataSource
- EventBridgeDataSource
- HttpDataSource
- LambdaDataSource
- NoneDataSource
- OpenSearchDataSource
- RdsDataSource

1. En général, vous devrez peut-être ajouter la directive d'importation au service que vous utilisez. Par exemple, il peut suivre les formes suivantes :

   ```
   import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'
   import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
   ```

   Par exemple, voici comment vous pouvez importer leAWS AppSyncet services DynamoDB :

   ```
   import * as appsync from 'aws-cdk-lib/aws-appsync';
   import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
   ```

2. Certains services tels que RDS nécessitent une configuration supplémentaire dans le fichier de pile avant de créer la source de données (par exemple, création d'un VPC, rôles et informations d'identification d'accès). Consultez les exemples dans les pages CDK correspondantes pour plus d'informations.
3. Pour la plupart des sources de données, en particulierAWSservices, vous allez créer une nouvelle instance de la source de données dans votre fichier de pile. Cela ressemblera généralement à ce qui suit :

   ```
   const add_data_source_func = new service_scope.resource_name(scope: Construct,
    id: string, props: data_source_props);
   ```

Par exemple, voici un exemple de table Amazon DynamoDB :

```
const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {
  partitionKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
  sortKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
  tableClass: dynamodb.TableClass.STANDARD,
});
```

> **Note**
>
> La plupart des sources de données auront au moins un accessoire obligatoire (sera indiquésansun?symbole). Consultez la documentation du CDK pour savoir quels accessoires sont nécessaires.

4. Ensuite, vous devez lier la source de données à l'API GraphQL. La méthode recommandée consiste à l'ajouter lorsque vous créez une fonction pour votre résolveur de pipeline. Par exemple, l'extrait ci-dessous est une fonction qui analyse tous les éléments d'une table DynamoDB :

```
const add_func = new appsync.AppsyncFunction(this, 'func_ID', {
  name: 'func_name_in_console',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('data_source_name_in_console',
 add_ddb_table),
  code: appsync.Code.fromInline(`
      export function request(ctx) {
        return { operation: 'Scan' };
      }

      export function response(ctx) {
        return ctx.result.items;
      }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
```

```
});
```

Dans ledataSourceaccessoires, vous pouvez appeler l'API GraphQL (add_api) et utilisez l'une de ses méthodes intégrées (addDynamoDbDataSource) pour établir l'association entre la table et l'API GraphQL. Les arguments sont le nom de ce lien qui existera dans leAWS AppSyncconsole (data_source_name_in_consoledans cet exemple) et la méthode table (add_ddb_table). Plus d'informations sur ce sujet seront révélées dans la section suivante lorsque vous commencerez à créer des résolveurs.

Il existe d'autres méthodes pour lier une source de données. Techniquement, vous pourriez ajouterapià la liste des accessoires dans la fonction de table. Par exemple, voici l'extrait de l'étape 3, mais avec unapiaccessoires contenant une API GraphQL :

```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {
  ...
});

const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {

 ...

  api: add_api
});
```

Vous pouvez également appeler leGraphqlApiconstruire séparément :

```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {
  ...
});

const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {
  ...
});

const link_data_source =
 add_api.addDynamoDbDataSource('data_source_name_in_console', add_ddb_table);
```

Nous recommandons de créer l'association uniquement dans les accessoires de la fonction. Sinon, vous devrez soit lier manuellement votre fonction de résolution à la source de données dans leAWS AppSyncconsole (si vous souhaitez continuer à utiliser la valeur de

la console)`data_source_name_in_console`) ou créez une association distincte dans la fonction sous un autre nom tel que`data_source_name_in_console_2`. Cela est dû aux limites de la manière dont les accessoires traitent les informations.

> **Note**
>
> Vous devrez redéployer l'application pour voir vos modifications.

Politique de confiance IAM

Si vous utilisez un rôle IAM existant pour votre source de données, vous devez accorder à ce rôle les autorisations appropriées pour effectuer des opérations sur votreAWSressource, telle que`PutItem`sur une table Amazon DynamoDB. Vous devez également modifier la politique de confiance relative à ce rôle pour permettreAWS AppSyncpour l'utiliser pour accéder aux ressources, comme indiqué dans l'exemple de politique suivant :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Effect": "Allow",
        "Principal": {
            "Service": "appsync.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
        }
    ]
}
```

Vous pouvez également ajouter des conditions à votre politique de confiance afin de limiter l'accès à la source de données comme vous le souhaitez. À l'heure actuelle,`SourceArn`et`SourceAccount`les clés peuvent être utilisées dans ces conditions. Par exemple, la politique suivante limite l'accès à votre source de données au compte`123456789012`:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
```

```
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "123456789012"
        }
      }
    }
  ]
}
```

Vous pouvez également limiter l'accès à une source de données à une API spécifique, telle que`abcdefghijklmnopq`, en appliquant la politique suivante :

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnEquals": {
          "aws:SourceArn": "arn:aws:appsync:us-west-2:123456789012:apis/
abcdefghijklmnopq"
        }
      }
    }
  ]
}
```

Vous pouvez limiter l'accès à tousAWS AppSyncLes API d'une région spécifique, telles que`us-east-1`, en appliquant la politique suivante :

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
```

```
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnEquals": {
          "aws:SourceArn": "arn:aws:appsync:us-east-1:123456789012:apis/*"
        }
      }
    }
  ]
}
```

Dans la section suivante (Configuration des résolveurs), nous allons ajouter notre logique métier de résolveur et l'associer aux champs de notre schéma pour traiter les données de notre source de données.

Pour plus d'informations sur la configuration de la politique de rôle, voirModifier un rôledans leGuide de l'utilisateur IAM.

Pour plus d'informations concernant l'accès entre comptes deAWS Lambdarésolveurs pourAWS AppSync, voirCréation de comptes croisésAWS Lambdarésolveurs pourAWS AppSync.

Étape 3 : Configuration des résolveurs

Dans les sections précédentes, vous avez appris à créer votre schéma GraphQL et votre source de données, puis à les lier ensemble dans le AWS AppSync service. Dans votre schéma, vous avez peut-être défini un ou plusieurs champs (opérations) dans votre requête et votre mutation. Bien que le schéma décrive les types de données que les opérations demanderaient à la source de données, il n'a jamais mis en œuvre le comportement de ces opérations en fonction des données.

Le comportement d'une opération est toujours implémenté dans le résolveur, qui sera lié au champ effectuant l'opération. Pour plus d'informations sur le fonctionnement général des résolveurs, consultez la page Résolveurs.

Dans AWS AppSync, votre résolveur est lié à un environnement d'exécution, qui est l'environnement dans lequel votre résolveur s'exécute. Les environnements d'exécution dictent la langue dans laquelle votre résolveur sera écrit. Deux environnements d'exécution sont actuellement pris en charge : APPSYNC_JS (JavaScript) et Apache Velocity Template Language (VTL).

Lors de la mise en œuvre de résolveurs, ils suivent une structure générale :

- Avant l'étape : lorsqu'une demande est faite par le client, les données de la demande sont transmises aux résolveurs des champs de schéma utilisés (généralement vos requêtes, mutations, abonnements). Le résolveur commencera à traiter les données de la demande à l'aide d'un gestionnaire avant étape, ce qui permet d'effectuer certaines opérations de prétraitement avant que les données ne passent par le résolveur.
- Fonction (s) : Une fois l'étape précédente exécutée, la demande est transmise à la liste des fonctions. La première fonction de la liste s'exécutera sur la source de données. Une fonction est un sous-ensemble du code de votre résolveur contenant son propre gestionnaire de requêtes et de réponses. Un gestionnaire de demandes prendra les données de la demande et effectuera des opérations sur la source de données. Le gestionnaire de réponses traitera la réponse de la source de données avant de la renvoyer à la liste. S'il existe plusieurs fonctions, les données de la demande seront envoyées à la fonction suivante de la liste à exécuter. Les fonctions de la liste seront exécutées en série dans l'ordre défini par le développeur. Une fois que toutes les fonctions ont été exécutées, le résultat final est transmis à l'étape suivante.
- Étape suivante : L'étape suivante est une fonction de gestion qui vous permet d'effectuer certaines opérations finales sur la réponse de la fonction finale avant de la transmettre à la réponse GraphQL.

Ce flux est un exemple de résolveur de pipeline. Les résolveurs de pipeline sont pris en charge dans les deux environnements d'exécution. Cependant, il s'agit d'une explication simplifiée de ce que les résolveurs de pipeline peuvent faire. De plus, nous ne décrivons qu'une seule configuration de résolveur possible. Pour plus d'informations sur les configurations de résolveurs prises en charge, consultez l'aperçu des JavaScript résolveurs pour APPSYNC_JS ou l'aperçu du modèle de mappage des résolveurs pour VTL.

Comme vous pouvez le constater, les résolveurs sont modulaires. Pour que les composants du résolveur fonctionnent correctement, ils doivent être capables de vérifier l'état de l'exécution à partir d'autres composants. Dans la section Résolveurs, vous savez que chaque composant du résolveur peut recevoir des informations vitales sur l'état de l'exécution sous forme d'un ensemble d'arguments (`args`,`context`, etc.). En AWS AppSync, cela est géré strictement par le`context`. Il s'agit d'un conteneur contenant les informations relatives au champ en cours de résolution. Cela peut inclure tout, des arguments transmis aux résultats, aux données d'autorisation, aux données d'en-tête, etc. Pour plus d'informations sur le contexte, consultez la référence de l'objet de contexte Resolver pour APPSYNC_JS ou la référence de contexte du modèle de mappage Resolver pour VTL.

Le contexte n'est pas le seul outil que vous pouvez utiliser pour implémenter votre résolveur. AWS AppSync prend en charge un large éventail d'utilitaires pour la génération de valeur, la gestion

des erreurs, l'analyse syntaxique, la conversion, etc. Vous pouvez voir une liste d'utilitaires ici pour APPSYNC_JS ou ici pour VTL.

Dans les sections suivantes, vous allez apprendre à configurer les résolveurs dans votre API GraphQL.

Rubriques

- Configuration des résolveurs (JavaScript)
- Configuration des résolveurs (VTL)

Configuration des résolveurs (JavaScript)

Les résolveurs GraphQL connectent les champs d'un schéma de type à une source de données. Les résolveurs sont le mécanisme par lequel les demandes sont satisfaites.

Résolveurs dansAWS AppSyncutiliserJavaScriptpour convertir une expression GraphQL dans un format utilisable par la source de données. Les modèles de mappage peuvent également être écrits dansLangage de modèle Apache Velocity (VTL)pour convertir une expression GraphQL dans un format utilisable par la source de données.

Cette section décrit comment configurer les résolveurs à l'aide deJavaScript. LeTutoriels Resolver (JavaScript)la section fournit des didacticiels approfondis sur la façon d'implémenter des résolveurs en utilisantJavaScript. LeRéférence du résolveur (JavaScript)cette section fournit une explication des opérations utilitaires qui peuvent être utilisées avecJavaScriptrésolveurs.

Nous vous recommandons de suivre ce guide avant d'essayer d'utiliser l'un des didacticiels mentionnés ci-dessus.

Dans cette section, nous expliquerons comment créer et configurer des résolveurs pour les requêtes et les mutations.

> **Note**
>
> Ce guide part du principe que vous avez créé votre schéma et que vous avez au moins une requête ou une mutation. Si vous recherchez des abonnements (données en temps réel), consultezceguide.

Dans cette section, nous allons fournir quelques étapes générales pour configurer les résolveurs, ainsi qu'un exemple utilisant le schéma ci-dessous :

```
// schema.graphql file

input CreatePostInput {
  title: String
  date: AWSDateTime
}

type Post {
  id: ID!
  title: String
  date: AWSDateTime
}

type Mutation {
  createPost(input: CreatePostInput!): Post
}

type Query {
  getPost: [Post]
}
```

Création de résolveurs de requêtes de base

Cette section explique comment créer un résolveur de requêtes de base.

Console

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
   a. Dans leTableau de bord des API, choisissez votre API GraphQL.
   b. Dans leBarre latérale, choisissezSchéma.
2. Entrez les détails de votre schéma et de votre source de données. Voir leConception de votre schémaetJoindre une source de donnéessections pour plus d'informations.
3. À côté duSchémaéditeur, il y a une fenêtre appeléeRésolveurs. Cette zone contient une liste des types et des champs tels que définis dans votreSchémafenêtre. Vous pouvez associer des résolveurs aux champs. Vous allez probablement associer des résolveurs à vos opérations sur le terrain. Dans cette section, nous allons examiner les configurations de

requêtes simples. En vertu duRequêtetapez, choisissezJoindreà côté du champ de votre requête.

4. Sur leAttacher un résolveurpage, sousType de résolveur, vous pouvez choisir entre des résolveurs de type pipeline ou unitaire. Pour plus d'informations sur ces types, voirRésolveurs. Ce guide utiliserapipeline resolvers.

   > **Tip**
   >
   > Lorsque vous créez des résolveurs de pipeline, vos sources de données seront associées aux fonctions de pipeline. Les fonctions sont créées une fois que vous avez créé le résolveur de pipeline lui-même, c'est pourquoi il n'est pas possible de le définir dans cette page. Si vous utilisez un résolveur d'unités, la source de données est directement liée au résolveur. Vous devez donc la définir dans cette page.

   PourTemps d'exécution du résolveur, choisissezAPPSYNC_JSpour activer leJavaScripttemps d'exécution.

5. Vous pouvez activermise en cachepour cette API. Nous vous recommandons de désactiver cette fonctionnalité pour le moment. Sélectionnez Create (Créer).

6. Sur leModifier le résolveurpage, il existe un éditeur de code appeléCode du résolveurqui vous permet d'implémenter la logique du gestionnaire de résolution et de la réponse (étapes avant et après). Pour plus d'informations, consultez leJavaScriptvue d'ensemble des résolveurs.

   > **Note**
   >
   > Dans notre exemple, nous allons simplement laisser la demande vide et la réponse définie pour renvoyer le dernier résultat de la source de données ducontexte:

   ```
   import {util} from '@aws-appsync/utils';

   export function request(ctx) {
       return {};
   }

   export function response(ctx) {
       return ctx.prev.result;
   }
   ```

En dessous de cette section, il y a un tableau appeléFonctions. Les fonctions vous permettent d'implémenter du code qui peut être réutilisé dans plusieurs résolveurs. Au lieu de constamment réécrire ou copier le code, vous pouvez stocker le code source sous forme de fonction à ajouter à un résolveur chaque fois que vous en avez besoin.

Les fonctions constituent l'essentiel de la liste des opérations d'un pipeline. Lorsque vous utilisez plusieurs fonctions dans un résolveur, vous définissez l'ordre des fonctions, et elles seront exécutées dans cet ordre de manière séquentielle. Ils sont exécutés après l'exécution de la fonction de demande et avant le début de la fonction de réponse.

Pour ajouter une nouvelle fonction, sousFonctions, choisissezAjouter une fonction, puisCréer une nouvelle fonction. Vous pouvez également voir unCréer une fonctionbouton pour choisir à la place.

a. Choisissez une source de données. Il s'agira de la source de données sur laquelle le résolveur agira.

> **Note**
>
> Dans notre exemple, nous attachons un résolveur pour`getPost`, qui récupère un`Post`objet par`id`. Supposons que nous ayons déjà configuré une table DynamoDB pour ce schéma. Sa clé de partition est définie sur`id`et est vide.

b. Entrez un`Function name`.

c. En dessousCode de fonction, vous devez implémenter le comportement de la fonction. Cela peut prêter à confusion, mais chaque fonction aura son propre gestionnaire de requêtes et de réponses local. La demande est exécutée, puis la source de données est invoquée pour traiter la demande, puis la réponse de la source de données est traitée par le gestionnaire de réponses. Le résultat est enregistré danscontexteobjet. Ensuite, la fonction suivante de la liste s'exécutera ou sera transmise au gestionnaire de réponses après étape s'il s'agit de la dernière.

> **Note**
>
> Dans notre exemple, nous associons un résolveur à`getPost`, qui obtient une liste de`Post`objets de la source de données. Notre fonction de requête demandera les données de notre table, la table transmettra sa réponse au

contexte (ctx), puis la réponse renverra le résultat dans le contexte.AWS AppSyncSa force réside dans son interconnexion avec les autresAWSservices. Parce que nous utilisons DynamoDB, nous avonssuite d'opérationspour simplifier ce genre de choses. Nous avons également quelques exemples standard pour d'autres types de sources de données.
Notre code ressemblera à ceci :

```
import { util } from '@aws-appsync/utils';

/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

Au cours de cette étape, nous avons ajouté deux fonctions :

- `request`: Le gestionnaire de demandes effectue l'opération de récupération par rapport à la source de données. L'argument contient l'objet de contexte (`ctx`), ou certaines données accessibles à tous les résolveurs effectuant une opération particulière. Par exemple, il peut contenir des données d'autorisation, les noms de champs en cours de résolution, etc. L'instruction return exécute une`Scan`opération (voiricipour des exemples). Comme nous travaillons avec DynamoDB, nous sommes autorisés à utiliser certaines des opérations de ce service. L'analyse effectue une extraction de base de tous les éléments de notre tableau. Le résultat de cette opération est stocké dans l'objet de contexte sous la forme d'un`result`conteneur avant d'être transmis au gestionnaire de réponses. Le`request`est exécuté avant la réponse dans le pipeline.
- `response`: le gestionnaire de réponses qui renvoie le résultat du`request`. L'argument est l'objet de contexte mis à jour et l'instruction de retour

estctx.prev.result. À ce stade du guide, cette valeur ne vous est peut-être pas familière.ctxfait référence à l'objet de contexte.prevfait référence à l'opération précédente dans le pipeline, qui était notrerequest. Leresultcontient le ou les résultats du résolveur lorsqu'il se déplace dans le pipeline. Si vous mettez tout cela ensemble,ctx.prev.resultrenvoie le résultat de la dernière opération effectuée, qui était le gestionnaire de demandes.

d. ChoisissezCréezune fois que tu auras terminé.

7. De retour sur l'écran du résolveur, sousFonctions, choisissez leAjouter une fonctionmenu déroulant et ajoutez votre fonction à votre liste de fonctions.
8. ChoisissezEnregistrerpour mettre à jour le résolveur.

CLI

Pour ajouter votre fonction

- Créez une fonction pour votre résolveur de pipeline à l'aide ducreate-functioncommande.

  Vous devez entrer quelques paramètres pour cette commande particulière :

  1. Leapi-iddе votre API.
  2. Lenamede la fonction dans leAWS AppSyncconsole.
  3. Ledata-source-name, ou le nom de la source de données que la fonction utilisera. Il doit déjà être créé et lié à votre API GraphQL dans leAWS AppSyncservice.
  4. Leruntime, ou environnement et langage de la fonction. PourJavaScript, le nom doit êtreAPPSYNC_JS, et le temps d'exécution,1.0.0.
  5. Lecode, ou les gestionnaires de demandes et de réponses de votre fonction. Bien que vous puissiez le saisir manuellement, il est beaucoup plus facile de l'ajouter à un fichier .txt (ou dans un format similaire), puis de le transmettre comme argument.

  > **Note**
  >
  > Notre code de requête se trouvera dans un fichier transmis en argument :
  >
  > ```
  > import { util } from '@aws-appsync/utils';
  > ```

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

Voici un exemple de commande :

```
aws appsync create-function \
--api-id abcdefghijklmnopqrstuvwxyz \
--name get_posts_func_1 \
--data-source-name table-for-posts \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file://~/path/to/file/{filename}.{fileType}
```

Une sortie sera renvoyée dans la CLI. Voici un exemple:

```
{
    "functionConfiguration": {
        "functionId": "ejglgvmcabdn7lx75ref4qeig4",
        "functionArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/functions/ejglgvmcabdn7lx75ref4qeig4",
        "name": "get_posts_func_1",
        "dataSourceName": "table-for-posts",
        "maxBatchSize": 0,
        "runtime": {
            "name": "APPSYNC_JS",
            "runtimeVersion": "1.0.0"
        },
        "code": "Code output goes here"
    }
```

```
}
```

> **Note**
>
> Assurez-vous d'enregistrer le`functionId`quelque part car cela sera utilisé pour attacher la fonction au résolveur.

Pour créer votre résolveur

- Créez une fonction de pipeline pour`Query`en exécutant le`create-resolver`commande.

  Vous devez entrer quelques paramètres pour cette commande particulière :

  1. Le`api-id`de votre API.
  2. Le`type-name`, ou le type d'objet spécial de votre schéma (requête, mutation, abonnement).
  3. Le`field-name`, ou l'opération sur le terrain dans le type d'objet spécial auquel vous souhaitez associer le résolveur.
  4. Le`kind`, qui spécifie un résolveur d'unité ou de pipeline. Réglez ce paramètre sur`PIPELINE`pour activer les fonctions de pipeline.
  5. Le`pipeline-config`, ou la ou les fonctions à associer au résolveur. Assurez-vous de connaître le`functionId`valeurs de vos fonctions. L'ordre d'inscription est important.
  6. Le`runtime`, qui était`APPSYNC_JS`(JavaScript). Le`runtimeVersion`est actuellement`1.0.0`.
  7. Le`code`, qui contient les gestionnaires d'étapes avant et après.

  > **Note**
  >
  > Notre code de requête se trouvera dans un fichier transmis en argument :

  ```
  import { util } from '@aws-appsync/utils';

  /**
   * Sends a request to `put` an item in the DynamoDB data source
   */
  export function request(ctx) {
    const { id, ...values } = ctx.args;
  ```

```
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}

/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}
```

Voici un exemple de commande :

```
aws appsync create-resolver \
--api-id abcdefghijklmnopqrstuvwxyz \
--type-name Query \
--field-name getPost \
--kind PIPELINE \
--pipeline-config functions=ejglgvmcabdn7lx75ref4qeig4 \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}
```

Une sortie sera renvoyée dans la CLI. Voici un exemple:

```
{
    "resolver": {
        "typeName": "Mutation",
        "fieldName": "getPost",
        "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Mutation/resolvers/getPost",
        "kind": "PIPELINE",
        "pipelineConfig": {
            "functions": [
                "ejglgvmcabdn7lx75ref4qeig4"
            ]
        },
        "maxBatchSize": 0,
        "runtime": {
```

```
            "name": "APPSYNC_JS",
            "runtimeVersion": "1.0.0"
        },
        "code": "Code output goes here"
    }
}
```

CDK

> **Tip**
>
> Avant d'utiliser le CDK, nous vous recommandons de consulter les CDKdocumentation officielleainsi queAWS AppSyncc'estRéférence CDK.
> Les étapes répertoriées ci-dessous ne montreront qu'un exemple général de l'extrait utilisé pour ajouter une ressource particulière. C'estpascensé être une solution fonctionnelle dans votre code de production. Nous partons également du principe que vous disposez déjà d'une application fonctionnelle.

Une application de base nécessite les éléments suivants :

1. Directives d'importation de services
2. Code de schéma
3. Générateur de sources de données
4. Code de fonction
5. Code du résolveur

À partir duConception de votre schémaetJoindre une source de donnéessections, nous savons que le fichier de pile inclura les directives d'importation du formulaire :

```
import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'
import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
```

> **Note**
>
> Dans les sections précédentes, nous avons uniquement indiqué comment importerAWS AppSyncconstructions. Dans le code réel, vous devrez importer davantage de services

juste pour exécuter l'application. Dans notre exemple, si nous devions créer une application CDK très simple, nous importerions au moins leAWS AppSyncservice ainsi que notre source de données, qui était une table DynamoDB. Nous aurions également besoin d'importer des constructions supplémentaires pour déployer l'application :

```
import * as cdk from 'aws-cdk-lib';
import * as appsync from 'aws-cdk-lib/aws-appsync';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
import { Construct } from 'constructs';
```

Pour résumer chacune d'entre elles :

- `import * as cdk from 'aws-cdk-lib';`: Cela vous permet de définir votre application CDK et des constructions telles que la pile. Il contient également des fonctions utilitaires utiles pour notre application, telles que la manipulation des métadonnées. Si vous connaissez cette directive d'importation, mais que vous vous demandez pourquoi la bibliothèque principale cdk n'est pas utilisée ici, consultez leMigrationpage.
- `import * as appsync from 'aws-cdk-lib/aws-appsync';`: Ceci importe leAWS AppSyncservice.
- `import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';`: Ceci importe leService DynamoDB.
- `import { Construct } from 'constructs';`: Nous en avons besoin pour définir la racineconstruire.

Le type d'importation dépend des services que vous appelez. Nous vous recommandons de consulter la documentation du CDK pour obtenir des exemples. Le schéma en haut de la page sera un fichier distinct dans votre application CDK sous la forme d'un `.graphql`fichier. Dans le fichier de pile, nous pouvons l'associer à un nouveau GraphQL sous la forme :

```
const add_api = new appsync.GraphqlApi(this, 'graphQL-example', {
  name: 'my-first-api',
  schema: appsync.SchemaFile.fromAsset(path.join(__dirname, 'schema.graphql')),
});
```

> **Note**
>
> Dans le champ d'applicationadd_api, nous ajoutons une nouvelle API GraphQL à l'aide dunewmot clé suivi deappsync.GraphqlApi(scope: Construct, id: string , props: GraphqlApiProps). Notre champ d'application estthis, l'identifiant CFN estgraphQL-example, et nos accessoires sontmy-first-api(nom de l'API dans la console) etschema.graphql(le chemin absolu vers le fichier de schéma).

Pour ajouter une source de données, vous devez d'abord ajouter votre source de données à la pile. Ensuite, vous devez l'associer à l'API GraphQL en utilisant la méthode spécifique à la source. L'association se produira lorsque vous ferez fonctionner votre résolveur. En attendant, prenons un exemple en créant la table DynamoDB à l'aide dedynamodb.Table:

```
const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
  partitionKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
});
```

> **Note**
>
> Si nous devions l'utiliser dans notre exemple, nous ajouterions une nouvelle table DynamoDB avec l'identifiant CFN deposts-tableet une clé de partition deid (S).

Ensuite, nous devons implémenter notre résolveur dans le fichier de pile. Voici un exemple de requête simple qui recherche tous les éléments d'une table DynamoDB :

```
const add_func = new appsync.AppsyncFunction(this, 'func-get-posts', {
  name: 'get_posts_func_1',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('table-for-posts', add_ddb_table),
  code: appsync.Code.fromInline(`
      export function request(ctx) {
        return { operation: 'Scan' };
      }
```

```
      export function response(ctx) {
        return ctx.result.items;
      }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});

new appsync.Resolver(this, 'pipeline-resolver-get-posts', {
  add_api,
  typeName: 'Query',
  fieldName: 'getPost',
  code: appsync.Code.fromInline(`
      export function request(ctx) {
        return {};
      }

      export function response(ctx) {
        return ctx.prev.result;
      }
 `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
  pipelineConfig: [add_func],
});
```

Note

Tout d'abord, nous avons créé une fonction appeléeadd_func. Cet ordre de création peut sembler un peu paradoxal, mais vous devez créer les fonctions dans votre résolveur de pipeline avant de créer le résolveur lui-même. Une fonction se présente sous la forme suivante :

```
AppsyncFunction(scope: Construct, id: string, props: AppsyncFunctionProps)
```

Notre objectif étaitthis, votre identifiant CFN étaitfunc-get-posts, et nos accessoires contenaient les détails de la fonction réelle. À l'intérieur des accessoires, nous avons inclus :

- Lenamede la fonction qui sera présente dansAWS AppSyncconsole (get_posts_func_1).
- L'API GraphQL que nous avons créée précédemment (add_api).

- La source de données ; c'est le point où nous lions la source de données à la valeur de l'API GraphQL, puis nous l'associons à la fonction. Nous prenons le tableau que nous avons créé (`add_ddb_table`) et attachez-le à l'API GraphQL (`add_api`) en utilisant l'un des`GraphqlApi`méthodes (`addDynamoDbDataSource`). La valeur de l'identifiant (`table-for-posts`) est le nom de la source de données dansAWS AppSyncconsole. Pour obtenir la liste des méthodes spécifiques à la source, consultez les pages suivantes :
  - DynamoDbDataSource
  - EventBridgeDataSource
  - HttpDataSource
  - LambdaDataSource
  - NoneDataSource
  - OpenSearchDataSource
  - RdsDataSource
- Le code contient les gestionnaires de demandes et de réponses de notre fonction, à savoir un simple scan et un retour.
- Le runtime indique que nous voulons utiliser le runtime APPSYNC_JS version 1.0.0. Notez qu'il s'agit actuellement de la seule version disponible pour APPSYNC_JS.

Ensuite, nous devons associer la fonction au résolveur de pipeline. Nous avons créé notre résolveur en utilisant le formulaire :

```
Resolver(scope: Construct, id: string, props: ResolverProps)
```

Notre objectif étaitthis, votre identifiant CFN étaitpipeline-resolver-get-posts, et nos accessoires contenaient les détails de la fonction réelle. À l'intérieur des accessoires, nous avons inclus :

- L'API GraphQL que nous avons créée précédemment (`add_api`).
- Le nom du type d'objet spécial ; il s'agit d'une opération de requête, nous avons donc simplement ajouté la valeur`Query`.
- Le nom du champ (`getPost`) est le nom du champ dans le schéma sous`Query`type.
- Le code contient vos gestionnaires avant et après. Notre exemple renvoie simplement les résultats présents dans le contexte une fois que la fonction a effectué son opération.

- Le runtime indique que nous voulons utiliser le runtime APPSYNC_JS version 1.0.0. Notez qu'il s'agit actuellement de la seule version disponible pour APPSYNC_JS.
- La configuration du pipeline contient la référence à la fonction que nous avons créée (`add_func`).

Pour résumer ce qui s'est passé dans cet exemple, vous avez vu unAWS AppSyncfonction qui implémente un gestionnaire de requêtes et de réponses. La fonction était chargée d'interagir avec votre source de données. Le gestionnaire de demandes a envoyé un`Scan`opération pourAWS AppSync, en lui indiquant l'opération à effectuer sur votre source de données DynamoDB. Le gestionnaire de réponses a renvoyé la liste des éléments (`ctx.result.items`). La liste des éléments a ensuite été mappée sur`Post`Tapez GraphQL automatiquement.

Création de résolveurs de mutations de base

Cette section vous montrera comment créer un résolveur de mutations de base.

Console

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
   a. Dans leTableau de bord des API, choisissez votre API GraphQL.
   b. Dans leBarre latérale, choisissezSchéma.
2. En vertu duRésolveurssection et leMutationtapez, choisissezJoindreà côté de votre terrain.

   > **Note**
   >
   > Dans notre exemple, nous attachons un résolveur pour`createPost`, qui ajoute un`Post`objet sur notre table. Supposons que nous utilisions la même table DynamoDB que celle de la section précédente. Sa clé de partition est définie sur`id`et est vide.

3. Sur leAttacher un résolveurpage, sousType de résolveur, choisissez`pipeline resolvers`. Pour rappel, vous pouvez trouver plus d'informations sur les résolveursici. PourTemps d'exécution du résolveur, choisissezAPPSYNC_JSpour activer leJavaScripttemps d'exécution.
4. Vous pouvez activermise en cachepour cette API. Nous vous recommandons de désactiver cette fonctionnalité pour le moment. Sélectionnez Create (Créer).

5. ChoisissezAjouter une fonction, puis choisissezCréer une nouvelle fonction. Vous pouvez également voir unCréer une fonctionbouton pour choisir à la place.

   a. Choisissez votre source de données . Il doit s'agir de la source dont vous allez manipuler les données avec la mutation.

   b. Entrez un`Function name`.

   c. En dessousCode de fonction, vous devez implémenter le comportement de la fonction. Comme il s'agit d'une mutation, la demande effectuera idéalement une opération de changement d'état sur la source de données invoquée. Le résultat sera traité par la fonction de réponse.

   > **Note**
   >
   > `createPost`est en train d'ajouter, ou de « mettre », un nouveau`Post`dans le tableau avec nos paramètres comme données. Nous pourrions ajouter quelque chose comme ceci :
   >
   > ```
   > import { util } from '@aws-appsync/utils';
   >
   > /**
   >  * Sends a request to `put` an item in the DynamoDB data source
   >  */
   > export function request(ctx) {
   >   return {
   >     operation: 'PutItem',
   >     key: util.dynamodb.toMapValues({id: util.autoId()}),
   >     attributeValues: util.dynamodb.toMapValues(ctx.args.input),
   >   };
   > }
   >
   > /**
   >  * returns the result of the `put` operation
   >  */
   > export function response(ctx) {
   >   return ctx.result;
   > }
   > ```
   >
   > Dans cette étape, nous avons également ajouté`request`et`response`fonctions :

- `request`: Le gestionnaire de requêtes accepte le contexte comme argument. L'instruction return du gestionnaire de demandes exécute une`PutItem`commande, qui est une opération DynamoDB intégrée (voirici`ici`pour des exemples). Le`PutItem`la commande ajoute un`Post`objet vers notre table DynamoDB en prenant la partition`key`valeur (générée automatiquement par`util.autoid()`) et`attributes`à partir de l'argument contextuel saisi (ce sont les valeurs que nous transmettrons dans notre requête). Le`key`est le`id`et`attributes`sont les`date`et`title`arguments de champ. Ils sont tous deux préformatés via`util.dynamodb.toMapValues`assistant pour travailler avec la table DynamoDB.
- `response`: La réponse accepte le contexte mis à jour et renvoie le résultat du gestionnaire de demandes.

d. ChoisissezCréezune fois que tu auras terminé.

6. De retour sur l'écran du résolveur, sousFonctions, choisissez leAjouter une fonctionmenu déroulant et ajoutez votre fonction à votre liste de fonctions.
7. ChoisissezEnregistrerpour mettre à jour le résolveur.

CLI

Pour ajouter votre fonction

- Créez une fonction pour votre résolveur de pipeline à l'aide du`create-function`commande.

  Vous devez entrer quelques paramètres pour cette commande particulière :

  1. Le`api-id`de votre API.
  2. Le`name`de la fonction dans leAWS AppSyncconsole.
  3. Le`data-source-name`, ou le nom de la source de données que la fonction utilisera. Il doit déjà être créé et lié à votre API GraphQL dans leAWS AppSyncservice.
  4. Le`runtime`, ou environnement et langage de la fonction. PourJavaScript, le nom doit être`APPSYNC_JS`, et le temps d'exécution,`1.0.0`.

5. `Lecode`, ou les gestionnaires de demandes et de réponses de votre fonction. Bien que vous puissiez le saisir manuellement, il est beaucoup plus facile de l'ajouter à un fichier .txt (ou dans un format similaire) puis de le transmettre comme argument.

> **Note**
>
> Notre code de requête se trouvera dans un fichier transmis en argument :
>
> ```
> import { util } from '@aws-appsync/utils';
>
> /**
>  * Sends a request to `put` an item in the DynamoDB data source
>  */
> export function request(ctx) {
>   return {
>     operation: 'PutItem',
>     key: util.dynamodb.toMapValues({id: util.autoId()}),
>     attributeValues: util.dynamodb.toMapValues(ctx.args.input),
>   };
> }
>
> /**
>  * returns the result of the `put` operation
>  */
> export function response(ctx) {
>   return ctx.result;
> }
> ```

Voici un exemple de commande :

```
aws appsync create-function \
--api-id abcdefghijklmnopqrstuvwxyz \
--name add_posts_func_1 \
--data-source-name table-for-posts \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}
```

Une sortie sera renvoyée dans la CLI. Voici un exemple:

```
{
    "functionConfiguration": {
        "functionId": "vulcmbfcxffiram63psb4dduoa",
        "functionArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/functions/vulcmbfcxffiram63psb4dduoa",
        "name": "add_posts_func_1",
        "dataSourceName": "table-for-posts",
        "maxBatchSize": 0,
        "runtime": {
            "name": "APPSYNC_JS",
            "runtimeVersion": "1.0.0"
        },
        "code": "Code output foes here"
    }
}
```

> **Note**
>
> Assurez-vous d'enregistrer le`functionId`quelque part car cela sera utilisé pour attacher la fonction au résolveur.

Pour créer votre résolveur

- Créez une fonction de pipeline pour`Mutation`en exécutant le`create-resolver`commande.

  Vous devez entrer quelques paramètres pour cette commande particulière :

  1. Le`api-id`de votre API.
  2. Le`type-name`, ou le type d'objet spécial de votre schéma (requête, mutation, abonnement).
  3. Le`field-name`, ou l'opération sur le terrain dans le type d'objet spécial auquel vous souhaitez associer le résolveur.
  4. Le`kind`, qui spécifie un résolveur d'unité ou de pipeline. Réglez ce paramètre sur`PIPELINE`pour activer les fonctions de pipeline.
  5. Le`pipeline-config`, ou la ou les fonctions à associer au résolveur. Assurez-vous de connaître le`functionId`valeurs de vos fonctions. L'ordre d'inscription est important.

6. `Leruntime`, qui étaitAPPSYNC_JS(JavaScript). `LeruntimeVersion`est actuellement`1.0.0`.
7. `Lecode`, qui contient les étapes avant et après.

> **Note**
>
> Notre code de requête se trouvera dans un fichier transmis en argument :

```
import { util } from '@aws-appsync/utils';

/**
 * Sends a request to `put` an item in the DynamoDB data source
 */
export function request(ctx) {
  const { id, ...values } = ctx.args;
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}

/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}
```

Voici un exemple de commande :

```
aws appsync create-resolver \
--api-id abcdefghijklmnopqrstuvwxyz \
--type-name Mutation \
--field-name createPost \
--kind PIPELINE \
--pipeline-config functions=vulcmbfcxffiram63psb4dduoa \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}
```

Une sortie sera renvoyée dans la CLI. Voici un exemple:

```
{
    "resolver": {
        "typeName": "Mutation",
        "fieldName": "createPost",
        "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Mutation/resolvers/createPost",
        "kind": "PIPELINE",
        "pipelineConfig": {
            "functions": [
                "vulcmbfcxffiram63psb4dduoa"
            ]
        },
        "maxBatchSize": 0,
        "runtime": {
            "name": "APPSYNC_JS",
            "runtimeVersion": "1.0.0"
        },
        "code": "Code output goes here"
    }
}
```

CDK

> **Tip**
>
> Avant d'utiliser le CDK, nous vous recommandons de consulter les CDKdocumentation officielleainsi queAWS AppSyncc'estRéférence CDK.
> Les étapes répertoriées ci-dessous ne montreront qu'un exemple général de l'extrait utilisé pour ajouter une ressource particulière. C'estpascensé être une solution fonctionnelle dans votre code de production. Nous partons également du principe que vous disposez déjà d'une application fonctionnelle.

- Pour effectuer une mutation, en supposant que vous participez au même projet, vous pouvez l'ajouter au fichier de pile comme dans la requête. Voici une fonction modifiée et un résolveur pour une mutation qui ajoute un nouveau`Post`à la table :

```
const add_func_2 = new appsync.AppsyncFunction(this, 'func-add-post', {
  name: 'add_posts_func_1',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('table-for-posts-2', add_ddb_table),
      code: appsync.Code.fromInline(`
          export function request(ctx) {
            return {
              operation: 'PutItem',
              key: util.dynamodb.toMapValues({id: util.autoId()}),
              attributeValues: util.dynamodb.toMapValues(ctx.args.input),
            };
          }

          export function response(ctx) {
            return ctx.result;
          }
      `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});

new appsync.Resolver(this, 'pipeline-resolver-create-posts', {
  add_api,
  typeName: 'Mutation',
  fieldName: 'createPost',
      code: appsync.Code.fromInline(`
          export function request(ctx) {
            return {};
          }

          export function response(ctx) {
            return ctx.prev.result;
          }
      `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
  pipelineConfig: [add_func_2],
});
```

> **Note**
>
> Comme cette mutation et la requête sont structurées de la même manière, nous allons simplement expliquer les modifications que nous avons apportées pour effectuer la mutation.
> Dans la fonction, nous avons changé l'identifiant CFN en`func-add-post`et le nom de`add_posts_func_1`pour refléter le fait que nous ajoutons`Posts`à table.
> Dans la source de données, nous avons créé une nouvelle association avec notre table (`add_ddb_table`) dans leAWS AppSyncconsole en tant que`table-for-posts-2`parce que le`addDynamoDbDataSource`méthode l'exige. N'oubliez pas que cette nouvelle association utilise toujours la même table que celle que nous avons créée précédemment, mais que nous y avons désormais deux connexions dans leAWS AppSyncconsole : une pour la requête en tant que`table-for-posts`et un pour la mutation en tant que`table-for-posts-2`. Le code a été modifié pour ajouter un`Post`en générant son`id`valeur automatique et acceptation de la saisie d'un client pour le reste des champs.
> Dans le résolveur, nous avons changé la valeur de l'identifiant en`pipeline-resolver-create-posts`pour refléter le fait que nous ajoutons`Posts`à table. Pour refléter la mutation dans le schéma, le nom du type a été changé en`Mutation`, et le nom,`createPost`. La configuration du pipeline a été définie sur notre nouvelle fonction de mutation`add_func_2`.

Pour résumer ce qui se passe dans cet exemple,AWS AppSyncconvertit automatiquement les arguments définis dans`createPost`champ de votre schéma GraphQL dans les opérations DynamoDB. L'exemple stocke les enregistrements dans DynamoDB à l'aide d'une clé de`id`, qui est automatiquement créé à l'aide de notre`util.autoId()`assistant. Tous les autres champs que vous transmettez aux arguments de contexte (`ctx.args.input`) à partir de demandes faites dans leAWS AppSyncconsole ou autre seront stockés en tant qu'attributs de la table. La clé et les attributs sont automatiquement mappés dans un format DynamoDB compatible à l'aide du`util.dynamodb.toMapValues(values)`assistant.

AWS AppSync prend également en charge les workflows de test et de débogage pour la modification des résolveurs. Vous pouvez utiliser une maquette`context`objet pour voir la valeur transformée du modèle avant de l'invoquer. Vous pouvez éventuellement consulter la demande complète adressée

à une source de données de manière interactive lorsque vous exécutez une requête. Pour plus d'informations, voirTestez et déboguez les résolveurs (JavaScript)etSurveillance et journalisation.

Résolveurs avancés

Si vous suivez la section de pagination facultative dansConception de votre schéma, vous devez toujours ajouter votre résolveur à votre demande pour utiliser la pagination. Dans notre exemple, nous avons utilisé une pagination de requête appeléegetPostsde ne retourner qu'une partie des articles demandés à la fois. Le code de notre résolveur sur ce champ peut ressembler à ceci :

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  const { limit = 20, nextToken } = ctx.args;
  return { operation: 'Scan', limit, nextToken };
}

/**
 * @returns the result of the `put` operation
 */
export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

Dans la demande, nous transmettons le contexte de la demande. Notrelimitest*20*, ce qui signifie que nous retournons jusqu'à 20Postsdans la première requête. NotrenextTokenle curseur est fixé au premierPostentrée dans la source de données. Ils sont transmis aux arguments. La demande effectue ensuite un scan à partir du premierPostjusqu'au nombre limite de numérisation. La source de données stocke le résultat dans le contexte, qui est transmis à la réponse. La réponse renvoie lePostsil est récupéré, puis définit lenextTokenest réglé surPostentrée juste après la limite. La demande suivante est envoyée pour faire exactement la même chose, mais en commençant par le décalage juste après la première requête. N'oubliez pas que ces types de demandes sont effectués de manière séquentielle et non en parallèle.

Testez et déboguez les résolveurs (JavaScript)

AWS AppSyncexécute des résolveurs sur un champ GraphQL par rapport à une source de données. Lorsque vous travaillez avec des résolveurs de pipeline, les fonctions interagissent avec vos sources de données. Comme décrit dans leJavaScriptvue d'ensemble des résolveurs, les fonctions

communiquent avec les sources de données à l'aide de gestionnaires de requêtes et de réponses écrits enJavaScriptet en courant sur leAPPSYNC_JStemps d'exécution. Cela vous permet de fournir une logique et des conditions personnalisées avant et après la communication avec la source de données.

Pour aider les développeurs à écrire, tester et déboguer ces résolveurs, leAWS AppSyncLa console fournit également des outils pour créer une requête et une réponse GraphQL avec des données fictives jusqu'au résolveur de champs individuel. En outre, vous pouvez effectuer des requêtes, des mutations et des abonnements dansAWS AppSyncconsole et consultez un flux de journal détaillé de l'ensemble de la demande d'AmazonCloudWatch. Cela inclut les résultats de la source de données.

Tests à l'aide de données fictives

Lorsqu'un résolveur GraphQL est invoqué, il contient un`context`objet contenant des informations pertinentes sur la demande. Ces informations incluent les arguments d'un client, les informations d'identité et les données du champ GraphQL parent. Il stocke également les résultats de la source de données, qui peuvent être utilisés dans le gestionnaire de réponses. Pour plus d'informations sur cette structure et les utilitaires d'assistance disponibles à utiliser lors de la programmation, consultez leRéférence de l'objet contextuel du résolveur.

Lorsque vous écrivez ou modifiez une fonction de résolution, vous pouvez transmettre unse moqueroucontexte de testobjet dans l'éditeur de console. Cela vous permet de voir comment les gestionnaires de demandes et de réponses s'évaluent sans réellement se heurter à une source de données. Par exemple, vous pouvez transmettre un argument de test `firstname: Shaggy` et voir comment il est analysé lors de l'utilisation de `ctx.args.firstname` dans votre modèle de code. Vous pouvez également tester l'évaluation de n'importe quelle annotation d'utilitaire telle que `util.autoId()` ou `util.time.nowISO8601()`.

Tester les résolveurs

Cet exemple utilisera leAWS AppSyncconsole pour tester les résolveurs.

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
   a. Dans leTableau de bord des API, choisissez votre API GraphQL.
   b. Dans leBarre latérale, choisissezFonctions.
2. Choisissez une fonction existante.
3. Au sommet duFonction de mise à jourpage, choisissezSélectionnez le contexte de test, puis choisissezCréation d'un nouveau contexte.

4. Sélectionnez un exemple d'objet de contexte ou renseignez le JSON manuellement dans leConfiguration du contexte de testfenêtre ci-dessous.
5. Entrez unNom du contexte du texte.
6. Choisissez le bouton Enregistrer.
7. Pour évaluer votre résolveur à l'aide de cet objet de contexte simulé, choisissez Exécuter test

Pour un exemple plus pratique, supposons que vous ayez une application stockant un type GraphQL deDogqui utilise la génération automatique d'identifiants pour les objets et les stocke dans Amazon DynamoDB. Vous souhaitez également écrire des valeurs à partir des arguments d'une mutation GraphQL et n'autoriser que certains utilisateurs à voir une réponse. L'extrait suivant montre à quoi pourrait ressembler le schéma :

```
type Dog {
  breed: String
  color: String
}

type Mutation {
  addDog(firstname: String, age: Int): Dog
}
```

Vous pouvez écrire unAWS AppSyncfonction et ajoutez-la à votre`addDog`résolveur pour gérer la mutation. Pour tester votreAWS AppSyncfonction, vous pouvez remplir un objet de contexte comme dans l'exemple suivant. Le suivant possède les arguments `name` et `age` du client, ainsi qu'un `username` renseigné dans l'objet `identity` :

```
{
    "arguments" : {
        "firstname": "Shaggy",
        "age": 4
    },
    "source" : {},
    "result" : {
        "breed" : "Miniature Schnauzer",
        "color" : "black_grey"
    },
    "identity": {
        "sub" : "uuid",
        "issuer" : " https://cognito-idp.{region}.amazonaws.com/{userPoolId}",
```

```
        "username" : "Nadia",
        "claims" : { },
        "sourceIp" :[  "x.x.x.x" ],
        "defaultAuthStrategy" : "ALLOW"
    }
}
```

Vous pouvez tester votreAWS AppSyncfonction à l'aide du code suivant :

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id: util.autoId() }),
    attributeValues: util.dynamodb.toMapValues(ctx.args),
  };
}

export function response(ctx) {
  if (ctx.identity.username === 'Nadia') {
    console.log("This request is allowed")
    return ctx.result;
  }
  util.unauthorized();
}
```

Le gestionnaire de demandes et de réponses évalué contient les données de votre objet de contexte de test et la valeur générée par`util.autoId()`. De plus, si vous deviez modifier `username` avec une valeur autre que `Nadia`, les résultats ne seront pas renvoyés, car le contrôle d'autorisation échouerait. Pour plus d'informations sur le contrôle d'accès détaillé, voirCas d'utilisation des autorisations.

Tester les gestionnaires de demandes et de réponses avecAWS AppSyncles API

Vous pouvez utiliser le`EvaluateCode`Commande API pour tester à distance votre code avec des données simulées. Pour commencer à utiliser la commande, assurez-vous d'avoir ajouté le`appsync:evaluateMappingCode`autorisation de respecter votre politique. Par exemple :

```
{
    "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateCode",
            "Resource": "arn:aws:appsync:<region>:<account>:*"
        }
    ]
}
```

Vous pouvez utiliser la commande à l'aide duAWS CLIouAWSSDK. Par exemple, prenons leDogschéma et sesAWS AppSyncgestionnaires de demandes et de réponses de fonctions de la section précédente. À l'aide de l'interface de ligne de commande de votre station locale, enregistrez le code dans un fichier nommé`code.js`, puis enregistrez le`context`objet vers un fichier nommé`context.json`. Depuis votre shell, exécutez la commande suivante :

```
$ aws appsync evaluate-code \
  --code file://code.js \
  --function response \
  --context file://context.json \
  --runtime name=APPSYNC_JS,runtimeVersion=1.0.0
```

La réponse contient un`evaluationResult`contenant la charge utile renvoyée par votre gestionnaire. Il contient également un`logs`objet, qui contient la liste des journaux générés par votre gestionnaire lors de l'évaluation. Cela permet de déboguer facilement l'exécution de votre code et de consulter les informations relatives à votre évaluation pour vous aider à résoudre le problème. Par exemple :

```
{
    "evaluationResult": "{\"breed\":\"Miniature Schnauzer\",\"color\":\"black_grey\"}",
    "logs": [
        "INFO - code.js:13:5: \"This request is allowed\""
    ]
}
```

Le`evaluationResult`peut être analysé au format JSON, ce qui donne :

```
{
  "breed": "Miniature Schnauzer",
  "color": "black_grey"
}
```

À l'aide du SDK, vous pouvez facilement intégrer des tests issus de votre suite de tests préférée pour valider le comportement de vos gestionnaires. Nous vous recommandons de créer des tests à l'aide duFramework de test Jest, mais n'importe quelle suite de tests fonctionne. L'extrait suivant montre une exécution de validation hypothétique. Notez que nous nous attendons à ce que la réponse d'évaluation soit un JSON valide, nous utilisons donc`JSON.parse`pour récupérer le JSON à partir de la chaîne de réponse :

```
const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })
const runtime = {name:'APPSYNC_JS',runtimeVersion:'1.0.0')

test('request correctly calls DynamoDB', async () => {
  const code = fs.readFileSync('./code.js', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateCode({ code, context, runtime, function:
 'request' }).promise()
  const result = JSON.parse(response.evaluationResult)

  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})
```

Cela donne le résultat suivant :

```
Ran all test suites.
> jest

PASS ./index.test.js
# request correctly calls DynamoDB (543 ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 totalTime: 1.511 s, estimated 2 s
```

Débogage d'une requête en direct

Rien ne remplace un test de bout en bout et une journalisation pour déboguer une application de production.AWS AppSyncvous permet de consigner les erreurs et les détails complets des demandes à l'aide d'AmazonCloudWatch. De plus, vous pouvez utiliser leAWS AppSyncconsole pour tester les

requêtes, les mutations et les abonnements GraphQL, et les données du journal de diffusion en direct pour chaque requête sont renvoyées dans l'éditeur de requêtes pour un débogage en temps réel. Pour les abonnements, les journaux affichent les informations de temps de connexion.

Pour ce faire, vous devez disposer d'AmazonCloudWatchjournaux activés à l'avance, comme décrit dansSurveillance et journalisation. Ensuite, dans leAWS AppSyncconsole, choisissezRequêtesonglet, puis entrez une requête GraphQL valide. Dans la section inférieure droite, cliquez et faites glisser leJournauxfenêtre pour ouvrir la vue des journaux. À l'aide de la flèche « Play » en haut de la page, exécutez votre requête GraphQL. Dans quelques instants, l'intégralité de vos journaux de demandes et de réponses relatifs à l'opération est diffusée dans cette section et vous pouvez les consulter dans la console.

Résolveurs de pipelines (JavaScript)

AWS AppSyncexécute des résolveurs sur un champ GraphQL. Dans certains cas, les applications nécessitent l'exécution de plusieurs opérations pour résoudre un seul champ GraphQL. Avec les résolveurs de pipeline, les développeurs peuvent désormais composer des opérations appelées fonctions et les exécuter en séquence. Les résolveurs de pipeline sont utiles pour les applications qui, par exemple, doivent effectuer un contrôle d'autorisation avant d'extraire des données pour un champ.

Pour plus d'informations sur l'architecture d'unJavaScriptrésolveur de pipeline, voir leJavaScriptvue d'ensemble des résolveurs.

Création d'un résolveur de pipeline

Dans leAWS AppSyncconsole, accédez àSchémapage.

Enregistrez le schéma suivant :

```
schema {
    query: Query
    mutation: Mutation
}

type Mutation {
    signUp(input: Signup): User
}

type Query {
    getUser(id: ID!): User
```

```
}

input Signup {
    username: String!
    email: String!
}

type User {
    id: ID!
    username: String
    email: AWSEmail
}
```

Nous allons raccorder un résolveur de pipeline au champ signUp sur le type Mutation. Dans leMutationtapez sur le côté droit, choisissezJoindreà côté du`signUp`champ de mutation. Réglez le résolveur sur`pipeline resolver`et le`APPSYNC_JS`runtime, puis créez le résolveur.

Notre résolveur de pipeline inscrit un utilisateur en validant tout d'abord l'adresse e-mail saisie, puis en enregistrant l'utilisateur dans le système. Nous allons encapsuler la validation de l'e-mail dans unValider l'e-mailfonction et sauvegarde de l'utilisateur dans unEnregistrer l'utilisateurfonction. La fonction validateEmail s'exécute d'abord, puis si l'e-mail est valide, alors la fonction saveUser s'exécute.

Le flux d'exécution sera le suivant :

1. Gestionnaire de requêtes du résolveur Mutation.signup
2. Fonction validateEmail
3. Fonction saveUser
4. Gestionnaire de réponses du résolveur Mutation.signup

Parce que nous allons probablement réutiliser leValider l'e-mailfonctionne dans d'autres résolveurs de notre API, nous voulons éviter d'accéder`ctx.args`car ils changeront d'un champ GraphQL à l'autre. Au lieu de cela, nous pouvons utiliser le `ctx.stash` pour stocker l'attribut e-mail depuis l'argument de champ de saisie `signUp(input: Signup)`.

Mettez à jour le code de votre résolveur en remplaçant vos fonctions de demande et de réponse :

```
export function request(ctx) {
    ctx.stash.email = ctx.args.input.email
    return {};
```

```
}

export function response(ctx) {
    return ctx.prev.result;
}
```

ChoisissezCréezouEnregistrerpour mettre à jour le résolveur.

Créer une fonction

Depuis la page du résolveur de pipeline, dans leFonctionssection, cliquez surAjouter une fonction, puisCréer une nouvelle fonction. Il est également possible de créer des fonctions sans passer par la page du résolveur ; pour ce faire, dansAWS AppSyncconsole, accédez àFonctionspage. Choisissez le bouton Créer une fonction. Créons une fonction permettant de vérifier si un e-mail est valide et provient d'un domaine spécifique. Si l'e-mail n'est pas valide, la fonction renvoie une erreur. Dans le cas contraire, elle transmet les données saisies.

Assurez-vous d'avoir créé une source de données deAUCUNEtype. Choisissez cette source de données dansNom de la source de donnéesliste. Pour lenom de la fonction, entrez`validateEmail`. Dans lecode de fonctionzone, remplacez tout par cet extrait de code :

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { email } = ctx.stash;
  const valid = util.matches(
    '^[a-zA-Z0-9_.+-]+@(?:(?:[a-zA-Z0-9-]+\.)?[a-zA-Z]+\.)?(myvaliddomain)\.com',
    email
  );
  if (!valid) {
    util.error(`"${email}" is not a valid email.`);
  }

  return { payload: { email } };
}

export function response(ctx) {
  return ctx.result;
}
```

Passez en revue vos entrées, puis choisissezCréez. Nous venons de créer notre fonction validateEmail. Répétez ces étapes pour créerEnregistrer l'utilisateurfonction avec le code suivant

(Pour des raisons de simplicité, nous utilisons unAUCUNEsource de données et prétendez que l'utilisateur a été enregistré dans le système après l'exécution de la fonction.) :

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  return ctx.prev.result;
}

export function response(ctx) {
  ctx.result.id = util.autoId();
  return ctx.result;
}
```

Nous venons de créer notreEnregistrer l'utilisateurfonction.

Ajouter une fonction à un résolveur de pipeline

Nos fonctions auraient dû être ajoutées automatiquement au résolveur de pipeline que nous venons de créer. Si ce n'était pas le cas, ou si vous avez créé les fonctions viaFonctionspage, vous pouvez cliquer surAjouter une fonctionretour sur lesignUppage de résolution pour les joindre. Ajoutez les deuxValider l'e-mailetEnregistrer l'utilisateurfonctions du résolveur. La fonction validateEmail doit être placée avant la fonction saveUser. Au fur et à mesure que vous ajoutez de nouvelles fonctions, vous pouvez utilisermonteretdéplacer vers le basdes options pour réorganiser l'ordre d'exécution de vos fonctions. Passez en revue vos modifications, puis choisissezEnregistrer.

Exécution d'une requête

Dans leAWS AppSyncconsole, accédez àRequêtespage. Dans l'explorateur, assurez-vous d'utiliser votre mutation. Si ce n'est pas le cas, choisissez`Mutation`dans la liste déroulante, puis choisissez+. Entrez la requête suivante :

```
mutation {
  signUp(input: {email: "nadia@myvaliddomain.com", username: "nadia"}) {
    id
    username
  }
}
```

Cela devrait renvoyer quelque chose comme :

```
{
  "data": {
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "username": "nadia"
    }
  }
}
```

Nous avons inscrit notre utilisateur et validé l'e-mail saisi à l'aide d'un résolveur de pipeline.

Configuration des résolveurs (VTL)

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Les résolveurs GraphQL connectent les champs d'un schéma de type à une source de données. Les résolveurs sont le mécanisme par lequel les demandes sont satisfaites. AWS AppSync peut créer et connecter automatiquement des résolveurs à partir d'un schéma ou créer un schéma et connecter des résolveurs à partir d'une table existante sans que vous ayez à écrire de code.

Résolveurs AWS AppSync utilisés JavaScript pour convertir une expression GraphQL dans un format utilisable par la source de données. Les modèles de mappage peuvent également être écrits dans le langage VTL (Apache Velocity Template Language) pour convertir une expression GraphQL dans un format utilisable par la source de données.

Cette section explique comment configurer les résolveurs à l'aide de VTL. Un guide de programmation de type didacticiel d'introduction pour l'écriture de résolveurs se trouve dans le guide de programmation du modèle de mappage Resolver, et les utilitaires d'assistance disponibles pour la programmation se trouvent dans la référence contextuelle du modèle de mappage Resolver. AWS AppSync possède également des flux de test et de débogage intégrés que vous pouvez utiliser lorsque vous modifiez ou créez à partir de zéro. Pour plus d'informations, consultez la section Tester et déboguer les résolveurs.

Nous vous recommandons de suivre ce guide avant d'essayer d'utiliser l'un des didacticiels mentionnés ci-dessus.

Dans cette section, nous expliquerons comment créer un résolveur, ajouter un résolveur pour les mutations et utiliser des configurations avancées.

Créez votre premier résolveur

En suivant les exemples des sections précédentes, la première étape consiste à créer un résolveur adapté à votre `Query` type.

Console

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
   a. Dans le tableau de bord des API, choisissez votre API GraphQL.
   b. Dans la barre latérale, choisissez Schema.
2. Sur le côté droit de la page, il y a une fenêtre appelée Resolvers. Cette zone contient une liste des types et des champs tels que définis dans votre fenêtre de schéma sur le côté gauche de la page. Vous pouvez associer des résolveurs aux champs. Par exemple, sous le type de requête, choisissez Joindre à côté du `getTodos` champ.
3. Sur la page Create Resolver, choisissez la source de données que vous avez créée dans le guide Joindre une source de données. Dans la fenêtre Configurer les modèles de mappage, vous pouvez choisir les modèles de mappage de demande et de réponse génériques à l'aide de la liste déroulante de droite ou écrire le vôtre.

> **Note**
>
> L'appariement d'un modèle de mappage de demande à un modèle de mappage de réponse est appelé résolveur d'unités. Les résolveurs d'unités sont généralement conçus pour effectuer des opérations par cœur ; nous recommandons de les utiliser uniquement pour des opérations uniques avec un petit nombre de sources de données. Pour les opérations plus complexes, nous recommandons d'utiliser des résolveurs de pipeline, qui peuvent exécuter plusieurs opérations avec plusieurs sources de données de manière séquentielle.
> Pour plus d'informations sur la différence entre les modèles de mappage de demandes et de réponses, consultez Unit Resolvers.
> Pour plus d'informations sur l'utilisation des résolveurs de pipeline, consultez la section Résolveurs de pipeline.

4. Pour les cas d'utilisation courants, la AWS AppSync console possède des modèles intégrés que vous pouvez utiliser pour obtenir des éléments à partir de sources de données (par exemple, toutes les requêtes relatives aux articles, les recherches individuelles, etc.). Par exemple, dans la version simple du schéma de Designing your schema where `getTodos` didn't have pagination, le modèle de mappage des demandes pour la liste des éléments est le suivant :

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
}
```

5. Vous avez toujours besoin d'un modèle de mappage des réponses pour accompagner la demande. La console fournit un modèle par défaut avec la valeur de transmission suivante pour les listes :

```
$util.toJson($ctx.result.items)
```

   Dans cet exemple, l'objet `context` (dont l'alias est `$ctx`) pour les listes d'éléments se présente sous la forme `$context.result.items`. Si votre opération GraphQL renvoie un seul élément, ce sera le cas. `$context.result` AWS AppSyncfournit des fonctions d'assistance pour les opérations courantes, telles que la `$util.toJson` fonction répertoriée précédemment, afin de formater correctement les réponses. Pour une liste complète des fonctions, consultez la référence de l'utilitaire de modèle de mappage Resolver.

6. Choisissez Save Resolver.

API

1. Créez un objet résolveur en appelant l'`CreateResolver`API.
2. Vous pouvez modifier les champs de votre résolveur en appelant l'`UpdateResolver`API.

CLI

1. Créez un résolveur en exécutant la `create-resolver`commande.

   Vous devez saisir 6 paramètres pour cette commande particulière :

   1. Celui `api-id` de votre API.

2. Le `type-name` type que vous souhaitez modifier dans votre schéma. Dans l'exemple de console, c'était le cas`Query`.
3. Le `field-name` champ que vous souhaitez modifier dans votre type. Dans l'exemple de console, c'était le cas`getTodos`.
4. La source `data-source-name` de données que vous avez créée dans le guide Joindre une source de données.
5. Le`request-mapping-template`, qui est le corps de la demande. Dans l'exemple de console, c'était :

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
}
```

6. Le`response-mapping-template`, qui est le corps de la réponse. Dans l'exemple de console, c'était :

```
$util.toJson($ctx.result.items)
```

Voici un exemple de commande :

```
aws appsync create-resolver --api-id abcdefghijklmnopqrstuvwxyz --type-name
 Query --field-name getTodos --data-source-name TodoTable --request-mapping-
template "{ "version" : "2017-02-28", "operation" : "Scan", }" --response-
mapping-template ""$"util.toJson("$"ctx.result.items)"
```

Une sortie sera renvoyée dans la CLI. Voici un exemple:

```
{
    "resolver": {
        "kind": "UNIT",
        "dataSourceName": "TodoTable",
        "requestMappingTemplate": "{ version : 2017-02-28, operation : Scan, }",
        "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Query/resolvers/getTodos",
        "typeName": "Query",
        "fieldName": "getTodos",
        "responseMappingTemplate": "$util.toJson($ctx.result.items)"
```

```
    }
}
```

2. Pour modifier les champs et/ou les modèles de mappage d'un résolveur, exécutez la `update-resolver`commande.

   À l'exception du `api-id` paramètre, les paramètres utilisés dans la `create-resolver` commande seront remplacés par les nouvelles valeurs de la `update-resolver` commande.

Ajouter un résolveur pour les mutations

L'étape suivante consiste à créer un résolveur adapté à votre `Mutation` type.

Console

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
   a. Dans le tableau de bord des API, choisissez votre API GraphQL.
   b. Dans la barre latérale, choisissez Schema.
2. Sous le type de mutation, choisissez Attacher à côté du `addTodo` champ.
3. Sur la page Create Resolver, choisissez la source de données que vous avez créée dans le guide Joindre une source de données.
4. Dans la fenêtre Configurer les modèles de mappage, vous devez modifier le modèle de demande car il s'agit d'une mutation dans laquelle vous ajoutez un nouvel élément à DynamoDB. Utilisez le modèle de mappage de demande suivant :

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

5. AWS AppSync convertit automatiquement les arguments définis dans le `addTodo` champ à partir de votre schéma GraphQL en opérations DynamoDB. L'exemple précédent stocke les enregistrements dans DynamoDB à l'aide d'une clé `id` de, qui est transmise par l'argument de mutation sous la forme. `$ctx.args.id` Tous les autres champs

que vous traversez sont automatiquement mappés aux attributs DynamoDB avec. `$util.dynamodb.toMapValuesJson($ctx.args)`

Pour ce résolveur, utilisez ensuite le modèle de mappage de réponse suivant :

```
$util.toJson($ctx.result)
```

AWS AppSync prend également en charge les flux de travail de test et de débogage pour l'édition de résolveurs. Vous pouvez utiliser un objet `context` simulé pour afficher la valeur transformée du modèle avant l'appel. Le cas échéant, vous pouvez afficher l'intégralité de l'exécution de la requête sur une source de données de façon interactive lorsque vous exécutez une requête. Pour plus d'informations, consultez les sections Tester et déboguer les résolveurs et Surveillance et journalisation.

6. Choisissez Save Resolver.

API

Vous pouvez également le faire avec les API en utilisant les commandes de la section Créez votre premier résolveur et les détails des paramètres de cette section.

CLI

Vous pouvez également le faire dans la CLI en utilisant les commandes de la section Créez votre premier résolveur et les détails des paramètres de cette section.

À ce stade, si vous n'utilisez pas les résolveurs avancés, vous pouvez commencer à utiliser votre API GraphQL comme indiqué dans Utilisation de votre API.

Résolveurs avancés

Si vous suivez la section Avancé et que vous créez un exemple de schéma dans Conception de votre schéma pour effectuer un scan paginé, utilisez plutôt le modèle de demande suivant pour le `getTodos` champ :

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "limit": $util.defaultIfNull(${ctx.args.limit}, 20),
    "nextToken": $util.toJson($util.defaultIfNullOrBlank($ctx.args.nextToken, null))
```

```
}
```

Pour ce scénario de pagination, le mappage de réponse est plus qu'une simple transmission, car il doit contenir à la fois le curseur (afin que le client sache quelle est la prochaine page à démarrer) et le jeu de résultats. Le modèle de mappage est ainsi :

```
{
    "todos": $util.toJson($context.result.items),
    "nextToken": $util.toJson($context.result.nextToken)
}
```

Les champs du précédent modèle de mappage de réponse doivent correspondre aux champs définis dans votre type `TodoConnection`.

Dans le cas de relations où vous avez une `Comments` table et que vous résolvez le champ de commentaires sur le `Todo` type (qui renvoie un type de`[Comment]`), vous pouvez utiliser un modèle de mappage qui exécute une requête sur la deuxième table. Pour ce faire, vous devez déjà avoir créé une source de données pour la `Comments` table, comme indiqué dans la section Joindre une source de données.

> **Note**
>
> Nous utilisons une opération de requête sur une deuxième table à des fins d'illustration uniquement. Vous pouvez utiliser une autre opération sur DynamoDB à la place. En outre, vous pouvez extraire les données d'une autre source de données, telle qu'AWS LambdaAmazon OpenSearch Service, car la relation est contrôlée par votre schéma GraphQL.

Console

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
   a. Dans le tableau de bord des API, choisissez votre API GraphQL.
   b. Dans la barre latérale, choisissez Schema.
2. Sous le type Todo, choisissez Joindre à côté du `comments` champ.
3. Sur la page Créer un résolveur, choisissez la source de données de votre table de commentaires. Le nom par défaut du tableau des commentaires dans les guides de

démarrage rapide estAppSyncCommentTable, mais il peut varier en fonction du nom que vous lui avez donné.

4. Ajoutez l'extrait suivant à votre modèle de mappage de demandes :

```
{
    "version": "2017-02-28",
    "operation": "Query",
    "index": "todoid-index",
    "query": {
        "expression": "todoid = :todoid",
        "expressionValues": {
            ":todoid": {
                "S": $util.toJson($context.source.id)
            }
        }
    }
}
```

5. Le `context.source` fait référence à l'objet parent du champ actif en cours de résolution. Dans cet exemple, `source.id` fait référence à l'objet `Todo` individuel, qui est ensuite utilisé pour l'expression de la requête.

   Vous pouvez utiliser le modèle de mappage de réponse de transmission, comme suit :

```
$util.toJson($ctx.result.items)
```

6. Choisissez Save Resolver.
7. Enfin, revenez sur la page Schéma de la console, attachez un résolveur au `addComment` champ et spécifiez la source de données de la `Comments` table. Dans ce cas, le modèle de mappage de requête est un simple `PutItem` avec le `todoid` spécifique qui est commenté en tant qu'argument, mais vous utilisez l'utilitaire `$utils.autoId()` pour créer une clé de tri unique pour le commentaire de la façon suivante :

```
{
    "version": "2017-02-28",
    "operation": "PutItem",
    "key": {
        "todoid": { "S": $util.toJson($context.arguments.todoid) },
        "commentid": { "S": "$util.autoId()" }
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
```

```
}
```

Utilisez un modèle de réponse de transmission comme suit :

```
$util.toJson($ctx.result)
```

API

Vous pouvez également le faire avec les API en utilisant les commandes de la section Créez votre premier résolveur et les détails des paramètres de cette section.

CLI

Vous pouvez également le faire dans la CLI en utilisant les commandes de la section Créez votre premier résolveur et les détails des paramètres de cette section.

Résolveurs Lambda directs (VTL)

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Avec les résolveurs Lambda directs, vous pouvez contourner l'utilisation de modèles de mappage VTL lorsque vous utilisez des sources de données. AWS Lambda AWS AppSync peut fournir une charge utile par défaut à votre fonction Lambda ainsi qu'une traduction par défaut de la réponse d'une fonction Lambda en un type GraphQL. Vous pouvez choisir de fournir un modèle de demande, un modèle de réponse ou aucun des deux et AWS AppSync vous les traiterez en conséquence.

Pour en savoir plus sur la charge utile des demandes par défaut et la traduction des réponses AWS AppSync fournies, consultez la référence du résolveur Direct Lambda. Pour plus d'informations sur la configuration d'une source de AWS Lambda données et la configuration d'une politique de confiance IAM, voir Joindre une source de données.

Configurer des résolveurs Lambda directs

Les sections suivantes vous montrent comment associer des sources de données Lambda et ajouter des résolveurs Lambda à vos champs.

Ajouter une source de données Lambda

Avant de pouvoir activer les résolveurs Lambda directs, vous devez ajouter une source de données Lambda.

Console

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
   a. Dans le tableau de bord des API, choisissez votre API GraphQL.
   b. Dans la barre latérale, sélectionnez Sources de données.
2. Choisissez Create data source.
   a. Dans Nom de la source de données, entrez un nom pour votre source de données, tel que**`myFunction`**.
   b. Pour Type de source de données, sélectionnez AWS Lambdafonction.
   c. Pour Région, choisissez la région appropriée.
   d. Pour Function ARN, choisissez la fonction Lambda dans la liste déroulante. Vous pouvez rechercher le nom de la fonction ou saisir manuellement l'ARN de la fonction que vous souhaitez utiliser.
   e. Créez un nouveau rôle IAM (recommandé) ou choisissez un rôle existant disposant de l'autorisation `lambda:invokeFunction` IAM. Les rôles existants nécessitent une politique de confiance, comme expliqué dans la section Joindre une source de données.

      Voici un exemple de politique IAM qui dispose des autorisations requises pour effectuer des opérations sur la ressource :

```
{
     "Version": "2012-10-17",
     "Statement": [
         {
             "Effect": "Allow",
             "Action": [ "lambda:invokeFunction" ],
             "Resource": [
                 "arn:aws:lambda:us-
west-2:123456789012:function:myFunction",
                 "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:*"
             ]
         }
```

```
    ]
}
```

3. Cliquez sur le bouton Créer.

CLI

1. Créez un objet de source de données en exécutant la `create-data-source`commande.

   Vous devez saisir 4 paramètres pour cette commande particulière :

   1. Celui `api-id` de votre API.
   2. Le `name` de votre source de données. Dans l'exemple de console, il s'agit du nom de la source de données.
   3. La source `type` de données. Dans l'exemple de console, il s'agit d'AWS Lambdaune fonction.
   4. Le`lambda-config`, qui est l'ARN de la fonction dans l'exemple de console.

   > **Note**
   >
   > Il existe d'autres paramètres tels `Region` que ceux qui doivent être configurés, mais ils seront généralement définis par défaut sur les valeurs de configuration de votre CLI.

   Un exemple de commande peut ressembler à ceci :

```
aws appsync create-data-source --api-id abcdefghijklmnopqrstuvwxyz
 --name myFunction --type AWS_LAMBDA --lambda-config
 lambdaFunctionArn=arn:aws:lambda:us-west-2:102847592837:function:appsync-
lambda-example
```

   Une sortie sera renvoyée dans la CLI. Voici un exemple:

```
{
    "dataSource": {
        "dataSourceArn": "arn:aws:appsync:us-west-2:102847592837:apis/
abcdefghijklmnopqrstuvwxyz/datasources/myFunction",
```

```
        "type": "AWS_LAMBDA",
        "name": "myFunction",
        "lambdaConfig": {
            "lambdaFunctionArn": "arn:aws:lambda:us-
west-2:102847592837:function:appsync-lambda-example"
        }
    }
}
```

2. Pour modifier les attributs d'une source de données, exécutez la `update-data-source`commande.

   À l'exception du `api-id` paramètre, les paramètres utilisés dans la `create-data-source` commande seront remplacés par les nouvelles valeurs de la `update-data-source` commande.

Activer les résolveurs Lambda directs

Après avoir créé une source de données Lambda et configuré le rôle IAM approprié pour permettre d'invoquer la fonction, vous pouvez la lier AWS AppSync à un résolveur ou à une fonction de pipeline.

Console

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.

   a. Dans le tableau de bord des API, choisissez votre API GraphQL.

   b. Dans la barre latérale, choisissez Schema.

2. Dans la fenêtre Resolvers, choisissez un champ ou une opération, puis cliquez sur le bouton Joindre.
3. Sur la page Créer un nouveau résolveur, choisissez la fonction Lambda dans la liste déroulante.
4. Afin de tirer parti des résolveurs Lambda directs, vérifiez que les modèles de mappage de demandes et de réponses sont désactivés dans la section Configurer les modèles de mappage.
5. Cliquez sur le bouton Enregistrer le résolveur.

CLI

- Créez un résolveur en exécutant la `create-resolver`commande.

  Vous devez saisir 6 paramètres pour cette commande particulière :

  1. Celui `api-id` de votre API.
  2. Le `type-name` type indiqué dans votre schéma.
  3. Le champ `field-name` de votre schéma.
  4. Le`data-source-name`, ou le nom de votre fonction Lambda.
  5. Le`request-mapping-template`, qui est le corps de la demande. Dans l'exemple de console, cela a été désactivé :

     ```
     " "
     ```

  6. Le`response-mapping-template`, qui est le corps de la réponse. Dans l'exemple de console, cela a également été désactivé :

     ```
     " "
     ```

  Un exemple de commande peut ressembler à ceci :

  ```
  aws appsync create-resolver --api-id abcdefghijklmnopqrstuvwxyz --type-name
   Subscription --field-name onCreateTodo --data-source-name LambdaTest --request-
  mapping-template " " --response-mapping-template " "
  ```

  Une sortie sera renvoyée dans la CLI. Voici un exemple:

  ```
  {
      "resolver": {
          "resolverArn": "arn:aws:appsync:us-west-2:102847592837:apis/
  abcdefghijklmnopqrstuvwxyz/types/Subscription/resolvers/onCreateTodo",
          "typeName": "Subscription",
          "kind": "UNIT",
          "fieldName": "onCreateTodo",
          "dataSourceName": "LambdaTest"
      }
  }
  ```

Lorsque vous désactivez vos modèles de mappage, plusieurs comportements supplémentaires se produisent dans AWS AppSync :

- En désactivant un modèle de mappage, vous indiquez AWS AppSync que vous acceptez les traductions de données par défaut spécifiées dans la référence du résolveur Direct Lambda.
- En désactivant le modèle de mappage des demandes, votre source de données Lambda recevra une charge utile composée de l'ensemble de l'objet Context.
- En désactivant le modèle de mappage de réponse, le résultat de votre appel Lambda sera traduit en fonction de la version du modèle de mappage de demandes ou si le modèle de mappage de demandes est également désactivé.

Résolveurs de test et de débogage (VTL)

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

AWS AppSync exécute des résolveurs sur un champ GraphQL par rapport à une source de données. Comme décrit dans la section Présentation du modèle de mappage Resolver, les résolveurs communiquent avec les sources de données à l'aide d'un langage de modélisation. Cela vous permet de personnaliser le comportement et d'appliquer une logique et des conditions avant et après la communication avec la source de données. Pour un guide de programmation de type didacticiel d'introduction à l'écriture de résolveurs, consultez le guide de programmation du modèle de mappage Resolver.

Pour aider les développeurs à écrire, tester et déboguer ces résolveurs, la AWS AppSync console fournit également des outils permettant de créer une requête et une réponse GraphQL avec des données fictives jusqu'au résolveur de champs individuel. En outre, vous pouvez effectuer des requêtes, des mutations et des abonnements dans la AWS AppSync console et consulter un flux de journal détaillé de l'intégralité CloudWatch de la demande provenant d'Amazon. Cela inclut les résultats d'une source de données.

Tester avec des données fictives

Lorsqu'un résolveur GraphQL est invoqué, il contient un `context` objet contenant des informations sur la demande. Ces informations incluent les arguments d'un client, les informations d'identité et les

données du champ GraphQL parent. Il contient également les résultats de la source de données, qui peuvent être utilisés dans le modèle de réponse. Vous trouverez plus de détails sur cette structure et les utilitaires d'annotations disponibles à utiliser lors de la programmation dans le document Référence du contexte du modèle de mappage des résolveurs.

Lorsque vous écrivez ou modifiez un résolveur, vous pouvez transmettre un objet de contexte fictif ou de test dans l'éditeur de console. Cela vous permet de voir comment les deux modèles de la demande et de la réponse évaluent, sans réellement s'exécuter par rapport à une source de données. Par exemple, vous pouvez transmettre un argument de test `firstname: Shaggy` et voir comment il est analysé lors de l'utilisation de `$ctx.args.firstname` dans votre modèle de code. Vous pouvez également tester l'évaluation de n'importe quelle annotation d'utilitaire telle que `$util.autoId()` ou `util.time.nowISO8601()`.

Tester les résolveurs

Cet exemple utilisera la AWS AppSync console pour tester les résolveurs.

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
   a. Dans le tableau de bord des API, choisissez votre API GraphQL.
   b. Dans la barre latérale, choisissez Schema.
2. Si ce n'est pas déjà fait, sous le type et à côté du champ, choisissez Joindre pour ajouter votre résolveur.

   Pour plus d'informations sur la création d'un résolveur complet, voir Configuration des résolveurs.

   Sinon, sélectionnez le résolveur qui se trouve déjà dans le champ.
3. En haut de la page Modifier le résolveur, choisissez Sélectionner le contexte de test, puis Créer un nouveau contexte.
4. Sélectionnez un exemple d'objet de contexte ou renseignez le JSON manuellement dans la fenêtre de contexte d'exécution ci-dessous.
5. Entrez un nom de contexte de texte.
6. Choisissez le bouton Enregistrer.
7. En haut de la page Modifier le résolveur, sélectionnez Exécuter le test.

Pour un exemple plus pratique, supposons que vous disposiez d'une application stockant un type GraphQL Dog qui utilise la génération automatique d'identifiants pour les objets et les stocke dans

Amazon DynamoDB. Il se peut aussi que vous souhaitiez écrire certaines valeurs des arguments d'une mutation GraphQL et permettre uniquement à des utilisateurs spécifiques de voir une réponse. Le schéma pourrait s'apparenter à ce qui suit :

```
type Dog {
  breed: String
  color: String
}

type Mutation {
  addDog(firstname: String, age: Int): Dog
}
```

Lorsque vous ajoutez un résolveur pour la `addDog` mutation, vous pouvez renseigner un objet de contexte comme dans l'exemple suivant. Le suivant possède les arguments `name` et `age` du client, ainsi qu'un `username` renseigné dans l'objet `identity` :

```
{
    "arguments" : {
        "firstname": "Shaggy",
        "age": 4
    },
    "source" : {},
    "result" : {
        "breed" : "Miniature Schnauzer",
        "color" : "black_grey"
    },
    "identity": {
        "sub" : "uuid",
        "issuer" : " https://cognito-idp.{region}.amazonaws.com/{userPoolId}",
        "username" : "Nadia",
        "claims" : { },
        "sourceIp" :[  "x.x.x.x" ],
        "defaultAuthStrategy" : "ALLOW"
    }
}
```

Vous pouvez effectuer le test en utilisant les modèles de mappage de demande et de réponse suivants :

Modèle de demande

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : { "S" : "$util.autoId()" }
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

Modèle de réponse

```
#if ($context.identity.username == "Nadia")
  $util.toJson($ctx.result)
#else
  $util.unauthorized()
#end
```

Le modèle évalué possède les données de votre objet de contexte du test et la valeur générée à partir de `$util.autoId()`. De plus, si vous deviez modifier `username` avec une valeur autre que `Nadia`, les résultats ne seront pas renvoyés, car le contrôle d'autorisation échouerait. Pour plus d'informations sur le contrôle d'accès détaillé, consultez la section Cas d'utilisation des autorisations.

Tester des modèles de mappage avec AWS AppSync les API

Vous pouvez utiliser la commande `EvaluateMappingTemplate` API pour tester à distance vos modèles de mappage avec des données simulées. Pour commencer à utiliser la commande, assurez-vous d'avoir ajouté l'`appsync:evaluateMappingTemplate`autorisation à votre politique. Par exemple :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateMappingTemplate",
            "Resource": "arn:aws:appsync:<region>:<account>:*"
        }
    ]
}
```

Vous pouvez tirer parti de la commande à l'aide des AWSSDK AWS CLIou. Prenons par exemple le Dog schéma et ses modèles de mappage de demande/réponse de la section précédente. À l'aide de la CLI de votre station locale, enregistrez le modèle de demande dans un fichier nommérequest.vtl, puis enregistrez l'contextobjet dans un fichier nommécontext.json. Depuis votre shell, exécutez la commande suivante :

```
aws appsync evaluate-mapping-template --template file://request.vtl --context file://
context.json
```

La commande renvoie la réponse suivante :

```
{
  "evaluationResult": "{\n    \"version\" : \"2017-02-28\",\n
 \"operation\" : \"PutItem\",\n    \"key\" : {\n        \"id\" : { \"S\" :
 \"afcb4c85-49f8-40de-8f2b-248949176456\" }\n    },\n    \"attributeValues\" :
 {\"firstname\":{\"S\":\"Shaggy\"},\"age\":{\"N\":4}}\n}\n"
}
```

evaluationResultIl contient les résultats du test de votre modèle fourni avec le modèle fournicontext. Vous pouvez également tester vos modèles à l'aide des AWS SDK. Voici un exemple d'utilisation du AWS SDK pour la JavaScript version 2 :

```
const AWS = require('aws-sdk')
const client = new AWS.AppSync({ region: 'us-east-2' })

const template = fs.readFileSync('./request.vtl', 'utf8')
const context = fs.readFileSync('./context.json', 'utf8')

client
  .evaluateMappingTemplate({ template, context })
  .promise()
  .then((data) => console.log(data))
```

À l'aide du SDK, vous pouvez facilement intégrer les tests de votre suite de tests préférée pour valider le comportement de votre modèle. Nous vous recommandons de créer des tests à l'aide du Jest Testing Framework, mais toutes les suites de tests fonctionnent. L'extrait suivant montre une exécution de validation hypothétique. Notez que nous nous attendons à ce que la réponse d'évaluation soit un JSON valide. Nous utilisons donc JSON.parse pour récupérer le JSON à partir de la réponse sous forme de chaîne :

```
const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })

test('request correctly calls DynamoDB', async () => {
  const template = fs.readFileSync('./request.vtl', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateMappingTemplate({ template,
 context }).promise()
  const result = JSON.parse(response.evaluationResult)

  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})
```

Cela donne le résultat suivant :

```
Ran all test suites.
> jest

PASS ./index.test.js
# request correctly calls DynamoDB (543 ms)

Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 1.511 s, estimated 2 s
```

Débogage d'une requête en direct

Rien ne remplace un end-to-end test et une journalisation pour déboguer une application de production. AWS AppSync vous permet de consigner les erreurs et de fournir les détails complets des demandes à l'aide d'Amazon CloudWatch. De plus, vous pouvez utiliser la console AWS AppSync pour tester les requêtes, les mutations et les abonnements GraphQL, ainsi que les données de journalisation des flux en direct de chaque demande dans l'éditeur de requête afin de déboguer en temps réel. Pour les abonnements, les journaux affichent les informations de temps de connexion.

Pour ce faire, vous devez activer Amazon CloudWatch Logs à l'avance, comme décrit dans Surveillance et journalisation. Ensuite, dans la console AWS AppSync , choisissez l'onglet Requêtes

et saisissez une requête GraphQL valable. Dans la section inférieure droite, cliquez et faites glisser la fenêtre Logs pour ouvrir la vue des logs. À l'aide de la flèche « Play » en haut de la page, exécutez votre requête GraphQL. Dans quelques instants, vos journaux complets de demande et de réponse pour l'opération sont diffusés vers cette section et vous pouvez les voir dans la console.

Résolveurs de pipeline (VTL)

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

AWS AppSync exécute des résolveurs sur un champ GraphQL. Dans certains cas, les applications nécessitent l'exécution de plusieurs opérations pour résoudre un seul champ GraphQL. Avec les résolveurs de pipeline, les développeurs peuvent désormais composer des opérations appelées fonctions et les exécuter en séquence. Les résolveurs de pipeline sont utiles pour les applications qui, par exemple, doivent effectuer un contrôle d'autorisation avant d'extraire des données pour un champ.

Un résolveur de pipeline est composé d'un modèle de mappage Avant, d'un modèle de mappage Après et d'une liste de Fonctions. Chaque fonction possède un modèle de mappage de requêtes et de réponses qu'elle exécute par rapport à une source de données. Comme un résolveur de pipeline délègue l'exécution à une liste de fonctions, il n'est lié à aucune source de données. Les résolveurs d'unités et les fonctions sont des primitives qui exécutent des opérations sur des sources de données. Consultez la présentation du modèle de mappage Resolver pour plus d'informations.

Création d'un résolveur de pipeline

Dans la console AWS AppSync , accédez à la page Schéma.

Enregistrez le schéma suivant :

```
schema {
    query: Query
    mutation: Mutation
}

type Mutation {
```

```
    signUp(input: Signup): User
}

type Query {
    getUser(id: ID!): User
}

input Signup {
    username: String!
    email: String!
}

type User {
    id: ID!
    username: String
    email: AWSEmail
}
```

Nous allons raccorder un résolveur de pipeline au champ signUp sur le type Mutation. Dans le champ Type de mutation sur le côté droit, choisissez Attacher à côté du champ de `signUp` mutation. Sur la page de création d'un résolveur, cliquez sur Actions, puis sur Mettre à jour le moteur d'exécution. Choisissez`Pipeline Resolver`, puis choisissez`VTL`, puis choisissez Mettre à jour. La page doit désormais afficher trois sections : une zone de texte avant le mappage du modèle, une section Fonctions et une zone de texte après le mappage du modèle.

Notre résolveur de pipeline inscrit un utilisateur en validant tout d'abord l'adresse e-mail saisie, puis en enregistrant l'utilisateur dans le système. Nous allons encapsuler la validation d'e-mail dans une fonction validateEmail et l'enregistrement de l'utilisateur dans une fonction saveUser. La fonction validateEmail s'exécute d'abord, puis si l'e-mail est valide, alors la fonction saveUser s'exécute.

Le flux d'exécution sera comme suit :

1. Modèle de mappage de demande de résolveur Mutation.signUp
2. Fonction validateEmail
3. Fonction saveUser
4. Modèle de mappage de réponse de résolveur Mutation.signUp

Comme nous réutiliserons probablement la fonction ValidateEmail dans d'autres résolveurs de notre API, nous voulons éviter d'`$ctx.argsy` accéder car ceux-ci changeront d'un champ GraphQL à

l'autre. Au lieu de cela, nous pouvons utiliser le `$ctx.stash` pour stocker l'attribut e-mail depuis l'argument de champ de saisie `signUp(input: Signup)`.

Modèle de mappage BEFORE :

```
## store email input field into a generic email key
$util.qr($ctx.stash.put("email", $ctx.args.input.email))
{}
```

La console fournit un modèle de mappage AFTER passthrough par défaut que nous utiliserons :

```
$util.toJson($ctx.result)
```

Choisissez Créer ou Enregistrer pour mettre à jour le résolveur.

Créer une fonction

Sur la page du résolveur de pipeline, dans la section Fonctions, cliquez sur Ajouter une fonction, puis sur Créer une nouvelle fonction. Il est également possible de créer des fonctions sans passer par la page du résolveur ; pour cela, dans la AWS AppSync console, rendez-vous sur la page Fonctions. Choisissez le bouton Créer une fonction. Créons une fonction permettant de vérifier si un e-mail est valide et provient d'un domaine spécifique. Si l'e-mail n'est pas valide, la fonction renvoie une erreur. Dans le cas contraire, elle transmet les données saisies.

Sur la page des nouvelles fonctions, choisissez Actions, puis Mettre à jour le moteur d'exécution. ChoisissezVTL, puis Mettre à jour. Assurez-vous d'avoir créé une source de données de type NONE. Choisissez cette source de données dans la liste des noms des sources de données. Pour le nom de la fonction, entrez`validateEmail`. Dans la zone du code de fonction, remplacez tout par cet extrait de code :

```
#set($valid = $util.matches("^[a-zA-Z0-9_.+-]+@(?:(?:[a-zA-Z0-9-]+\.)?[a-zA-Z]+\.)?
(myvaliddomain)\.com", $ctx.stash.email))
#if (!$valid)
    $util.error("$ctx.stash.email is not a valid email.")
#end
{
    "payload": { "email": $util.toJson(${ctx.stash.email}) }
}
```

Collez ceci dans le modèle de mappage des réponses :

```
$util.toJson($ctx.result)
```

Passez en revue vos modifications, puis choisissez Créer. Nous venons de créer notre fonction validateEmail. Répétez ces étapes pour créer la fonction SaveUser avec les modèles de mappage de demandes et de réponses suivants (par souci de simplicité, nous utilisons une source de données NONE et prétendons que l'utilisateur a été enregistré dans le système après l'exécution de la fonction.) :

Modèle de mappage de demande :

```
## $ctx.prev.result contains the signup input values. We could have also
## used $ctx.args.input.
{
    "payload": $util.toJson($ctx.prev.result)
}
```

Modèle de mappage de réponse :

```
## an id is required so let's add a unique random identifier to the output
$util.qr($ctx.result.put("id", $util.autoId()))
$util.toJson($ctx.result)
```

Nous venons de créer notre fonction SaveUser.

Ajout d'une fonction à un résolveur de pipeline

Nos fonctions auraient dû être ajoutées automatiquement au résolveur de pipeline que nous venons de créer. Si ce n'était pas le cas, ou si vous avez créé les fonctions via la page Fonctions, vous pouvez cliquer sur Ajouter une fonction sur la page du résolveur pour les joindre. Ajoutez les fonctions ValidateEmail et SaveUser au résolveur. La fonction validateEmail doit être placée avant la fonction saveUser. Au fur et à mesure que vous ajoutez des fonctions, vous pouvez utiliser les options de déplacement vers le haut et de déplacement vers le bas pour réorganiser l'ordre d'exécution de vos fonctions. Passez en revue vos modifications, puis choisissez Enregistrer.

Exécution d'une requête

Dans la console AWS AppSync , accédez à la page Queries (Requêtes). Dans l'explorateur, assurez-vous d'utiliser votre mutation. Si ce n'est pas le cas, choisissez `Mutation` dans la liste déroulante, puis choisissez+. Entrez la requête suivante :

```
mutation {
  signUp(input: {
    email: "nadia@myvaliddomain.com"
    username: "nadia"
  }) {
    id
    email
  }
}
```

Cela devrait renvoyer quelque chose comme :

```
{
  "data": {
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "email": "nadia@myvaliddomain.com"
    }
  }
}
```

Nous avons inscrit notre utilisateur et validé l'e-mail saisi à l'aide d'un résolveur de pipeline. Pour suivre un didacticiel plus complet sur les résolveurs de pipeline, vous pouvez consulter le Didacticiel : Résolveurs de Pipeline

Étape 4 : Utilisation d'une API : exemple de CDK

> **Tip**
>
> Avant d'utiliser le CDK, nous vous recommandons de consulter les CDKdocumentation officielleavecAWS AppSyncc'estRéférence CDK.
> Nous vous recommandons également de veiller à ce que votreAWSCLIetNPMles installations fonctionnent sur votre système.

Dans cette section, nous allons créer une application CDK simple qui peut ajouter et récupérer des éléments à partir d'une table DynamoDB. Il s'agit d'un exemple de démarrage rapide utilisant une partie du code duConception de votre schéma,Joindre une source de données, etConfiguration des résolveurs (JavaScript)sections.

Configuration d'un projet CDK

> **Warning**
>
> Ces étapes peuvent ne pas être totalement précises en fonction de votre environnement. Nous supposons que les utilitaires nécessaires sont installés sur votre système, un moyen d'interfacer avecAWSservices et configurations appropriées en place.

La première étape consiste à installerAWSCDK. Dans votre CLI, vous pouvez entrer la commande suivante :

```
npm install -g aws-cdk
```

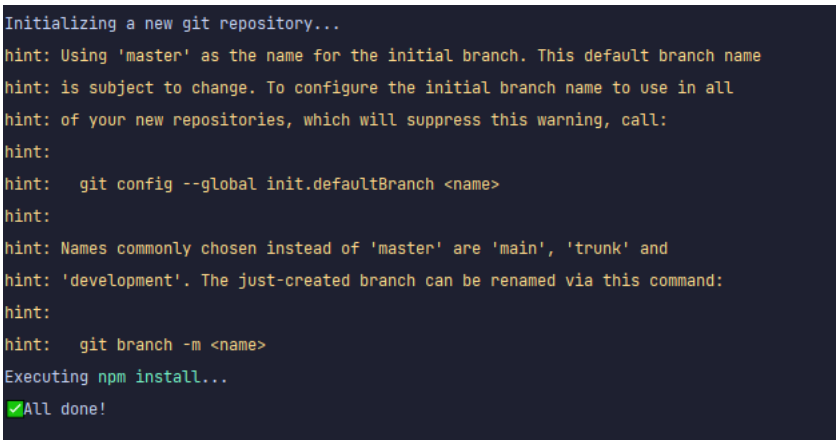
Ensuite, vous devez créer un répertoire de projet, puis y accéder. Voici un exemple d'ensemble de commandes permettant de créer un répertoire et d'y accéder :

```
mkdir example-cdk-app
cd example-cdk-app
```

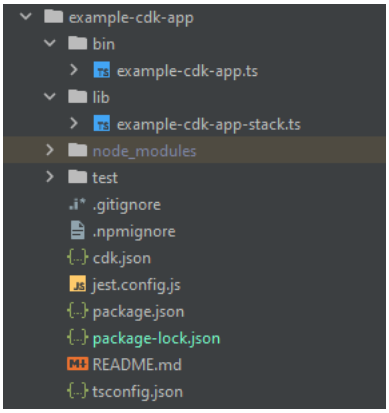
Ensuite, vous devez créer une application. Notre service utilise principalementTypeScript. Dans le répertoire de votre projet, entrez la commande suivante :

```
cdk init app --language typescript
```

Dans ce cas, une application CDK ainsi que ses fichiers d'initialisation seront installés :



La structure de votre projet peut ressembler à ceci :



Vous remarquerez que nous avons plusieurs annuaires importants :

- `bin`: Le fichier bin initial créera l'application. Nous n'aborderons pas cela dans ce guide.
- `lib`: Le répertoire lib contient vos fichiers de pile. Vous pouvez considérer les fichiers de pile comme des unités d'exécution individuelles. Les constructions se trouveront dans nos fichiers de pile. Il s'agit essentiellement de ressources pour un service qui sera créé dansAWS CloudFormationlorsque l'application est déployée. C'est là que se déroulera la majeure partie de notre codage.
- `node_modules`: Ce répertoire est créé par NPM et contient toutes les dépendances des packages que vous avez installées à l'aide dunpmcommande.

Notre fichier de pile initial peut contenir quelque chose comme ceci :

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
// import * as sqs from 'aws-cdk-lib/aws-sqs';

export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // The code that defines your stack goes here

    // example resource
    // const queue = new sqs.Queue(this, 'ExampleCdkAppQueue', {
    //   visibilityTimeout: cdk.Duration.seconds(300)
    // });
  }
}
```

Il s'agit du code standard pour créer une pile dans notre application. Dans cet exemple, la majeure partie de notre code entrera dans le champ d'application de cette classe.

Pour vérifier que votre fichier de pile se trouve dans l'application, dans le répertoire de votre application, exécutez la commande suivante dans le terminal :

```
cdk ls
```

Une liste de vos piles devrait apparaître. Si ce n'est pas le cas, vous devrez peut-être recommencer les étapes ou consulter la documentation officielle pour obtenir de l'aide.

Si vous souhaitez créer vos modifications de code avant le déploiement, vous pouvez toujours exécuter la commande suivante dans le terminal :

```
npm run build
```

Et pour voir les modifications avant le déploiement :

```
cdk diff
```

Avant d'ajouter notre code au fichier de pile, nous allons effectuer un bootstrap. Le bootstrapping nous permet de fournir des ressources pour le CDK avant le déploiement de l'application. Plus d'informations sur ce processus peuvent être trouvéesici. Pour créer un bootstrap, la commande est la suivante :

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

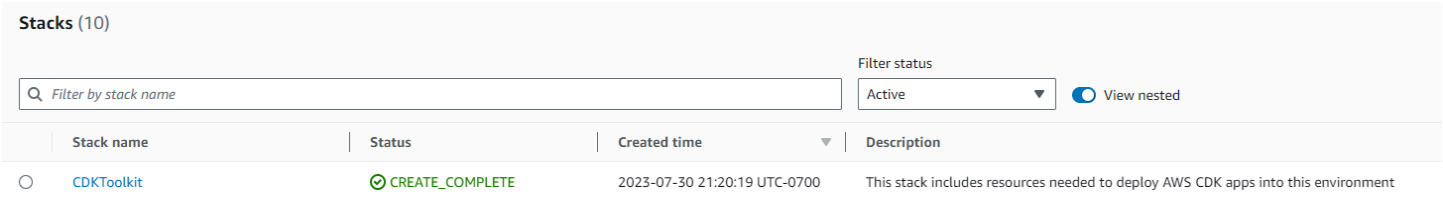
> **Tip**
>
> Cette étape nécessite plusieurs autorisations IAM sur votre compte. Votre bootstrap sera refusé si vous ne les avez pas. Dans ce cas, vous devrez peut-être supprimer les ressources incomplètes causées par le bootstrap, telles que le compartiment S3 qu'il génère.

Bootstrap lancera plusieurs ressources. Le message final ressemblera à ceci :

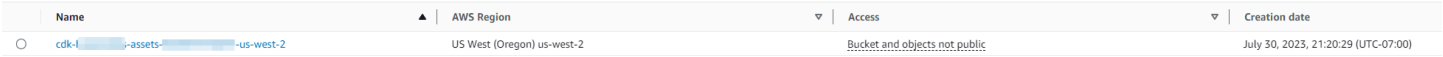
```
⏳ Bootstrapping environment
Trusted accounts for deployment: (none)
Trusted accounts for lookup: (none)
Using default execution policy of 'arn:aws:iam::aws:policy/AdministratorAccess'. Pass '--cloudformation-execution-policies' to customize.
CDKToolkit: creating CloudFormation changeset...
✅ Environment bootstrapped.
```

Cela se fait une fois par compte et par région, vous n'aurez donc pas à le faire souvent. Les principales ressources du bootstrap sontAWS CloudFormationstack et le compartiment Amazon S3.

Le compartiment Amazon S3 est utilisé pour stocker les fichiers et les rôles IAM qui accordent les autorisations nécessaires pour effectuer des déploiements. Les ressources requises sont définies dans unAWS CloudFormationpile, appelée pile bootstrap, qui est généralement nomméeCDKToolkit. Comme n'importe quelAWS CloudFormationpile, elle apparaît dans leAWS CloudFormationconsole une fois qu'elle a été déployée :



Il en va de même pour le bucket :



Pour importer les services dont nous avons besoin dans notre fichier de pile, nous pouvons utiliser la commande suivante :

```
npm install aws-cdk-lib # V2 command
```

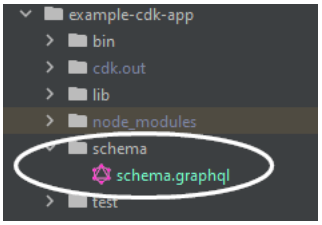
> **Tip**
>
> Si vous rencontrez des problèmes avec la V2, vous pouvez installer les bibliothèques individuelles à l'aide des commandes V1 :
>
> ```
> npm install @aws-cdk/aws-appsync @aws-cdk/aws-dynamodb
> ```
>
> Nous ne le recommandons pas car la V1 est obsolète.

Implémentation d'un projet CDK - Schéma

Nous pouvons maintenant commencer à implémenter notre code. Nous devons d'abord créer notre schéma. Vous pouvez simplement créer un `.graphql`fichier dans votre application :

```
mkdir schema
touch schema.graphql
```

Dans notre exemple, nous avons inclus un répertoire de premier niveau appelé`schema`contenant notre`schema.graphql`:



Dans notre schéma, incluons un exemple simple :

```
input CreatePostInput {
    title: String
    content: String
}

type Post {
    id: ID!
    title: String
    content: String
}

type Mutation {
    createPost(input: CreatePostInput!): Post
}

type Query {
    getPost: [Post]
}
```

De retour dans notre fichier de pile, nous devons nous assurer que les directives d'importation suivantes sont définies :

```
import * as cdk from 'aws-cdk-lib';
```

```
import * as appsync from 'aws-cdk-lib/aws-appsync';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
import { Construct } from 'constructs';
```

Dans la classe, nous ajouterons du code pour créer notre API GraphQL et la connecter à notre`schema.graphql`fichier :

```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // makes a GraphQL API
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });
  }
}
```

Nous ajouterons également du code pour imprimer l'URL, la clé d'API et la région de GraphQL :

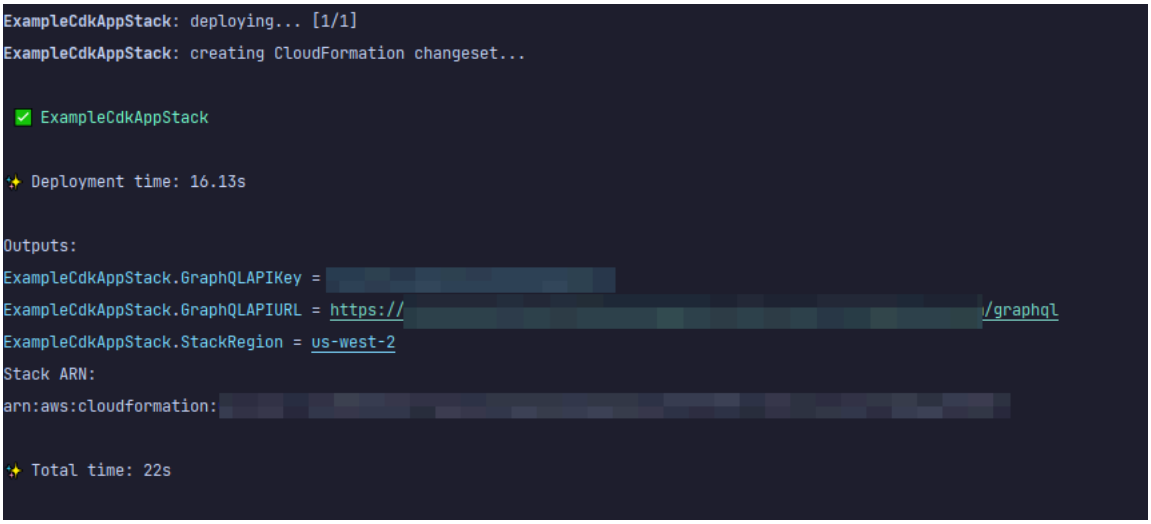
```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Makes a GraphQL API construct
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });

    // Prints out URL
    new cdk.CfnOutput(this, "GraphQLAPIURL", {
      value: api.graphqlUrl
    });

    // Prints out the AppSync GraphQL API key to the terminal
    new cdk.CfnOutput(this, "GraphQLAPIKey", {
      value: api.apiKey || ''
    });

    // Prints out the stack region to the terminal
    new cdk.CfnOutput(this, "Stack Region", {
```

```
      value: this.region
    });
  }
}
```

À ce stade, nous allons à nouveau utiliser Deploy notre application :

```
cdk deploy
```

Voici le résultat :



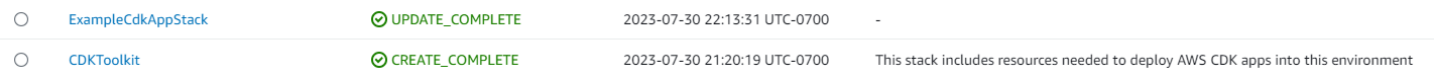
Il semble que notre exemple ait été un succès, mais vérifions-leAWS AppSyncconsole juste pour confirmer :

Il semblerait que notre API ait été créée. Nous allons maintenant vérifier le schéma attaché à l'API :

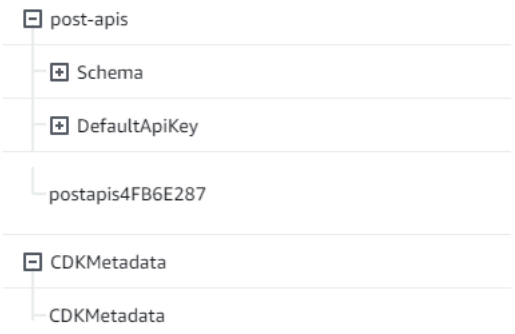
Schema

```
input CreatePostInput {
  title: String
  date: AWSDateTime
}

type Post {
  id: ID!
  title: String
  date: AWSDateTime
}

type Mutation {
  createPost(input: CreatePostInput!): Post
}

type Query {
  getPost: [Post]
}
```

Cela semble correspondre à notre code de schéma, donc c'est réussi. Une autre façon de le confirmer du point de vue des métadonnées est de consulter leAWS CloudFormationpile :



Lorsque nous déployons notre application CDK, elle passe parAWS CloudFormationpour faire tourner des ressources comme le bootstrap. Chaque pile de notre application correspond à une carte 1:1 avec unAWS CloudFormationpile. Si vous revenez au code de pile, le nom de la pile a été extrait du nom de la classe`ExampleCdkAppStack`. Vous pouvez voir les ressources qu'il a créées, qui correspondent également à nos conventions de dénomination, dans notre structure d'API GraphQL :



Implémentation d'un projet CDK - Source de données

Ensuite, nous devons ajouter notre source de données. Notre exemple utilisera une table DynamoDB. Dans la classe stack, nous allons ajouter du code pour créer une nouvelle table :

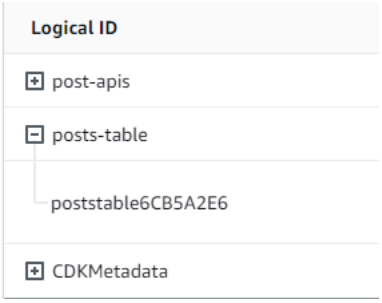
```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Makes a GraphQL API construct
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });

    //creates a DDB table
    const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
      partitionKey: {
        name: 'id',
        type: dynamodb.AttributeType.STRING,
      },
    });

    // Prints out URL
    new cdk.CfnOutput(this, "GraphQLAPIURL", {
      value: api.graphqlUrl
    });

    // Prints out the AppSync GraphQL API key to the terminal
    new cdk.CfnOutput(this, "GraphQLAPIKey", {
      value: api.apiKey || ''
    });

    // Prints out the stack region to the terminal
    new cdk.CfnOutput(this, "Stack Region", {
      value: this.region
    });
  }
}
```

À ce stade, déployons à nouveau :

```
cdk deploy
```

Nous devrions vérifier la console DynamoDB pour notre nouvelle table :

Le nom de notre pile est correct et le nom de la table correspond à notre code. Si nous vérifions notreAWS CloudFormationempilez à nouveau, nous allons maintenant voir la nouvelle table :



Implémentation d'un projet CDK - Resolver

Cet exemple utilisera deux résolveurs : l'un pour interroger la table et l'autre pour y ajouter des éléments. Puisque nous utilisons des résolveurs de pipeline, nous devrons déclarer deux résolveurs de pipeline avec une fonction dans chacun. Dans la requête, nous allons ajouter le code suivant :

```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Makes a GraphQL API construct
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });

    //creates a DDB table
    const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
      partitionKey: {
        name: 'id',
        type: dynamodb.AttributeType.STRING,
      },
    });

    // Creates a function for query
    const add_func = new appsync.AppsyncFunction(this, 'func-get-post', {
      name: 'get_posts_func_1',
      api,
      dataSource: api.addDynamoDbDataSource('table-for-posts', add_ddb_table),
      code: appsync.Code.fromInline(`
          export function request(ctx) {
          return { operation: 'Scan' };
```

```
        }

        export function response(ctx) {
        return ctx.result.items;
        }
`),
    runtime: appsync.FunctionRuntime.JS_1_0_0,
  });

  // Creates a function for mutation
  const add_func_2 = new appsync.AppsyncFunction(this, 'func-add-post', {
    name: 'add_posts_func_1',
    api,
    dataSource: api.addDynamoDbDataSource('table-for-posts-2', add_ddb_table),
    code: appsync.Code.fromInline(`
        export function request(ctx) {
          return {
          operation: 'PutItem',
          key: util.dynamodb.toMapValues({id: util.autoId()}),
          attributeValues: util.dynamodb.toMapValues(ctx.args.input),
          };
        }

        export function response(ctx) {
          return ctx.result;
        }
    `),
    runtime: appsync.FunctionRuntime.JS_1_0_0,
  });

  // Adds a pipeline resolver with the get function
  new appsync.Resolver(this, 'pipeline-resolver-get-posts', {
    api,
    typeName: 'Query',
    fieldName: 'getPost',
    code: appsync.Code.fromInline(`
        export function request(ctx) {
        return {};
        }

        export function response(ctx) {
        return ctx.prev.result;
        }
`),
```

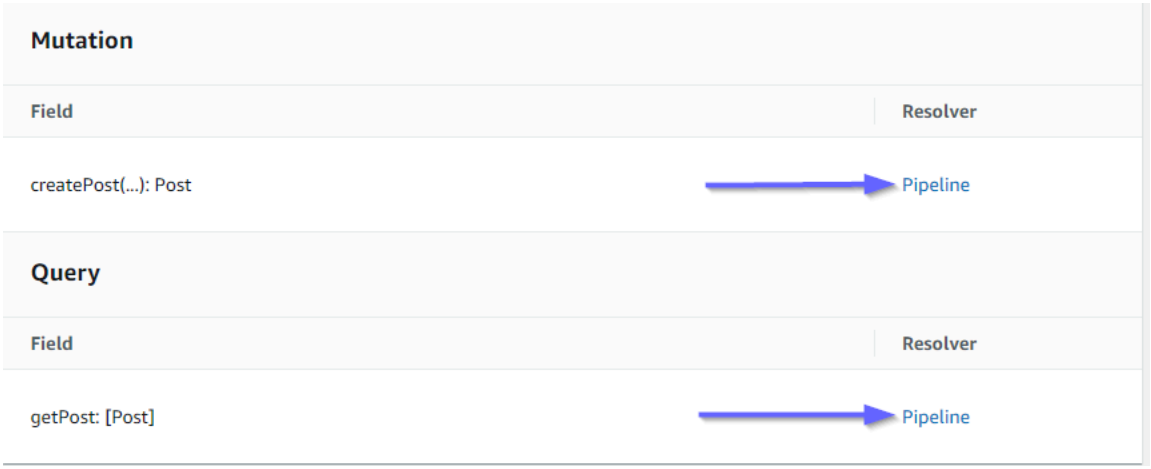
```
      runtime: appsync.FunctionRuntime.JS_1_0_0,
      pipelineConfig: [add_func],
    });

    // Adds a pipeline resolver with the create function
    new appsync.Resolver(this, 'pipeline-resolver-create-posts', {
      api,
      typeName: 'Mutation',
      fieldName: 'createPost',
      code: appsync.Code.fromInline(`
          export function request(ctx) {
          return {};
          }

          export function response(ctx) {
          return ctx.prev.result;
          }
  `),
      runtime: appsync.FunctionRuntime.JS_1_0_0,
      pipelineConfig: [add_func_2],
    });

    // Prints out URL
    new cdk.CfnOutput(this, "GraphQLAPIURL", {
      value: api.graphqlUrl
    });

    // Prints out the AppSync GraphQL API key to the terminal
    new cdk.CfnOutput(this, "GraphQLAPIKey", {
      value: api.apiKey || ''
    });

    // Prints out the stack region to the terminal
    new cdk.CfnOutput(this, "Stack Region", {
      value: this.region
    });
  }
}
```

Dans cet extrait, nous avons ajouté un résolveur de pipeline appelé`pipeline-resolver-create-posts`avec une fonction appelée`func-add-post`attaché à celui-ci. C'est le code qui ajoutera`Posts`à table. L'autre résolveur de pipeline a été appelé`pipeline-resolver-get-posts`avec une fonction appelée`func-get-post`qui récupère`Posts`ajoutée à la table.

Nous allons le déployer pour l'ajouter auAWS AppSyncservice :

```
cdk deploy
```

Vérifions leAWS AppSyncconsole pour voir s'ils étaient attachés à notre API GraphQL :



Cela semble correct. Dans le code, ces deux résolveurs étaient attachés à l'API GraphQL que nous avons créée (désignée par`api`props (valeur présente à la fois dans les résolveurs et les fonctions). Dans l'API GraphQL, les champs auxquels nous avons attaché nos résolveurs étaient également spécifiés dans les accessoires (définis par le`typename`et`fieldname`accessoires dans chaque résolveur).

Voyons si le contenu des résolveurs est correct, en commençant par`pipeline-resolver-get-posts`:

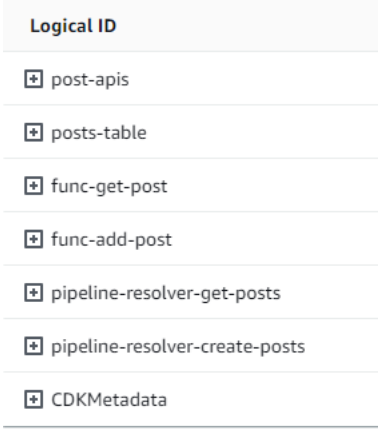
Les gestionnaires avant et après correspondent à nos`code`valeur des accessoires. Nous pouvons également voir qu'une fonction appelée`add_posts_func_1`, qui correspond au nom de la fonction que nous avons attachée au résolveur.

Regardons le contenu du code de cette fonction :

Cela correspond aucodeaccessoires du`add_posts_func_1`fonction. Notre requête a été chargée avec succès, alors vérifions-la :

Ils correspondent également au code. Si nous regardons`get_posts_func_1`:

Tout semble être en place. Pour confirmer cela du point de vue des métadonnées, nous pouvons consulter notre pile dansAWS CloudFormationencore une fois :



Maintenant, nous devons tester ce code en effectuant quelques requêtes.

Mise en œuvre d'un projet CDK - Demandes

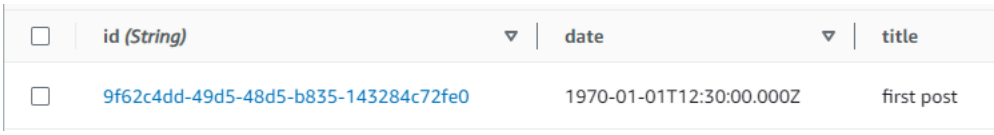
Pour tester notre application dans leAWS AppSyncconsole, nous avons effectué une requête et une mutation :

```
query MyQuery {
  getPost {
    id
    date
    title
  }
}

mutation MyMutation {
  createPost(input: {date: "1970-01-01T12:30:00.000Z", title: "first post"}) {
    date
    id
    title
  }
}

```

`MyMutation`contient un`createPost`opération avec les arguments`1970-01-01T12:30:00.000Z`et`first post`. Il renvoie le`date`et`title`que nous avons transmis ainsi que le`id`valeur. L'exécution de la mutation donne le résultat suivant :

```
{
  "data": {
    "createPost": {
      "date": "1970-01-01T12:30:00.000Z",
      "id": "4dc1c2dd-0aa3-4055-9eca-7c140062ada2",
      "title": "first post"
    }
  }
}
```

Si nous vérifions rapidement la table DynamoDB, nous pouvons voir notre entrée dans la table lorsque nous la scannons :



id *(String)*	date	title
9f62c4dd-49d5-48d5-b835-143284c72fe0	1970-01-01T12:30:00.000Z	first post

De retour dans le`AWS AppSync`console, si nous exécutons la requête pour récupérer ceci`Post`, nous obtenons le résultat suivant :

```
{
  "data": {
    "getPost": [
      {
        "id": "9f62c4dd-49d5-48d5-b835-143284c72fe0",
```

```
        "date": "1970-01-01T12:30:00.000Z",
        "title": "first post"
      }
    ]
  }
}
```

Données en temps réel

AWS AppSyncvous permet d'utiliser des abonnements pour mettre en œuvre des mises à jour d'applications en direct, des notifications push, etc. Lorsque les clients invoquent les opérations d'abonnement GraphQL, une WebSocket connexion sécurisée est automatiquement établie et maintenue par. AWS AppSync Les applications peuvent ensuite distribuer des données en temps réel à partir d'une source de données aux abonnés tout en gérant en AWS AppSync permanence les exigences de connexion et de dimensionnement de l'application. Les sections suivantes vous montreront comment AWS AppSync fonctionnent les abonnements.

Directives d'abonnement au schéma GraphQL

Les abonnements dans AWS AppSync sont appelés en tant que réponse à une mutation. Cela signifie que vous pouvez transformer n'importe quelle source de données dans AWS AppSync en source de données en temps réel, en spécifiant une directive de schéma GraphQL lors d'une mutation.

Les bibliothèques AWS Amplify clientes gèrent automatiquement la gestion des connexions par abonnement. Les bibliothèques utilisent Pure WebSockets comme protocole réseau entre le client et le service.

> **Note**
>
> Pour contrôler l'autorisation au moment de la connexion à un abonnement, vous pouvez utiliser AWS Identity and Access Management (IAM)AWS Lambda, les groupes d'identités Amazon Cognito ou les groupes d'utilisateurs Amazon Cognito pour l'autorisation au niveau du champ. Pour le contrôle précis des accès sur les abonnements, vous pouvez attacher des résolveurs à vos champs d'abonnement et appliquer une logique en utilisant l'identité de l'appelant et les sources de données AWS AppSync . Pour plus d'informations, veuillez consulter Autorisation et authentification.

Les abonnements sont déclenchés à partir des mutations et le jeu de sélection de mutations est envoyé aux abonnés.

L'exemple suivant montre comment utiliser les abonnements GraphQL. Il ne spécifie pas de source de données, car celle-ci peut être Lambda, Amazon DynamoDB ou Amazon Service. OpenSearch

Pour commencer à utiliser les abonnements, vous devez ajouter un point d'entrée d'abonnement à votre schéma comme suit :

```
schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
}
```

Supposons que vous disposez d'un site de blog et que vous souhaitez vous abonner à de nouveaux blogs et à des modifications de blogs existants. Pour ce faire, vous devez ajouter la définition `Subscription` suivante à votre schéma :

```
type Subscription {
    addedPost: Post
    updatedPost: Post
    deletedPost: Post
}
```

Supposons encore que vous disposez des mutations suivantes :

```
type Mutation {
    addPost(id: ID! author: String! title: String content: String url: String): Post!
    updatePost(id: ID! author: String! title: String content: String url: String ups:
 Int! downs: Int! expectedVersion: Int!): Post!
    deletePost(id: ID!): Post!
}
```

Vous pouvez transformer ces champs en temps réel en ajoutant une directive `@aws_subscribe(mutations: ["mutation_field_1", "mutation_field_2"])` pour chacun des abonnements pour lesquels vous souhaitez recevoir des notifications, comme suit :

```
type Subscription {
    addedPost: Post
```

```
    @aws_subscribe(mutations: ["addPost"])
    updatedPost: Post
    @aws_subscribe(mutations: ["updatePost"])
    deletedPost: Post
    @aws_subscribe(mutations: ["deletePost"])
}
```

Comme il `@aws_subscribe(mutations: ["",..,""])` prend un ensemble d'entrées de mutation, vous pouvez spécifier plusieurs mutations, ce qui déclenche un abonnement. Si vous vous abonnez à partir d'un client, votre requête GraphQL peut ressembler à ce qui suit :

```
subscription NewPostSub {
    addedPost {
        __typename
        version
        title
        content
        author
        url
    }
}
```

Cette requête d'abonnement est nécessaire pour les connexions client et l'outillage.

Avec le WebSockets client pur, le filtrage des ensembles de sélection est effectué par client, car chaque client peut définir son propre ensemble de sélection. Dans ce cas, le jeu de sélection d'abonnement doit être un sous-ensemble du jeu de sélection de mutation. Par exemple, un abonnement `addedPost{author title}` lié à la mutation ne `addPost(...){id author title url version}` reçoit que l'auteur et le titre de l'article. Il ne reçoit pas les autres champs. Cependant, si la mutation n'avait pas l'auteur dans son ensemble de sélection, l'abonné obtiendrait une valeur `null` pour le champ auteur (ou une erreur dans le cas où le champ auteur est défini comme « required/not-null » dans le schéma).

L'ensemble de sélection des abonnements est essentiel lors de l'utilisation de Pure WebSockets. Si un champ n'est pas explicitement défini dans l'abonnement, il AWS AppSync ne renvoie pas le champ.

Dans l'exemple précédent, les abonnements n'avaient pas d'arguments. Supposons que votre schéma ressemble à ce qui suit :

```
type Subscription {
```

```
    updatedPost(id:ID! author:String): Post
    @aws_subscribe(mutations: ["updatePost"])
}
```

Dans ce cas, votre client définit un abonnement ainsi :

```
subscription UpdatedPostSub {
    updatedPost(id:"XYZ", author:"ABC") {
        title
        content
    }
}
```

Le type de retour d'un champ `subscription` dans votre schéma doit correspondre au type de retour du champ de mutation correspondant. L'exemple précédent illustrait cela avec `addPost` et `addedPost` qui ont été renvoyés en tant que type de `Post`.

Pour configurer les abonnements sur le client, voirCréation d'une application client.

Utilisation d'arguments d'abonnement

Pour utiliser les abonnements GraphQL, il est important de comprendre quand et comment utiliser les arguments. Vous pouvez apporter des modifications subtiles pour modifier comment et quand informer les clients des mutations survenues. Pour ce faire, consultez l'exemple de schéma du chapitre de démarrage rapide, qui crée « Todos ». Pour cet exemple de schéma, les mutations suivantes sont définies :

```
type Mutation {
    createTodo(input: CreateTodoInput!): Todo
    updateTodo(input: UpdateTodoInput!): Todo
    deleteTodo(input: DeleteTodoInput!): Todo
}
```

Dans l'exemple par défaut, les clients peuvent s'abonner aux mises à jour de n'importe quel produit en `Todo` utilisant le `onUpdateTodo` `subscription` paramètre sans argument :

```
subscription OnUpdateTodo {
  onUpdateTodo {
    description
    id
    name
```

```
    when
  }
}
```

Vous pouvez filtrer votre `subscription` en utilisant ses arguments. Par exemple, pour déclencher un uniquement `subscription` lorsqu'un fichier `todo` contenant un élément spécifique `ID` est mis à jour, spécifiez la `ID` valeur :

```
subscription OnUpdateTodo {
  onUpdateTodo(id: "a-todo-id") {
    description
    id
    name
    when
  }
}
```

Vous pouvez également transmettre plusieurs arguments. Par exemple, ce qui suit `subscription` montre comment être informé de toute `Todo` mise à jour à un endroit et à une heure spécifiques :

```
subscription todosAtHome {
  onUpdateTodo(when: "tomorrow", where: "at home") {
    description
    id
    name
    when
    where
  }
}
```

Notez que tous les arguments sont facultatifs. Si vous ne spécifiez aucun argument dans votre application`subscription`, vous serez abonné à toutes les `Todo` mises à jour effectuées dans votre application. Cependant, vous pouvez mettre à jour votre définition `subscription` de champ pour exiger l'`ID`argument. Cela forcerait la réponse d'un `todo` s spécifique `todo` au lieu de tous :

```
onUpdateTodo(
  id: ID!,
  name: String,
  when: String,
  where: String,
  description: String
```

```
): Todo
```

La valeur null de l'argument a une signification

Lorsque vous effectuez une requête d'abonnement dans AWS AppSync, une valeur d'argument `null` filtre les résultats différemment de l'omission complète de l'argument.

Revenons à l'exemple d'API todos où nous pourrions créer des todos. Consultez l'exemple de schéma du chapitre de démarrage rapide.

Modifions notre schéma pour inclure un nouveau `owner` champ, sur le `Todo` type, qui décrit le propriétaire. Le `owner` champ n'est pas obligatoire et peut uniquement être activé`UpdateTodoInput`. Consultez la version simplifiée suivante du schéma :

```
type Todo {
  id: ID!
  name: String!
  when: String!
  where: String!
  description: String!
  owner: String
}

input CreateTodoInput {
  name: String!
  when: String!
  where: String!
  description: String!
}

input UpdateTodoInput {
  id: ID!
  name: String
  when: String
  where: String
  description: String
  owner: String
}

type Subscription {
    onUpdateTodo(
        id: ID,
        name: String,
```

```
        when: String,
        where: String,
        description: String
    ): Todo @aws_subscribe(mutations: ["updateTodo"])
}
```

L'abonnement suivant renvoie toutes les Todo mises à jour :

```
subscription MySubscription {
  onUpdateTodo {
    description
    id
    name
    when
    where
  }
}
```

Si vous modifiez l'abonnement précédent pour ajouter l'argument de champowner: null, vous posez maintenant une autre question. Cet abonnement enregistre désormais le client pour qu'il soit informé de toutes les Todo mises à jour pour lesquelles aucun propriétaire n'a été fourni.

```
subscription MySubscription {
  onUpdateTodo(owner: null) {
    description
    id
    name
    when
    where
  }
}
```

> **Note**
>
> Depuis le 1er janvier 2022, MQTT over n' WebSockets est plus disponible en tant que protocole pour les abonnements AWS AppSync GraphQL dans les API. Pure WebSockets est le seul protocole pris en charge dansAWS AppSync.
> Les clients basés sur le AWS AppSync SDK ou les bibliothèques Amplify, publiés après novembre 2019, utilisent automatiquement WebSockets pure par défaut. La mise à niveau des clients vers la dernière version leur permet AWS AppSync d'utiliser le WebSockets moteur pur.

Pure WebSockets propose une charge utile plus importante (240 Ko), une plus grande variété d'options client et des CloudWatch indicateurs améliorés. Pour plus d'informations sur l'utilisation de WebSocket clients purs, consultezCréation d'un WebSocket client en temps réel.

Création d'API pub/sub génériques alimentées par serverless WebSockets

Certaines applications ne nécessitent que de simples WebSocket API permettant aux clients d'écouter une chaîne ou un sujet spécifique. Les données JSON génériques ne présentant aucune forme spécifique ni aucune exigence typographique stricte peuvent être transmises aux clients écoutant l'un de ces canaux selon un schéma purement et simple de publication/abonnement (pub/sub).

AWS AppSync À utiliser pour implémenter des WebSocket API pub/sub simples avec peu ou pas de connaissances GraphQL en quelques minutes en générant automatiquement du code GraphQL à la fois sur le backend de l'API et du côté client.

Création et configuration d'API pub-sub

Pour commencer, procédez comme suit :

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync .
   - Dans le Tableau de bord, choisissez Créer une API.
2. Sur l'écran suivant, choisissez Créer une API en temps réel, puis cliquez sur Suivant.
3. Entrez un nom convivial pour votre API pub/sub.
4. Vous pouvez activer les fonctionnalités d'API privées, mais nous vous recommandons de les désactiver pour le moment. Choisissez Suivant.
5. Vous pouvez choisir de générer automatiquement une API pub/sub fonctionnelle à l'aide de. WebSockets Nous vous recommandons également de désactiver cette fonctionnalité pour le moment. Choisissez Suivant.
6. Choisissez Create API, puis attendez quelques minutes. Une nouvelle API AWS AppSync pub/sub préconfigurée sera créée dans votre compte. AWS

L'API utilise des AWS AppSync résolveurs locaux intégrés (pour plus d'informations sur l'utilisation des résolveurs locaux, voir Tutoriel : résolveurs locaux dans le guide du AWS AppSync développeur)

pour gérer plusieurs canaux et WebSocket connexions pub/sub temporaires, qui fournissent et filtrent automatiquement les données aux clients abonnés en fonction uniquement du nom du canal. Les appels d'API sont autorisés à l'aide d'une clé d'API.

Une fois l'API déployée, quelques étapes supplémentaires vous sont proposées pour générer du code client et l'intégrer à votre application client. À titre d'exemple sur la façon d'intégrer rapidement un client, ce guide utilisera une simple application Web React.

1. Commencez par créer une application React standard à l'aide de NPM sur votre machine locale :

```
$ npx create-react-app mypubsub-app
$ cd mypubsub-app
```

> **Note**
>
> Cet exemple utilise les bibliothèques Amplify pour connecter les clients à l'API principale. Cependant, il n'est pas nécessaire de créer un projet Amplify CLI localement. Bien que React soit le client de choix dans cet exemple, les bibliothèques Amplify prennent également en charge les clients iOS, Android et Flutter, offrant les mêmes fonctionnalités dans ces différents environnements d'exécution. Les clients Amplify pris en charge fournissent des abstractions simples pour interagir avec les backends d'API AWS AppSync GraphQL avec quelques lignes de code, y compris des WebSocket fonctionnalités intégrées entièrement compatibles avec le protocole en temps réel : AWS AppSync WebSocket
>
> ```
> $ npm install @aws-amplify/api
> ```

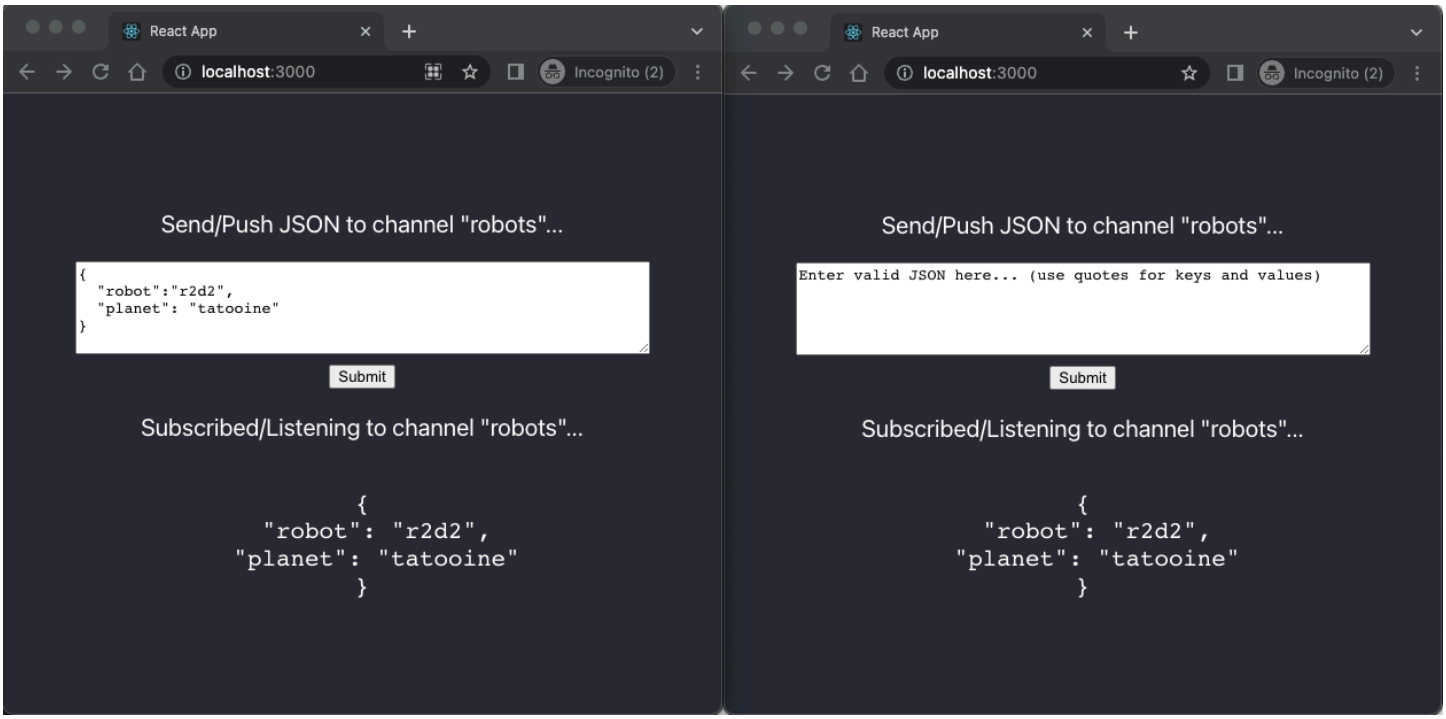
2. Dans la AWS AppSync console JavaScript, sélectionnez puis Télécharger pour télécharger un seul fichier contenant les détails de configuration de l'API et le code d'opérations GraphQL généré.
3. Copiez le fichier téléchargé dans le `/src` dossier de votre projet React.
4. Remplacez ensuite le contenu du `src/App.js` fichier standard existant par l'exemple de code client disponible dans la console.
5. Utilisez la commande suivante pour démarrer l'application localement :

```
$ npm start
```

6. Pour tester l'envoi et la réception de données en temps réel, ouvrez deux fenêtres de navigateur et accédez à *localhost:3000*. L'exemple d'application est configuré pour envoyer des données JSON génériques à un canal codé en dur nommé *robots*.
7. Dans l'une des fenêtres du navigateur, entrez le blob JSON suivant dans la zone de texte, puis cliquez sur Soumettre :

```
{
  "robot":"r2d2",
  "planet": "tatooine"
}
```

Les deux instances de navigateur sont abonnées au canal des *robots* et reçoivent les données publiées en temps réel, affichées en bas de l'application Web :



Tout le code d'API GraphQL nécessaire, y compris le schéma, les résolveurs et les opérations, est généré automatiquement pour permettre un cas d'utilisation générique de type pub/sub. Sur le backend, les données sont publiées sur le point AWS AppSync de terminaison en temps réel avec une mutation GraphQL telle que la suivante :

```
mutation PublishData {
    publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
        data
```

```
        name
    }
}
```

Les abonnés accèdent aux données publiées envoyées au canal temporaire spécifique avec un abonnement GraphQL associé :

```
subscription SubscribeToData {
    subscribe(name:"channel") {
        name
        data
    }
}
```

Implémentation d'API pub-sub dans des applications existantes

Au cas où vous auriez simplement besoin d'implémenter une fonctionnalité en temps réel dans une application existante, cette configuration générique d'API pub/sub peut être facilement intégrée à n'importe quelle application ou technologie d'API. Bien qu'il y ait des avantages à utiliser un point de terminaison d'API unique pour accéder, manipuler et combiner en toute sécurité des données provenant d'une ou de plusieurs sources de données en un seul appel réseau avec GraphQL, il n'est pas nécessaire de convertir ou de reconstruire une application basée sur REST existante à partir de zéro afin de tirer parti des fonctionnalités en temps réel AWS AppSync de celle-ci. Par exemple, vous pouvez avoir une charge de travail CRUD existante dans un point de terminaison d'API distinct, les clients envoyant et recevant des messages ou des événements de l'application existante à l'API générique pub/sub à des fins en temps réel et à des fins pub/sub uniquement.

Filtrage amélioré des abonnements

DansAWS AppSync, vous pouvez définir et activer la logique métier pour le filtrage des données sur le backend directement dans les résolveurs d'abonnement à l'API GraphQL en utilisant des filtres prenant en charge des opérateurs logiques supplémentaires. Vous pouvez configurer ces filtres, contrairement aux arguments d'abonnement définis dans la requête d'abonnement du client. Pour plus d'informations sur l'utilisation des arguments d'abonnement, consultezUtilisation d'arguments d'abonnement. Pour obtenir la liste des opérateurs, voirRéférence de l'utilitaire du modèle de mappage Resolver.

Aux fins du présent document, nous répartissons le filtrage des données en temps réel dans les catégories suivantes :

- Filtrage de base : filtrage basé sur les arguments définis par le client dans la requête d'abonnement.
- Filtrage amélioré : filtrage basé sur une logique définie de manière centralisée dans le backend du AWS AppSync service.

Les sections suivantes expliquent comment configurer des filtres d'abonnement améliorés et présentent leur utilisation pratique.

Définition des abonnements dans votre schéma GraphQL

Pour utiliser des filtres d'abonnement améliorés, vous définissez l'abonnement dans le schéma GraphQL, puis vous définissez le filtre amélioré à l'aide d'une extension de filtrage. Pour illustrer le fonctionnement du filtrage amélioré des abonnementsAWS AppSync, utilisez le schéma GraphQL suivant, qui définit une API de système de gestion des tickets, à titre d'exemple :

```
type Ticket {
 id: ID
 createdAt: AWSDateTime
 content: String
 severity: Int
 priority: Priority
 category: String
 group: String
 status: String

}

type Mutation {
 createTicket(input: TicketInput): Ticket
}

type Query {
 getTicket(id: ID!): Ticket
}

type Subscription {
 onSpecialTicketCreated: Ticket @aws_subscribe(mutations: ["createTicket"])
 onGroupTicketCreated(group: String!): Ticket @aws_subscribe(mutations:
 ["createTicket"])
}
```

```
enum Priority {
 none
 lowest
 low
 medium
 high
 highest
}

input TicketInput {
 content: String
 severity: Int
 priority: Priority
 category: String
 group: String
```

Supposons que vous créiez une source de `NONE` données pour votre API, puis que vous associiez un résolveur à la `createTicket` mutation à l'aide de cette source de données. Vos gestionnaires peuvent ressembler à ceci :

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
 return {
  payload: {
   id: util.autoId(),
   createdAt: util.time.nowISO8601(),
   status: 'pending',
   ...ctx.args.input,
  },
 };
}

export function response(ctx) {
 return ctx.result;
}
```

> **Note**
>
> Les filtres améliorés sont activés dans le gestionnaire du résolveur GraphQL dans le cadre d'un abonnement donné. Pour plus d'informations, consultez la section Référence du résolveur.

Pour implémenter le comportement du filtre amélioré, vous devez utiliser la `extensions.setSubscriptionFilter()` fonction pour définir une expression de filtre évaluée par rapport aux données publiées à partir d'une mutation GraphQL susceptible d'intéresser les clients abonnés. Pour plus d'informations sur les extensions de filtrage, consultez la section Extensions.

La section suivante explique comment utiliser les extensions de filtrage pour implémenter des filtres améliorés.

Création de filtres d'abonnement améliorés à l'aide d'extensions de filtrage

Les filtres améliorés sont écrits en JSON dans le gestionnaire de réponses des résolveurs de l'abonnement. Les filtres peuvent être regroupés dans une liste appelée `filterGroup` a. Les filtres sont définis à l'aide d'au moins une règle, chacune comportant des champs, des opérateurs et des valeurs. Définissons un nouveau résolveur pour on`SpecialTicketCreated` configurer un filtre amélioré. Vous pouvez configurer plusieurs règles dans un filtre qui sont évaluées à l'aide de la logique ET, tandis que plusieurs filtres d'un groupe de filtres sont évalués à l'aide de la logique OR :

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
 // simplfy return null for the payload
 return { payload: null };
}

export function response(ctx) {
 const filter = {
  or: [
   { severity: { ge: 7 }, priority: { in: ['high', 'medium'] } },
   { category: { eq: 'security' }, group: { in: ['admin', 'operators'] } },
  ],
 };
 extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));
```

```
  // important: return null in the response
 return null;
}
```

Sur la base des filtres définis dans l'exemple précédent, les tickets importants sont automatiquement envoyés aux clients API abonnés si un ticket est créé avec :

- `priority`niveau `high` ou `medium`

  AND

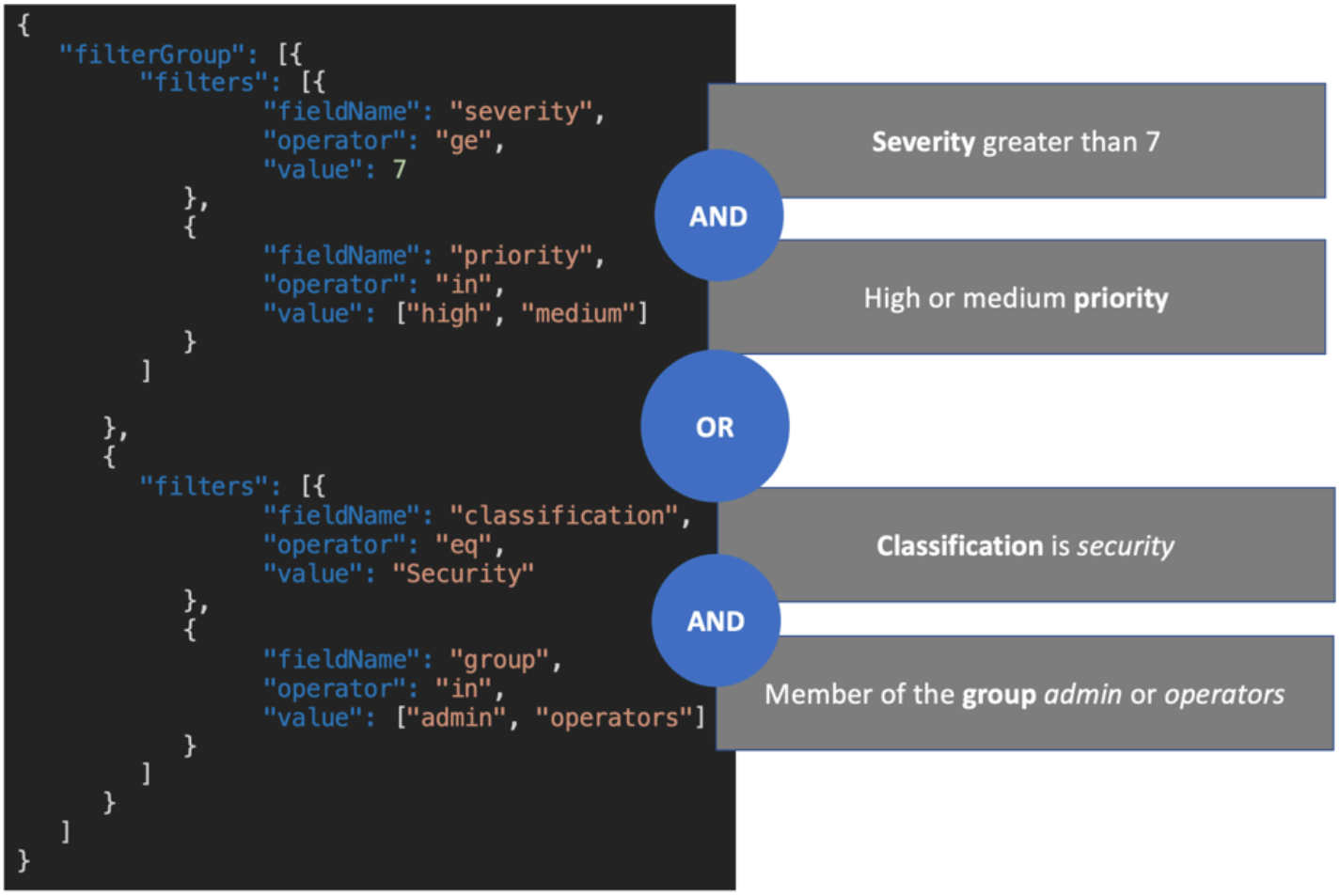
- `severity`niveau supérieur ou égal à 7 (`ge`)

OU

- `classification`billet réglé à `Security`

  AND

- `group`assignation définie sur `admin` ou `operators`



Les filtres définis dans le résolveur d'abonnement (filtrage amélioré) ont priorité sur le filtrage basé uniquement sur les arguments d'abonnement (filtrage de base). Pour plus d'informations sur l'utilisation des arguments d'abonnement, consultez la section Utilisation des arguments d'abonnement).

Si un argument est défini et requis dans le schéma GraphQL de l'abonnement, le filtrage basé sur l'argument donné n'a lieu que si l'argument est défini en tant que règle dans la méthode du résolveur. `extensions.setSubscriptionFilter()` Toutefois, s'il n'existe aucune méthode de `extensions` filtrage dans le résolveur d'abonnement, les arguments définis dans le client ne sont utilisés que pour le filtrage de base. Vous ne pouvez pas utiliser simultanément le filtrage de base et le filtrage amélioré.

Vous pouvez utiliser la `context` variable dans la logique d'extension de filtre de l'abonnement pour accéder aux informations contextuelles relatives à la demande. Par exemple, lorsque vous utilisez des groupes d'utilisateurs Amazon Cognito, des autorisateurs personnalisés OIDC ou Lambda pour l'autorisation, vous pouvez récupérer des informations sur vos utilisateurs au `context.identity`

moment de l'établissement de l'abonnement. Vous pouvez utiliser ces informations pour établir des filtres en fonction de l'identité de vos utilisateurs.

Supposons maintenant que vous souhaitez implémenter le comportement de filtre amélioré pouron`GroupTicketCreated`. L'on`GroupTicketCreated`abonnement nécessite un `group` nom obligatoire comme argument. Une fois créés, un `pending` statut est automatiquement attribué aux tickets. Vous pouvez configurer un filtre d'abonnement pour ne recevoir que les tickets nouvellement créés appartenant au groupe fourni :

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
 // simplfy return null for the payload
 return { payload: null };
}

export function response(ctx) {
 const filter = { group: { eq: ctx.args.group }, status: { eq: 'pending' } };
 extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

 return null;
}
```

Lorsque les données sont publiées à l'aide d'une mutation, comme dans l'exemple suivant :

```
mutation CreateTicket {
  createTicket(input: {priority: medium, severity: 2, group: "aws"}) {
    id
    priority
    severity
    status
    group
    createdAt
  }
}
```

Les clients abonnés écoutent les données à transmettre automatiquement WebSockets dès qu'un ticket est créé avec la `createTicket` mutation :

```
subscription OnGroup {
  onGroupTicketCreated(group: "aws") {
```

```
    category
    status
    severity
    priority
    id
    group
    createdAt
    content
  }
}
```

Les clients peuvent être abonnés sans arguments car la logique de filtrage est implémentée dans le AWS AppSync service avec un filtrage amélioré, ce qui simplifie le code client. Les clients reçoivent des données uniquement si les critères de filtre définis sont remplis.

Définition de filtres améliorés pour les champs de schéma imbriqués

Vous pouvez utiliser le filtrage des abonnements amélioré pour filtrer les champs de schéma imbriqués. Supposons que nous ayons modifié le schéma de la section précédente pour inclure les types de localisation et d'adresse :

```
type Ticket {
 id: ID
 createdAt: AWSDateTime
 content: String
 severity: Int
 priority: Priority
 category: String
 group: String
 status: String
 location: ProblemLocation
}

type Mutation {
 createTicket(input: TicketInput): Ticket
}

type Query {
 getTicket(id: ID!): Ticket
}

type Subscription {
 onSpecialTicketCreated: Ticket @aws_subscribe(mutations: ["createTicket"])
```

```
 onGroupTicketCreated(group: String!): Ticket @aws_subscribe(mutations:
 ["createTicket"])
}

type ProblemLocation {
 address: Address
}

type Address {
 country: String
}

enum Priority {
 none
 lowest
 low
 medium
 high
 highest
}

input TicketInput {
 content: String
 severity: Int
 priority: Priority
 category: String
 group: String
 location: AWSJSON
```

Avec ce schéma, vous pouvez utiliser un `.` séparateur pour représenter l'imbrication. L'exemple suivant ajoute une règle de filtre pour un champ de schéma imbriqué sous`location.address.country`. L'abonnement sera déclenché si l'adresse du ticket est définie comme suit USA :

```
import { util, extensions } from '@aws-appsync/utils';

export const request = (ctx) => ({ payload: null });

export function response(ctx) {
 const filter = {
  or: [
   { severity: { ge: 7 }, priority: { in: ['high', 'medium'] } },
   { category: { eq: 'security' }, group: { in: ['admin', 'operators'] } },
```

```
    { 'location.address.country': { eq: 'USA' } },
   ],
  };
  extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));
  return null;
}
```

Dans l'exemple ci-dessus, `location` représente le niveau d'imbrication un, `address` représente le niveau d'imbrication deux et `country` représente le niveau d'imbrication trois, tous ces éléments étant séparés par le séparateur. `.`

Vous pouvez tester cet abonnement en utilisant la `createTicket` mutation :

```
mutation CreateTicketInUSA {
  createTicket(input: {location: "{\"address\":{\"country\":\"USA\"}}"}) {
    category
    content
    createdAt
    group
    id
    location {
      address {
        country
      }
    }
    priority
    severity
    status
  }
}
```

Définition de filtres améliorés à partir du client

Vous pouvez utiliser le filtrage de base dans GraphQL avec des arguments d'abonnement. Le client qui effectue l'appel dans la requête d'abonnement définit les valeurs des arguments. Lorsque des filtres améliorés sont activés dans un résolveur AWS AppSync d'abonnement avec le `extensions` filtrage, les filtres principaux définis dans le résolveur ont la priorité et la priorité.

Configurez des filtres améliorés dynamiques définis par le client à l'aide d'un `filter` argument dans l'abonnement. Lorsque vous configurez ces filtres, vous devez mettre à jour le schéma GraphQL pour refléter le nouvel argument :

```
...
type Subscription {
    onSpecialTicketCreated(filter: String): Ticket
        @aws_subscribe(mutations: ["createTicket"])
}
...
```

Le client peut ensuite envoyer une demande d'abonnement comme dans l'exemple suivant :

```
subscription onSpecialTicketCreated($filter: String) {
     onSpecialTicketCreated(filter: $filter) {
        id
        group
        description
        priority
        severity
     }
 }
```

Vous pouvez configurer la variable de requête comme dans l'exemple suivant :

```
{"filter" : "{\"severity\":{\"le\":2}}"}
```

L'utilitaire de `util.transform.toSubscriptionFilter()` résolution peut être implémenté dans le modèle de mappage des réponses aux abonnements pour appliquer le filtre défini dans l'argument d'abonnement pour chaque client :

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
 // simplfy return null for the payload
 return { payload: null };
}

export function response(ctx) {
 const filter = ctx.args.filter;
 extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));
 return null;
}
```

Grâce à cette stratégie, les clients peuvent définir leurs propres filtres qui utilisent une logique de filtrage améliorée et des opérateurs supplémentaires. Les filtres sont attribués lorsqu'un client donné invoque la requête d'abonnement via une WebSocket connexion sécurisée. Pour plus d'informations sur l'utilitaire de transformation pour un filtrage amélioré, notamment sur le format de la charge utile de la variable de `filter` requête, consultez la section Présentation des JavaScript résolveurs.

Restrictions de filtrage améliorées supplémentaires

Vous trouverez ci-dessous plusieurs cas d'utilisation dans lesquels des restrictions supplémentaires sont imposées aux filtres améliorés :

- Les filtres améliorés ne prennent pas en charge le filtrage pour les listes d'objets de niveau supérieur. Dans ce cas d'utilisation, les données publiées issues de la mutation seront ignorées pour les abonnements améliorés.
- AWS AppSync prend en charge jusqu'à cinq niveaux de nidification. Les filtres appliqués aux champs de schéma après le niveau d'imbrication 5 seront ignorés. Prenons la réponse GraphQL ci-dessous. L'entrée `continent` sur le terrain `venue.address.country.metadata.continent` est autorisée car il s'agit d'un nid de niveau 5. Cependant, `financial` comme `venue.address.country.metadata.capital.financial` il s'agit d'un nid de niveau 6, le filtre ne fonctionnera pas :

```
{
    "data": {
        "onCreateFilterEvent": {
            "venue": {
                "address": {
                    "country": {
                        "metadata": {
                            "capital": {
                                "financial": "New York"
                            },
                            "continent" : "North America"
                        }
                    },
                    "state": "WA"
                },
                "builtYear": 2023
            },
            "private": false,
        }
    }
```

```
}
```

Désinscription des WebSocket connexions à l'aide de filtres

DansAWS AppSync, vous pouvez vous désinscrire de force et fermer (invalider) une WebSocket connexion depuis un client connecté en fonction d'une logique de filtrage spécifique. Cela est utile dans les scénarios liés aux autorisations, par exemple lorsque vous supprimez un utilisateur d'un groupe.

L'invalidation de l'abonnement se produit en réponse à une charge utile définie dans une mutation. Nous vous recommandons de traiter les mutations utilisées pour invalider les connexions d'abonnement comme des opérations administratives dans votre API et de définir les autorisations en conséquence en limitant leur utilisation à un utilisateur administrateur, à un groupe ou à un service principal. Par exemple, en utilisant des directives d'autorisation de schéma telles que `@aws_auth(cognito_groups: ["Administrators"])` ou`@aws_iam`. Pour plus d'informations, consultez la section Utilisation de modes d'autorisation supplémentaires.

Les filtres d'invalidation utilisent la même syntaxe et la même logique que les filtres d'abonnement améliorés. Définissez ces filtres à l'aide des utilitaires suivants :

- `extensions.invalidateSubscriptions()`— Défini dans le gestionnaire de réponse du résolveur GraphQL pour une mutation.
- `extensions.setSubscriptionInvalidationFilter()`— Défini dans le gestionnaire de réponses du résolveur GraphQL pour les abonnements liés à la mutation.

Pour plus d'informations sur les extensions de filtrage d'invalidation, consultez la section Présentation JavaScript des résolveurs.

Utilisation de l'invalidation de l'abonnement

Pour voir comment fonctionne l'invalidation des abonnementsAWS AppSync, utilisez le schéma GraphQL suivant :

```
type User {
  userId: ID!
  groupId: ID!
}
```

```
type Group {
  groupId: ID!
  name: String!
  members: [ID!]!
}

type GroupMessage {
  userId: ID!
  groupId: ID!
  message: String!
}

type Mutation {
    createGroupMessage(userId: ID!, groupId : ID!, message: String!): GroupMessage
    removeUserFromGroup(userId: ID!, groupId : ID!) : User @aws_iam
}

type Subscription {
    onGroupMessageCreated(userId: ID!, groupId : ID!): GroupMessage
        @aws_subscribe(mutations: ["createGroupMessage"])
}

type Query {
 none: String
}
```

Définissez un filtre d'invalidation dans le code du résolveur de `removeUserFromGroup` mutations :

```
import { extensions } from '@aws-appsync/utils';

export function request(ctx) {
 return { payload: null };
}

export function response(ctx) {
 const { userId, groupId } = ctx.args;
 extensions.invalidateSubscriptions({
  subscriptionField: 'onGroupMessageCreated',
  payload: { userId, groupId },
 });
 return { userId, groupId };
}
```

Lorsque la mutation est invoquée, les données définies dans l'`payload`objet sont utilisées pour annuler l'abonnement défini dans`subscriptionField`. Un filtre d'invalidation est également défini dans le modèle de mappage des réponses de l'`onGroupMessageCreated`abonnement.

Si la `extensions.invalidateSubscriptions()` charge utile contient un identifiant qui correspond aux identifiants du client abonné tels que définis dans le filtre, l'abonnement correspondant est désabonné. De plus, la WebSocket connexion est fermée. Définissez le code de résolution d'abonnement pour l'`onGroupMessageCreated`abonnement :

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
 // simplfy return null for the payload
 return { payload: null };
}

export function response(ctx) {
 const filter = { groupId: { eq: ctx.args.groupId } };
 extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

 const invalidation = { groupId: { eq: ctx.args.groupId }, userId: { eq:
 ctx.args.userId } };
 extensions.setSubscriptionInvalidationFilter(util.transform.toSubscriptionFilter(invalidation)

 return null;
}
```

Notez que le gestionnaire de réponse aux abonnements peut avoir des filtres d'abonnement et des filtres d'invalidation définis en même temps.

Supposons, par exemple, que le client A abonne un nouvel utilisateur avec l'ID *`user-1`* au groupe ayant l'ID *`group-1`* en utilisant la demande d'abonnement suivante :

```
onGroupMessageCreated(userId : "user-1", groupId: :"group-1"){...}
```

AWS AppSyncexécute le résolveur d'abonnement, qui génère des filtres d'abonnement et d'invalidation tels que définis dans le modèle de mappage de `onGroupMessageCreated` réponses précédent. Pour le client A, les filtres d'abonnement autorisent l'envoi de données uniquement à*`group-1`*, et les filtres d'invalidation sont définis pour *`user-1`* les *`group-1`* deux.

Supposons maintenant que le client B abonne un utilisateur avec l'ID *user-2* à un groupe avec l'ID *group-2* en utilisant la demande d'abonnement suivante :

```
onGroupMessageCreated(userId : "user-2", groupId: :"group-2"){...}
```

AWS AppSyncexécute le résolveur d'abonnement, qui génère des filtres d'abonnement et d'invalidation. Pour le client B, les filtres d'abonnement autorisent l'envoi de données uniquement à*group-2*, et les filtres d'invalidation sont définis pour les deux *user-2* et*group-2*.

Supposons ensuite qu'un nouveau message de groupe avec l'ID *message-1* soit créé à l'aide d'une demande de mutation, comme dans l'exemple suivant :

```
createGroupMessage(id: "message-1", groupId :
      "group-1", message: "test message"){...}
```

Les clients abonnés correspondant aux filtres définis reçoivent automatiquement la charge utile de données suivante via WebSockets :

```
{
  "data": {
    "onGroupMessageCreated": {
      "id": "message-1",
      "groupId": "group-1",
      "message": "test message",
    }
  }
}
```

Le client A reçoit le message car les critères de filtrage correspondent au filtre d'abonnement défini. Cependant, le client B ne reçoit pas le message, car l'utilisateur n'en fait pas partie*group-1*. De plus, la demande ne correspond pas au filtre d'abonnement défini dans le résolveur d'abonnement.

Enfin, supposons que cela *user-1* soit supprimé *group-1* lors de l'utilisation de la demande de mutation suivante :

```
removeUserFromGroup(userId: "user-1", groupId : "group-1"){...}
```

La mutation initie une invalidation d'abonnement telle que définie dans le code du gestionnaire de réponse du `extensions.invalidateSubscriptions()` résolveur. AWS AppSyncdésabonne

ensuite le client A et ferme sa WebSocket connexion. Le client B n'est pas affecté, car la charge utile d'invalidation définie dans la mutation ne correspond pas à son utilisateur ou à son groupe.

Lorsqu'une connexion est AWS AppSync invalidée, le client reçoit un message confirmant qu'il est désabonné :

```
{
  "message": "Subscription complete."
}
```

Utilisation de variables contextuelles dans les filtres d'invalidation des abonnements

Comme pour les filtres d'abonnement améliorés, vous pouvez utiliser la `contextvariable` de l'extension du filtre d'invalidation d'abonnement pour accéder à certaines données.

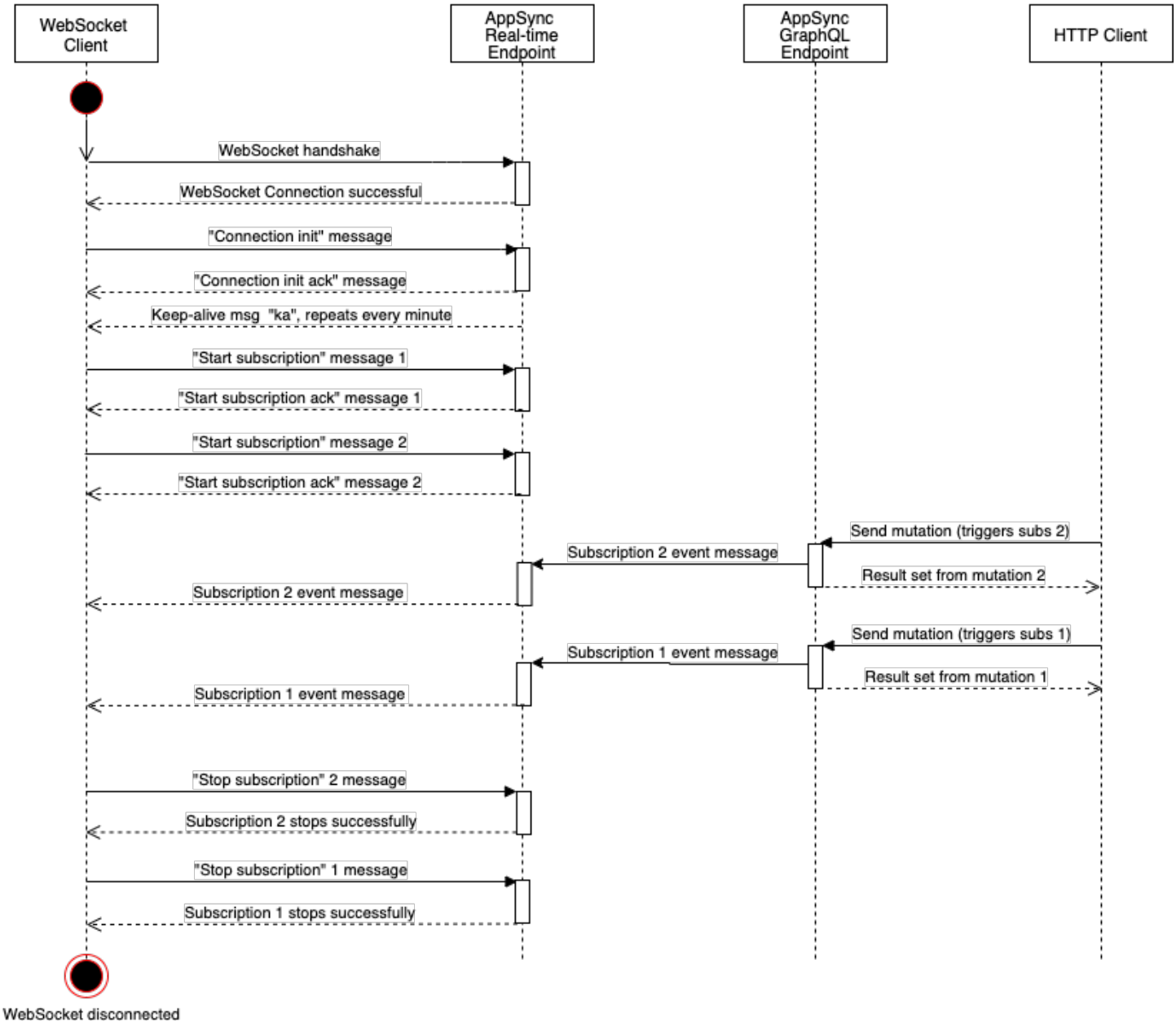
Par exemple, il est possible de configurer une adresse e-mail comme charge utile d'invalidation lors de la mutation, puis de l'associer à l'attribut e-mail ou à la réclamation d'un utilisateur abonné autorisé par les groupes d'utilisateurs Amazon Cognito ou OpenID Connect. Le filtre d'invalidation défini dans l'invalidateur `extensions.setSubscriptionInvalidationFilter()` d'abonnement vérifie si l'adresse e-mail définie par la `extensions.invalidateSubscriptions()` charge utile de la mutation correspond à l'adresse e-mail récupérée à partir du jeton JWT de l'utilisateur, déclenchant ainsi l'invalidation. `context.identity.claims.email`

Création d'un WebSocket client en temps réel

Les sections suivantes vous présenteront l'architecture qui sous-tend les fonctionnalités en temps réel AWS AppSync de ces fonctionnalités.

WebSocket Implémentation du client en temps réel pour les abonnements GraphQL

Le schéma de séquence et les étapes suivants montrent le flux de travail des abonnements en temps réel entre le WebSocket client, le client HTTP et AWS AppSync.



1. Le client établit une WebSocket connexion avec le point de terminaison AWS AppSync en temps réel. En cas d'erreur réseau, le client doit effectuer une interruption exponentielle instable. Pour plus d'informations, consultez la section Exponential Backoff and jitter sur le AWS blog d'architecture.
2. Après avoir établi la WebSocket connexion avec succès, le client envoie un `connection_init` message.
3. Le client attend un `connection_ack` message de AWS AppSync. Ce message inclut un `connectionTimeoutMs` paramètre, qui est le temps d'attente maximal en millisecondes pour un message `"ka"` (keep-alive).

4. AWS AppSync envoie `"ka"` des messages périodiquement. Le client enregistre l'heure à laquelle il a reçu chaque `"ka"` message. Si le client ne reçoit aucun `"ka"` message dans les `connectionTimeoutMs` millisecondes, il doit fermer la connexion.
5. Le client enregistre l'abonnement en envoyant un message d'abonnement `start`. Une seule WebSocket connexion prend en charge plusieurs abonnements, même s'ils utilisent des modes d'autorisation différents.
6. Le client attend l'envoi des messages `start_ack` par AWS AppSync pour confirmer les abonnements réussis. En cas d'erreur, AWS AppSync renvoie un message `"type": "error"`.
7. Le client écoute les événements d'abonnement, qui sont envoyés après l'appel d'une mutation correspondante. Les requêtes et les mutations sont généralement envoyées via `https://` au point de terminaison GraphQL AWS AppSync . Les abonnements passent par le point de terminaison AWS AppSync en temps réel à l'aide du WebSocket protocole secure (`wss://`).
8. Le client annule l'abonnement en envoyant un message d'abonnement `stop`.
9. Après avoir annulé tous les abonnements et vérifié qu'aucun message n'est transféré via le WebSocket, le client peut se déconnecter de la WebSocket connexion.

Détails de la poignée de main pour établir la connexion WebSocket

Pour établir une connexion et initier une poignée de main réussie avec AWS AppSync, un WebSocket client a besoin des éléments suivants :

- Le point final AWS AppSync en temps réel
- Chaîne de requête contenant `header` les `payload` paramètres suivants :
  - `header` : contient des informations relatives au point de terminaison et à l'autorisation AWS AppSync. Il s'agit d'une chaîne codée en base64 provenant d'un objet JSON sous forme de chaînes. Le contenu de l'objet JSON varie en fonction du mode d'autorisation.
  - `payload`: chaîne codée en Base64 de. `payload`

Avec ces exigences, un WebSocket client peut se connecter à l'URL, qui contient le point de terminaison en temps réel avec la chaîne de requête, en utilisant `graphql-ws` comme WebSocket protocole.

Découverte du point de terminaison en temps réel à partir du point de terminaison GraphQL

Le point de terminaison GraphQL AWS AppSync et le point de terminaison en temps réel AWS AppSync sont légèrement différents dans le protocole et le domaine. Vous pouvez récupérer le point

de terminaison GraphQL à l'aide de la commande AWS Command Line Interface (AWS CLI). `aws appsync get-graphql-api`

AWS AppSync Point de terminaison GraphQL :

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync point final en temps réel :

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

Les applications peuvent se connecter au point de terminaison GraphQL AWS AppSync (`https://`) en utilisant n'importe quel client HTTP pour les requêtes et les mutations. Les applications peuvent se connecter au point de terminaison AWS AppSync en temps réel (`wss://`) en utilisant n'importe quel WebSocket client pour les abonnements.

Avec les noms de domaine personnalisés, vous pouvez interagir avec les deux points de terminaison en utilisant un seul domaine. Par exemple, si vous le configurez `api.example.com` en tant que domaine personnalisé, vous pouvez interagir avec votre GraphQL et vos points de terminaison en temps réel à l'aide des URL suivantes :

AWS AppSync point de terminaison GraphQL de domaine personnalisé :

```
https://api.example.com/graphql
```

AWS AppSync point de terminaison en temps réel de domaine personnalisé :

```
wss://api.example.com/graphql/realtime
```

Format du paramètre d'en-tête basé sur le mode d'autorisation d’API AWS AppSync

Le format de l'`header`objet utilisé dans la chaîne de requête de connexion varie en fonction du mode d'autorisation de l'AWS AppSync API. Le champ `host` de l'objet fait référence au point de terminaison GraphQL AWS AppSync , qui est utilisé pour valider la connexion même si l'appel `wss://` est effectué par rapport au point de terminaison en temps réel. Pour initier la poignée de main et établir la connexion autorisée, `payload` doit être un objet JSON vide.

Clé API

En-tête de clé API

Contenu de l'en-tête

- `"host": <string>`: L'hôte du point de terminaison AWS AppSync GraphQL ou votre nom de domaine personnalisé.
- `"x-api-key": <string>`: clé d'API configurée pour l'AWS AppSync API.

Example (Exemple)

```
{
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com",
    "x-api-key":"da2-12345678901234567890123456"
}
```

Contenu de la charge utile

```
{}
```

URL de demande

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJob3N0IjoiZXhhbXBsZTEyMzQ1Njc4OTAwMDAuYXBwc3luYy1hcGkudXMtZWFzdC0xLmFtYXpvbmF3cy5jb20i
```

Groupes d'utilisateurs Amazon Cognito et OpenID Connect (OIDC)

Amazon Cognito et OIDCHeader

Contenu de l'en-tête :

- `"Authorization": <string>`: un jeton d'identification JWT. L'en-tête peut utiliser un schéma Bearer.
- `"host": <string>`: L'hôte du point de terminaison AWS AppSync GraphQL ou votre nom de domaine personnalisé.

Exemple :

```
{

 "Authorization":"eyEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJHWEFLSXBieU5WNHhsQjEXAMPLEnM2WldvPSIsImFsZ
zEE2DJH7sHOl2zxYi7f-SmEGoh2AD8emxQRYajByz-rE4Jh0QOymN2Ys-ZIkMpVBTPgu-
TMWDyOHhDUmUj2OP82yeZ3wlZAtr_gM4LzjXUXmI_K2yGjuXfXTaa1mvQEBG0mQfVd7SfwXB-
jcv4RYVi6j25qgow9Ew52ufurPqaK-3WAKG32KpV8J4-Wejq8t0c-
yA7sb8EnB551b7TU93uKRiVVK3E55Nk5ADPoam_WYE45i3s5qVAP_-InW75NUoOCGTsS8YWMfb6ecHYJ-1j-
bzA27zaT9VjctXn9byNFZmEXAMPLExw",
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

Contenu de la charge utile :

```
{}
```

URL de la demande :

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

IAM

En-tête IAM

Contenu de l'en-tête

- `"accept": "application/json, text/javascript"` : un paramètre `<string>` constant.
- `"content-encoding": "amz-1.0"` : un paramètre `<string>` constant.
- `"content-type": "application/json; charset=UTF-8"` : un paramètre `<string>` constant.
- `"host": <string>` : il s'agit de l'hôte du point de terminaison GraphQL AWS AppSync .
  - `"x-amz-date": <string>`: L'horodatage doit être en UTC et au format ISO 8601 suivant : YYYYMMDD'T'HHMMSS'Z'. Par exemple, 20150830T123600Z est un horodatage valide. N'incluez pas de millisecondes dans l'horodatage. Pour plus d'informations, consultez la section Gestion des dates dans la version 4 de Signature dans le Références générales AWS.
  - `"X-Amz-Security-Token": <string>`: le jeton de AWS session, qui est requis lors de l'utilisation d'informations d'identification de sécurité temporaires. Pour plus d'informations, consultez Utilisation d'informations d'identification temporaires avec des ressources AWS dans le Guide de l'utilisateur IAM.

- `"Authorization": <string>`: informations de signature de la version 4 (SigV4) pour le AWS AppSync point de terminaison. Pour plus d'informations sur le processus de signature, voir Tâche 4 : ajouter la signature à la requête HTTP dans le Références générales AWS.

La requête HTTP de signature SigV4 inclut une URL canonique, qui est le point de terminaison GraphQL AWS AppSync auquel `/connect` est ajouté. La AWS région du point de terminaison du service est la même région que celle dans laquelle vous utilisez l'AWS AppSync API, et le nom du service est « appsync ». La requête HTTP pour la signature est la suivante :

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql/
connect",
  data: "{}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

Example (Exemple)

```
{
  "accept": "application/json, text/javascript",
  "content-encoding": "amz-1.0",
  "content-type": "application/json; charset=UTF-8",
  "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
  "x-amz-date": "20200401T001010Z",
  "X-Amz-Security-Token":
 "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXRoZWFEXAMPLEcwRQIgAh97Cljq7wOPL8KsxP3YtDuyc/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKEn0zyUhBEXAMPLEjj//////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUdAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtwR+9zF7NaMMMse07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZOzlZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnGlSNVx7SMcfovkFDqQamm
+88y1OwwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVYlj9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
```

```
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEfwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCkcCGFnwBNFLmfnbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpgqJsPKt582caFKArIFIvO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+tOLnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOKtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtKOOweDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA==",
  "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
 Signature=83EXAMPLEbcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}
```

Contenu de la charge utile

```
{}
```

URL de demande

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyEXAMPLEHQiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoE
```

Pour signer la demande à l'aide d'un domaine personnalisé :

```
{
  url: "https://api.example.com/graphql/connect",
  data: "{}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

Example (Exemple)

```
{
  "accept": "application/json, text/javascript",
  "content-encoding": "amz-1.0",
  "content-type": "application/json; charset=UTF-8",
  "host": "api.example.com",
```

```
  "x-amz-date": "20200401T001010Z",
  "X-Amz-Security-Token":
 "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXRoZWFEXAMPLEcwRQIgAh97Cljq7wOPL8KsxP3YtDuyc/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKEn0zyUhBEXAMPLEjj/////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUdAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtwR+9zF7NaMMMse07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZOzlZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnGlSNVx7SMcfovkFDqQamm
+88y1OwwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVYlj9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEfwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCkcCGFnwBNFLmfnbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpgqJsPKt582caFKArIFIvO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+tOLnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOKtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtKOOweDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA==",
  "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
 Signature=83EXAMPLEbcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}
```

Contenu de la charge utile

```
{}
```

URL de demande

```
wss://api.example.com/graphql?
header=eyEXAMPLEHQiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoE
```

> **Note**
>
> Une WebSocket connexion peut comporter plusieurs abonnements (même avec différents modes d'authentification). Une façon de mettre cela en œuvre consiste à créer une WebSocket connexion pour le premier abonnement, puis à la fermer lorsque le dernier abonnement n'est pas enregistré. Vous pouvez optimiser cela en attendant quelques

secondes avant de fermer la WebSocket connexion, au cas où l'application serait abonnée immédiatement après le désenregistrement du dernier abonnement. Pour une application mobile, par exemple, lorsque vous passez d'un écran à un autre, lors du démontage, elle arrête un abonnement, et lors du montage de l'événement, elle lance un autre abonnement.

Autorisation Lambda

En-tête d'autorisation Lambda

Contenu de l'en-tête

- `"Authorization": <string>`: valeur transmise en tant que`authorizationToken`.
- `"host": <string>`: L'hôte du point de terminaison AWS AppSync GraphQL ou votre nom de domaine personnalisé.

Example (Exemple)

```
{

 "Authorization":"M0UzQzM1MkQtMkI0Ni00OTZCLUI1NkQtMUM0MTQ0QjVBRTczCkI1REEzRTIxLTk5NzItNDJENi1BQ
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

Contenu de la charge utile

```
{}
```

URL de demande

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

WebSocketFonctionnement en temps réel

Après avoir initié une WebSocket poignée de main réussie avec AWS AppSync, le client doit envoyer un message suivant pour se connecter AWS AppSync pour différentes opérations. Ces messages nécessitent les données suivantes :

- `type` : type de l'opération.

- `id`: identifiant unique pour l'abonnement. Nous vous recommandons d'utiliser un UUID à cet effet.
- `payload`: charge utile associée, en fonction du type d'opération.

Le `type` champ est le seul champ obligatoire ; les `payload` champs `id` et sont facultatifs.

Séquence des événements

Pour lancer, établir, enregistrer et traiter correctement la demande d'abonnement, le client doit suivre la séquence suivante :

1. Initialisation de la connexion (`connection_init`)
2. Accusé de réception de connexion (`connection_ack`)
3. Inscription à l'abonnement (`start`)
4. Accusé de réception d'abonnement (`start_ack`)
5. Traitement de l'abonnement (`data`)
6. Annulation de l'abonnement (`stop`)

Message d'initiation de connexion

Après une poignée de main réussie, le client doit envoyer le `connection_init` message pour commencer à communiquer avec le point de terminaison AWS AppSync en temps réel. Sans cette étape, tous les autres messages sont ignorés. Le message est une chaîne obtenue par conversion de l'objet JSON suivant :

```
{ "type": "connection_init" }
```

Message d'accusé de réception de connexion

Après avoir envoyé le message `connection_init`, le client doit attendre le message `connection_ack`. Tous les messages envoyés avant réception `connection_ack` sont ignorés. Le message doit se lire comme suit :

```
{
  "type": "connection_ack",
  "payload": {
    // Time in milliseconds waiting for ka message before the client should terminate
 the WebSocket connection
```

```
    "connectionTimeoutMs": 300000
  }
}
```

Message « keep-alive »

Outre le message d'accusé de réception de connexion, le client reçoit régulièrement des messages de maintien en vie. Si le client ne reçoit pas de message de maintien en vie pendant le délai d'expiration de la connexion, il doit fermer la connexion. AWS AppSync continue d'envoyer ces messages et de gérer les abonnements enregistrés jusqu'à ce que la connexion soit automatiquement interrompue (après 24 heures). Les messages Keep-Alive sont des battements de cœur et le client n'a pas besoin d'en accuser réception.

```
{ "type": "ka" }
```

Message d'inscription à l'abonnement

Une fois que le client a reçu un `connection_ack` message, il peut envoyer des messages d'enregistrement d'abonnement à AWS AppSync. Ce type de message est un objet JSON sous forme de chaîne qui contient les champs suivants :

- `"id": <string>`: ID de l'abonnement. Cet identifiant doit être unique pour chaque abonnement, sinon le serveur renvoie une erreur indiquant que l'identifiant d'abonnement est dupliqué.
- `"type": "start"` : un paramètre `<string>` constant.
- `"payload": <Object>`: un objet qui contient les informations relatives à l'abonnement.
  - `"data": <string>`: objet JSON stringifié qui contient une requête GraphQL et des variables.
    - `"query": <string>`: une opération GraphQL.
    - `"variables": <Object>`: objet contenant les variables de la requête.
  - `"extensions": <Object>`: objet contenant un objet d'autorisation.
- `"authorization": <Object>`: un objet qui contient les champs requis pour l'autorisation.

Objet d'autorisation pour l'enregistrement de l'abonnement

Les mêmes règles de la Format du paramètre d'en-tête basé sur le mode d'autorisation d'API AWS AppSync section s'appliquent à l'objet d'autorisation. La seule exception concerne l'IAM, où les informations de signature SigV4 sont légèrement différentes. Pour plus de détails, consultez l'exemple IAM.

Exemple d'utilisation de groupes d'utilisateurs Amazon Cognito :


```
{
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n
 __typename\\n message\\n }\\n }\",\"variables\":{}}",
      "extensions": {
        "authorization": {
          "Authorization":
 "eyEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJEXAMPLEBieU5WNHhsQjhPVW9YMnM2WldvPSIsImFsZyI6IlEXAMPLEn0.e
qTCtrYeboUJ4luRSTPXaNewNeEXAMPLE14C6sfg05tO0fOMpiUwj9k19gtNCCMqoSsjtQoUweFnH4JYa5EXAMPLEVxOyQEQ
RWvW7yQU3sNQRLEXAMPLEcd0yufBiCYs3dfQxTTdvR1B6Wz6CD78lfNeKqfzzUn2beMoup2h6EXAMPLE4ow8cUPUPvG0DzR
          "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
         }
      }
  },
  "type": "start"
}
```


Exemple d'utilisation d'IAM :


```
{
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n
 __typename\\n message\\n }\\n }\",\"variables\":{}}",
    "extensions": {
      "authorization": {
        "accept": "application/json, text/javascript",
        "content-type": "application/json; charset=UTF-8",
        "X-Amz-Security-Token":
 "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXRoZWFEXAMPLEcwRQIgAh97Cljq7wOPL8KsxP3YtDuyc/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKEn0zyUhBEXAMPLEjj/////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUdAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtwR+9zF7NaMMMse07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZOzlZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnGlSNVx7SMcfovkFDqQamm
+88y1OwwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVYlj9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
```

```
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEfwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCkcCGFnwBNFLmfnbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpgqJsPKt582caFKArIFIvO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+tOLnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOKtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtKOOweDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA==",
        "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
 Signature=b90131a61a7c4318e1c35ead5dbfdeb46339a7585bbdbeceeaff51f4022eb1fd",
        "content-encoding": "amz-1.0",
        "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
        "x-amz-date": "20200401T001010Z"
      }
    }
  },
  "type": "start"
}
```

Exemple d'utilisation d'un nom de domaine personnalisé :

```
{
  "id": "key-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n
 __typename\\n message\\n }\\n }\",\"variables\":{}}",
      "extensions": {
        "authorization": {
          "x-api-key": "da2-12345678901234567890123456",
          "host": "api.example.com"
         }
      }
  },
  "type": "start"
}
```

Il n'est pas nécessaire d'ajouter la signature SigV4 `/connect` à l'URL, et l'opération GraphQL en chaîne JSON la remplace. `data` Voici un exemple de demande de signature SigV4 :

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql",
```

```
  data: "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n __typename
\\n message\\n }\\n }\",\"variables\":{}}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

Message d'accusé de réception de l'abonnement

Après avoir envoyé le message de début d'abonnement, le client doit attendre AWS AppSync d'envoyer le `start_ack` message. Le `start_ack` message indique que l'abonnement est réussi.

Exemple de confirmation d'abonnement :

```
{
  "type": "start_ack",
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

Error message (Message d'erreur)

Si l'initialisation de la connexion ou l'enregistrement de l'abonnement échoue, ou si un abonnement est résilié depuis le serveur, le serveur envoie un message d'erreur au client :

- `"type": "error"` : un paramètre <`string`> constant.
- `"id": <string>`: L'identifiant de l'abonnement enregistré correspondant, le cas échéant.
- `"payload" <Object>`: objet contenant les informations d'erreur correspondantes.

Exemple :

```
{
  "type": "error",
  "payload": {
    "errors": [
      {
        "errorType": "LimitExceededError",
```

```
        "message": "Rate limit exceeded"
      }
    ]
  }
}
```

Traitement des messages de données

Lorsqu'un client soumet une mutation, AWS AppSync identifie tous les abonnés intéressés et envoie un `"type":"data"` message à chacun en utilisant l'abonnement `id` correspondant à l'opération `"start"` d'abonnement. Le client est censé suivre l'abonnement `id` qu'il envoie afin que, lorsqu'il reçoit un message de données, il puisse le faire correspondre à l'abonnement correspondant.

- `"type": "data"` : un paramètre <`string`> constant.
- `"id": <string>`: ID de l'abonnement enregistré correspondant.
- `"payload" <Object>`: objet contenant les informations d'abonnement.

Exemple :

```
{
  "type": "data",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": {
      "onCreateMessage": {
        "__typename": "Message",
        "message": "test"
      }
    }
  }
}
```

Message de désinscription de l'abonnement

Lorsque l'application souhaite arrêter d'écouter les événements d'abonnement, le client doit envoyer un message contenant l'objet JSON sous forme de chaîne suivant :

- `"type": "stop"` : un paramètre <`string`> constant.
- `"id": <string>`: ID de l'abonnement dont vous souhaitez annuler l'enregistrement.

Exemple :

```
{
  "type":"stop",
  "id":"ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

AWS AppSync renvoie un message de confirmation avec l'objet JSON stringifié suivant :

- `"type": "complete"` : un paramètre `<string>` constant.
- `"id": <string>`: ID de l'abonnement non enregistré.

Une fois que le client a reçu le message de confirmation, il ne reçoit plus aucun message pour cet abonnement en particulier.

Exemple :

```
{
  "type":"complete",
  "id":"eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

Déconnexion du WebSocket

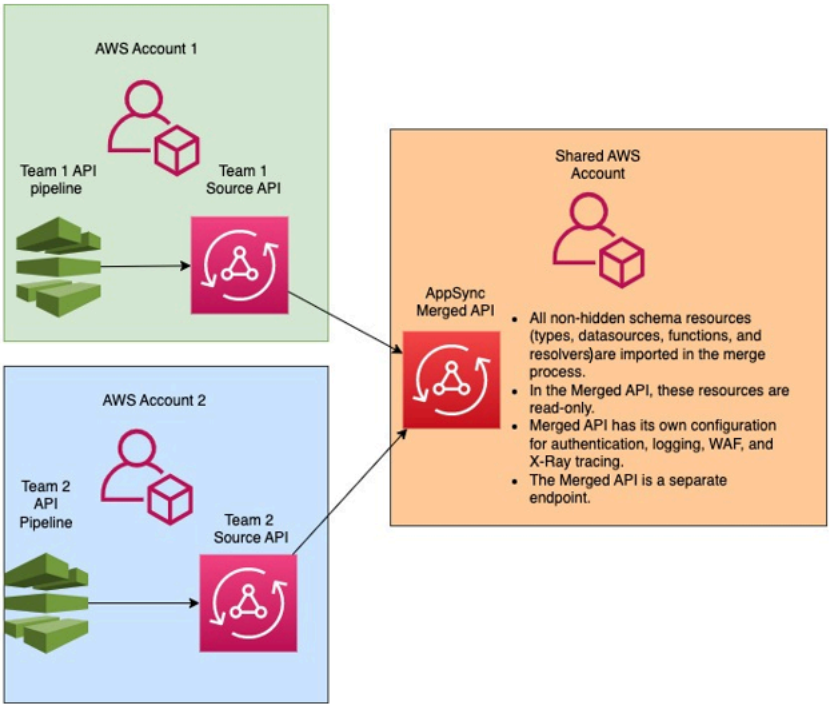
Avant de se déconnecter, afin d'éviter toute perte de données, le client doit disposer de la logique nécessaire pour vérifier qu'aucune opération n'est actuellement en cours via la WebSocket connexion. Tous les abonnements doivent être annulés avant de se déconnecter du. WebSocket

API fusionnées

Au fur et à mesure que l'utilisation de GraphQL se développe au sein d'une organisation, des compromis peuvent survenir entre l'API ease-of-use et la vitesse de développement des API. D'une part, les entreprises adoptent AWS AppSync GraphQL pour simplifier le développement d'applications en fournissant aux développeurs une API flexible qu'ils peuvent utiliser pour accéder, manipuler et combiner en toute sécurité les données d'un ou de plusieurs domaines de données en un seul appel réseau. D'autre part, les équipes d'une organisation responsables des différents domaines de données combinés en un seul point de terminaison d'API GraphQL peuvent souhaiter

pouvoir créer, gérer et déployer des mises à jour d'API indépendamment les unes des autres afin d'augmenter leurs vitesses de développement.

Pour résoudre cette tension, la fonctionnalité d'API AWS AppSync fusionnées permet aux équipes de différents domaines de données de créer et de déployer indépendamment des AWS AppSync API (par exemple, des schémas GraphQL, des résolveurs, des sources de données et des fonctions), qui peuvent ensuite être combinées en une seule API fusionnée. Cela permet aux entreprises de gérer une API interdomaines simple à utiliser, et permet aux différentes équipes qui contribuent à cette API de procéder rapidement et indépendamment à des mises à jour de l'API.



À l'aide des API fusionnées, les organisations peuvent importer les ressources de plusieurs AWS AppSync API sources indépendantes dans un seul point de terminaison d'API AWS AppSync fusionnées. Pour ce faire, vous AWS AppSync permet de créer une liste d'API AWS AppSync sources, puis de fusionner toutes les métadonnées associées aux API sources, y compris le schéma, les types, les sources de données, les résolveurs et les fonctions, dans une nouvelle API AWS AppSync fusionnée.

Lors des fusions, il est possible qu'un conflit de fusion se produise en raison d'incohérences dans le contenu des données de l'API source, telles que des conflits de dénomination de type lors de la combinaison de plusieurs schémas. Pour les cas d'utilisation simples où aucune définition des API source n'est en conflit, il n'est pas nécessaire de modifier les schémas de l'API source. L'API Merged

qui en résulte importe simplement tous les types, résolveurs, sources de données et fonctions à partir des AWS AppSync API source d'origine. Pour les cas d'utilisation complexes où des conflits surviennent, les utilisateurs/équipes devront résoudre les conflits par différents moyens. AWS AppSyncfournit aux utilisateurs plusieurs outils et exemples permettant de réduire les conflits de fusion.

Les fusions suivantes configurées dans AWS AppSync propageront les modifications apportées dans les API sources à l'API fusionnée associée.

API et fédération fusionnées

La communauté GraphQL propose de nombreuses solutions et modèles permettant de combiner des schémas GraphQL et de permettre la collaboration d'équipe via un graphe partagé. AWS AppSync Les API fusionnées adoptent une approche basée sur le temps de création pour la composition du schéma, dans laquelle les API sources sont combinées dans une API fusionnée distincte. Une autre approche consiste à superposer un routeur d'exécution sur plusieurs API ou sous-graphes sources. Dans cette approche, le routeur reçoit une demande, référence un schéma combiné qu'il gère sous forme de métadonnées, construit un plan de demande, puis distribue les éléments de la demande sur ses sous-graphiques/serveurs sous-jacents. Le tableau suivant compare l'approche de création de l'API AWS AppSync fusionnée aux approches d'exécution basées sur un routeur pour la composition de schémas GraphQL :

Feature	AppSync Merged API	Router-based solutions
Sub-graphs managed independently	Yes	Yes
Sub-graphs addressable independently	Yes	Yes
Automated schema compositi on	Yes	Yes
Automated conflict detection	Yes	Yes
Conflict resolution via schema directives	Yes	Yes
Supported sub-graph servers	AWS AppSync*	Varies

Network complexity	Single, merged API means no extra network hops.	Multi-layer architecture requires query planning and delegation, sub-query parsing and serialization/deserialization, and reference resolvers in sub-graphs to perform joins.
Observability support	Built-in monitoring, logging, and tracing. A single, Merged API server means simplified debugging.	Build-your-own observability across router and all associated sub-graph servers. Complex debugging across distributed system.
Authorization support	Built in support for multiple authorization modes.	Build-your-own authorization rules.
Cross account security	Built-in support for cross-AWS cloud account associations.	Build-your-own security model.
Subscriptions support	Yes	No

* Les API AWS AppSync fusionnées ne peuvent être associées qu'aux API AWS AppSync sources. Si vous avez besoin d'assistance pour la composition de schémas entre sous-graphes AWS AppSync et non AWS AppSync sous-graphes, vous pouvez connecter une ou plusieurs API AWS AppSync GraphQL et/ou Merged à une solution basée sur un routeur. Par exemple, consultez le blog de référence pour ajouter des AWS AppSync API en tant que sous-graphe à l'aide d'une architecture basée sur un routeur avec Apollo Federation v2 : Apollo GraphQL Federation with. AWS AppSync

Rubriques

- Limites de l'API fusionnée
- Création d'API fusionnées

Résolution des conflits liés aux API fusionnées

En cas de conflit de fusion, AWS AppSync fournit aux utilisateurs plusieurs outils et exemples pour les aider à résoudre le ou les problèmes.

Directives de schéma d'API fusionnées

AWS AppSynca introduit plusieurs directives GraphQL qui peuvent être utilisées pour réduire ou résoudre les conflits entre les API sources :

- @canonical : Cette directive définit la priorité des types/champs ayant des noms et des données similaires. Si deux API sources ou plus ont le même type ou champ GraphQL, l'une des API peut annoter leur type ou leur champ comme étant canonique, ce qui sera priorisé lors de la fusion. Les types/champs en conflit qui ne sont pas annotés avec cette directive dans d'autres API sources sont ignorés lors de la fusion.
- @hidden : Cette directive encapsule certains types/champs pour les supprimer du processus de fusion. Les équipes peuvent souhaiter supprimer ou masquer des types ou des opérations spécifiques dans l'API source afin que seuls les clients internes puissent accéder à des données typées spécifiques. Avec cette directive attachée, les types ou les champs ne sont pas fusionnés dans l'API Merged.
- @renamed : Cette directive modifie les noms des types/champs afin de réduire les conflits de dénomination. Dans certains cas, différentes API ont le même type ou le même nom de champ. Cependant, ils doivent tous être disponibles dans le schéma fusionné. Un moyen simple de les inclure tous dans l'API Merged consiste à renommer le champ en quelque chose de similaire mais différent.

Pour illustrer le schéma utilitaire fourni par les directives, considérez l'exemple suivant :

Dans cet exemple, supposons que nous voulons fusionner deux API sources. Deux schémas nous ont été fournis pour créer et récupérer des publications (par exemple, une section de commentaires ou des publications sur les réseaux sociaux). En supposant que les types et les champs sont très similaires, le risque de conflit est élevé lors d'une opération de fusion. Les extraits ci-dessous indiquent les types et les champs de chaque schéma.

Le premier fichier, appelé Source1.graphQL, est un schéma GraphQL qui permet à un utilisateur de créer à l'aide de la mutation. `Posts` `putPost` Chacune `Post` contient un titre et un identifiant. L'identifiant est utilisé pour faire référence aux `User` informations ou à celles de l'auteur (e-mail et adresse)`Message`, et à la charge utile (contenu). Le `User` type est annoté avec la balise @canonical.

```
# This snippet represents a file called Source1.graphql

type Mutation {
    putPost(id: ID!, title: String!): Post
}

type Post {
    id: ID!
    title: String!
}

type Message {
   id: ID!
   content: String
}

type User @canonical {
   id: ID!
   email: String!
   address: String!
}

type Query {
    singlePost(id: ID!): Post
    getMessage(id: ID!): Message
}
```

Le second fichier, appelé Source2.graphQL, est un schéma GraphQL qui fait des choses très similaires à Source1.graphQL. Notez toutefois que les champs de chaque type sont différents. Lors de la fusion de ces deux schémas, des conflits de fusion peuvent survenir en raison de ces différences.

Notez également que Source2.graphQL contient également plusieurs directives pour réduire ces conflits. Le `Post` type est annoté avec une balise @hidden pour se masquer pendant l'opération de fusion. Le `Message` type est annoté avec la balise @renamed pour modifier le nom du type `ChatMessage` en cas de conflit de dénomination avec un autre `Message` type.

```
# This snippet represents a file called Source2.graphql

type Post @hidden  {
    id: ID!
    title: String!
    internalSecret: String!
}

type Message @renamed(to: "ChatMessage") {
   id: ID!
   chatId: ID!
   from: User!
   to: User!
}

# Stub user so that we can link the canonical definition from Source1
type User {
   id: ID!
}

type Query {
    getPost(id: ID!): Post
    getMessage(id: ID!): Message @renamed(to: "getChatMessage")
}
```

Lorsque la fusion aura lieu, le résultat produira le `MergedSchema.graphql` fichier :

```
# This snippet represents a file called MergedSchema.graphql

type Mutation {
    putPost(id: ID!, title: String!): Post
}

# Post from Source2 was hidden so only uses the Source1 definition.
type Post {
    id: ID!
    title: String!
}

# Renamed from Message to resolve the conflict
type ChatMessage {
   id: ID!
   chatId: ID!
```

```
   from: User!
   to: User!
}

type Message {
   id: ID!
   content: String
}

# Canonical definition from Source1
type User {
   id: ID!
   email: String!
   address: String!
}

type Query {
    singlePost(id: ID!): Post
    getMessage(id: ID!): Message

    # Renamed from getMessage
    getChatMessage(id: ID!): ChatMessage
}
```

Plusieurs événements se sont produits lors de la fusion :

- Le *`User`* type de Source1.graphQL a été prioritaire par rapport à celui de Source2.graphQL en *`User`* raison de l'annotation @canonical.
- Le `Message` type de Source1.graphQL a été inclus dans la fusion. Cependant, le fichier `Message` from Source2.graphql présentait un conflit de dénomination. En raison de son annotation @renamed, il a également été inclus dans la fusion mais avec un autre nom`ChatMessage`.
- Le `Post` type de Source1.graphql a été inclus, mais pas le `Post` type de Source2.graphql. Normalement, il y aurait un conflit sur ce type, mais comme le `Post` type de Source2.graphQL comportait une annotation @hidden, ses données ont été masquées et n'ont pas été incluses dans la fusion. Cela n'a entraîné aucun conflit.
- Le `Query` type a été mis à jour pour inclure le contenu des deux fichiers. Cependant, une `GetMessage` requête a été renommée en `GetChatMessage` raison de la directive. Cela a résolu le conflit de dénomination entre les deux requêtes portant le même nom.

Il y a aussi le cas où aucune directive n'est ajoutée à un type en conflit. Ici, le type fusionné inclura l'union de tous les champs de toutes les définitions de source de ce type. Par exemple, considérez l'exemple suivant :

Ce schéma, appelé Source1.graphQL, permet de créer et de récupérer. `Posts` La configuration est similaire à celle de l'exemple précédent, mais avec moins d'informations.

```
# This snippet represents a file called Source1.graphql

type Mutation {
    putPost(id: ID!, title: String!): Post
}

type Post  {
    id: ID!
    title: String!
}

type Query {
    getPost(id: ID!): Post
}
```

Ce schéma, appelé Source2.graphQL, permet de créer et de récupérer `Reviews` (par exemple, des évaluations de films ou des critiques de restaurants). `Reviews`sont associés à `Post` la même valeur d'ID. Ensemble, ils contiennent le titre, l'ID de publication et le message de charge utile de l'article de critique complet.

Lors de la fusion, il y aura un conflit entre les deux `Post` types. Aucune annotation ne permettant de résoudre ce problème, le comportement par défaut consiste à effectuer une opération d'union sur les types en conflit.

```
# This snippet represents a file called Source2.graphql

type Mutation {
    putReview(id: ID!, postId: ID!, comment: String!): Review
}

type Post  {
    id: ID!
    reviews: [Review]
}
```

```
type Review {
   id: ID!
   postId: ID!
   comment: String!
}

type Query {
    getReview(id: ID!): Review
}
```

Lorsque la fusion aura lieu, le résultat produira le `MergedSchema.graphql` fichier :

```
# This snippet represents a file called MergedSchema.graphql

type Mutation {
    putReview(id: ID!, postId: ID!, comment: String!): Review
    putPost(id: ID!, title: String!): Post
}

type Post  {
    id: ID!
    title: String!
    reviews: [Review]
}

type Review {
   id: ID!
   postId: ID!
   comment: String!
}

type Query {
    getPost(id: ID!): Post
    getReview(id: ID!): Review
}
```

Plusieurs événements se sont produits lors de la fusion :

- Le `Mutation` type n'a rencontré aucun conflit et a été fusionné.
- Les champs `Post` de type ont été combinés via une opération syndicale. Remarquez comment l'union entre les deux a produit un single `idtitle`, un a et un single`reviews`.

- Le `Review` type n'a rencontré aucun conflit et a été fusionné.
- Le `Query` type n'a rencontré aucun conflit et a été fusionné.

Gestion des résolveurs sur des types partagés

Dans l'exemple ci-dessus, considérez le cas où Source1.graphQL a configuré un résolveur d'unités sur`Query.getPost`, qui utilise une source de données DynamoDB nommée. `PostDatasource` Ce résolveur renverra le `id` et `title` d'un `Post` type. Maintenant, considérez que Source2.graphQL a configuré un résolveur de pipeline sur lequel s'exécutent `Post.reviews` deux fonctions. `Function1`possède une source `None` de données attachée pour effectuer des contrôles d'autorisation personnalisés. `Function2`possède une source de données DynamoDB attachée pour interroger la table. `reviews`

```
query GetPostQuery {
    getPost(id: "1") {
        id,
        title,
        reviews
    }
}
```

Lorsque la requête ci-dessus est exécutée par un client sur le point de terminaison de l'API Merged, le AWS AppSync service exécute d'abord le résolveur d'unités pour `Query.getPost` from`Source1`, qui appelle `PostDatasource` et renvoie les données depuis DynamoDB. Ensuite, il exécute le résolveur de `Post.reviews` pipeline qui `Function1` exécute une logique d'autorisation personnalisée et `Function2` renvoie les avis en `$context.source` fonction de ce qui s'y `id` trouve. Le service traite la demande comme une seule exécution de GraphQL, et cette demande simple ne nécessitera qu'un seul jeton de demande.

Gestion des conflits entre résolveurs sur les types partagés

Prenons le cas suivant où nous implémentons également un résolveur `Query.getPost` afin de fournir plusieurs champs à la fois au-delà du résolveur de champs dans. `Source2` Source1.graphQL peut ressembler à ceci :

```
# This snippet represents a file called Source1.graphql

type Post  {
```

```
    id: ID!
    title: String!
    date: AWSDateTime!
}

type Query {
    getPost(id: ID!): Post
}
```

Source2.graphQL peut ressembler à ceci :

```
# This snippet represents a file called Source2.graphql

type Post  {
  id: ID!
  content: String!
  contentHash: String!
  author: String!
}

type Query {
    getPost(id: ID!): Post
}
```

Toute tentative de fusion de ces deux schémas générera une erreur de fusion car les API AWS AppSync fusionnées ne permettent pas d'associer plusieurs résolveurs de source au même champ. Afin de résoudre ce conflit, vous pouvez implémenter un modèle de résolution de champs qui obligerait Source2.graphQL à ajouter un type distinct qui définira les champs qu'il possède par rapport au type. `Post` Dans l'exemple suivant, nous ajoutons un type appelé`PostInfo`, qui contient les champs de contenu et d'auteur qui seront résolus par Source2.graphQL. Source1.graphql implémentera le résolveur attaché à`Query.getPost`, tandis que Source2.graphql associera désormais un résolveur pour garantir que toutes les données peuvent être récupérées avec `Post.postInfo` succès :

```
type Post  {
  id: ID!
  postInfo: PostInfo
}

type PostInfo {
   content: String!
```

```
    contentHash: String!
    author: String!
}

type Query {
    getPost(id: ID!): Post
}
```

Bien que la résolution d'un tel conflit nécessite de réécrire les schémas d'API source et, éventuellement, que les clients modifient leurs requêtes, l'avantage de cette approche est que la propriété des résolveurs fusionnés reste claire pour les équipes source.

Configuration des schémas

Deux parties sont chargées de configurer les schémas pour créer une API fusionnée :

- Propriétaires d'API fusionnés - Les propriétaires d'API fusionnés doivent configurer la logique d'autorisation de l'API fusionnée et les paramètres avancés tels que la journalisation, le suivi, la mise en cache et le support WAF.
- Propriétaires d'API source associés : les propriétaires d'API associés doivent configurer les schémas, les résolveurs et les sources de données qui constituent l'API fusionnée.

Comme le schéma de votre API fusionnée est créé à partir des schémas de vos API sources associées, il est en lecture seule. Cela signifie que les modifications du schéma doivent être initiées dans vos API sources. Dans la AWS AppSync console, vous pouvez basculer entre votre schéma fusionné et les schémas individuels des API sources incluses dans votre API fusionnée à l'aide de la liste déroulante située au-dessus de la fenêtre Schéma.

Configuration des modes d'autorisation

Plusieurs modes d'autorisation sont disponibles pour protéger votre API fusionnée. Pour en savoir plus sur les modes d'autorisation dansAWS AppSync, consultez la section Autorisation et authentification.

Les modes d'autorisation suivants peuvent être utilisés avec les API fusionnées :

- Clé API : stratégie d'autorisation la plus simple. Toutes les demandes doivent inclure une clé d'API sous l'en-tête de la `x-api-key` demande. Les clés API expirées sont conservées pendant 60 jours après la date d'expiration.

- AWSIdentity and Access Management (IAM) (IAM) : La stratégie d'autorisation AWS IAM autorise toutes les demandes signées sigv4.
- Groupes d'utilisateurs Amazon Cognito : autorisez vos utilisateurs via les groupes d'utilisateurs Amazon Cognito pour obtenir un contrôle plus précis.
- AWSAutorisateurs Lambda : fonction sans serveur qui vous permet d'authentifier et d'autoriser l'accès à votre AWS AppSync API à l'aide d'une logique personnalisée.
- OpenID Connect : ce type d'autorisation applique les jetons OpenID connect (OIDC) fournis par un service conforme à l'OIDC. Votre application peut tirer parti des utilisateurs et des privilèges définis par votre fournisseur OIDC pour le contrôle des accès.

Les modes d'autorisation d'une API fusionnée sont configurés par le propriétaire de l'API fusionnée. Au moment d'une opération de fusion, l'API fusionnée doit inclure le mode d'autorisation principal configuré sur une API source en tant que mode d'autorisation principal ou en tant que mode d'autorisation secondaire. Dans le cas contraire, elle sera incompatible et l'opération de fusion échouera en raison d'un conflit. Lorsque vous utilisez des directives multi-auth dans les API sources, le processus de fusion est capable de fusionner automatiquement ces directives dans le point de terminaison unifié. Dans le cas où le mode d'autorisation principal de l'API source ne correspond pas au mode d'autorisation principal de l'API Merged, il ajoutera automatiquement ces directives d'authentification afin de garantir que le mode d'autorisation pour les types de l'API source est cohérent.

Configuration des rôles d'exécution

Lorsque vous créez une API fusionnée, vous devez définir un rôle de service. Un rôle de AWS service est un rôle AWS Identity and Access Management (IAM) utilisé par les AWS services pour effectuer des tâches en votre nom.

Dans ce contexte, il est nécessaire que votre API Merged exécute des résolveurs qui accèdent aux données à partir des sources de données configurées dans vos API sources. Le rôle de service requis pour cela est le`mergedApiExecutionRole`, et il doit disposer d'un accès explicite pour exécuter des demandes sur les API sources incluses dans votre API fusionnée via l'autorisation `appsync:SourceGraphQL` IAM. Lors de l'exécution d'une requête GraphQL, le AWS AppSync service assumera ce rôle de service et autorisera le rôle à effectuer l'`appsync:SourceGraphQLaction`.

AWS AppSyncprend en charge l'autorisation ou le refus de cette autorisation sur des champs de haut niveau spécifiques de la demande, comme le fonctionnement du mode d'autorisation IAM pour les

API IAM. Pour non-top-level les champsAWS AppSync, vous devez définir l'autorisation sur l'ARN de l'API source lui-même. Afin de restreindre l'accès à des non-top-level champs spécifiques de l'API Merged, nous vous recommandons d'implémenter une logique personnalisée dans votre Lambda ou de masquer les champs de l'API source dans l'API Merged à l'aide de la directive @hidden. Si vous souhaitez autoriser le rôle à effectuer toutes les opérations de données au sein d'une API source, vous pouvez ajouter la politique ci-dessous. Notez que la première entrée de ressource permet d'accéder à tous les champs de niveau supérieur et que la seconde entrée couvre les résolveurs enfants qui autorisent la ressource API source elle-même :

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [ "appsync:SourceGraphQL"],
        "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId/*",
            "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId"]
    }]
}
```

Si vous souhaitez limiter l'accès à un champ de niveau supérieur spécifique, vous pouvez utiliser une politique comme celle-ci :

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [ "appsync:SourceGraphQL"],
        "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId/types/
Query/fields/<Field-1>",
            "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId"]
    }]
}
```

Vous pouvez également utiliser l'assistant de création d'API de AWS AppSync console pour générer un rôle de service permettant à votre API fusionnée d'accéder aux ressources configurées dans les API source qui se trouvent dans le même compte que votre API fusionnée. Si vos API sources ne figurent pas dans le même compte que votre API fusionnée, vous devez d'abord partager vos ressources à l'aide de AWS Resource Access Manager (AWS RAM).

Configuration des API fusionnées entre comptes à l'aide de AWS RAM

Lorsque vous créez une API fusionnée, vous pouvez éventuellement associer des API sources provenant d'autres comptes partagés via AWS Resource Access Manager (AWS RAM). AWS RAMvous permet de partager vos ressources en toute sécurité entre les AWS comptes, au sein de votre organisation ou de vos unités organisationnelles (UO), ainsi qu'avec les rôles et les utilisateurs IAM.

AWS AppSyncs'intègre afin de prendre AWS RAM en charge la configuration et l'accès aux API sources sur plusieurs comptes à partir d'une seule API fusionnée. AWS RAMvous permet de créer un partage de ressources, ou un conteneur de ressources et les ensembles d'autorisations qui seront partagés pour chacun d'entre eux. Vous pouvez ajouter AWS AppSync des API à un partage de ressources dansAWS RAM. Dans un partage de ressources, AWS AppSync fournit trois ensembles d'autorisations différents qui peuvent être associés à une AWS AppSync API en RAM :

1. `AWSRAMPermissionAppSyncSourceApiOperationAccess`: ensemble d'autorisations par défaut ajouté lors du partage d'une AWS AppSync API AWS RAM si aucune autre autorisation n'est spécifiée. Cet ensemble d'autorisations est utilisé pour partager une AWS AppSync API source avec le propriétaire d'une API fusionnée. Cet ensemble d'autorisations inclut l'autorisation pour `appsync:AssociateMergedGraphqlApi` l'API source ainsi que l'`appsync:SourceGraphQL`autorisation requise pour accéder aux ressources de l'API source lors de l'exécution.
2. `AWSRAMPermissionAppSyncMergedApiOperationAccess`: Cet ensemble d'autorisations doit être configuré lors du partage d'une API fusionnée avec le propriétaire de l'API source. Cet ensemble d'autorisations donnera à l'API source la possibilité de configurer l'API fusionnée, y compris la possibilité d'associer toutes les API source détenues par le principal cible à l'API fusionnée et de lire et de mettre à jour les associations d'API source de l'API fusionnée.
3. `AWSRAMPermissionAppSyncAllowSourceGraphQLAccess`: Cet ensemble d'autorisations permet `appsync:SourceGraphQL` d'utiliser l'autorisation avec une AWS AppSync API. Il est destiné à être utilisé pour partager une API source avec un propriétaire d'API fusionnée. Contrairement à l'ensemble d'autorisations par défaut pour l'accès aux opérations de l'API source, cet ensemble d'autorisations inclut uniquement l'autorisation d'exécution`appsync:SourceGraphQL`. Si un utilisateur choisit de partager l'accès aux opérations de l'API fusionnée avec un propriétaire de l'API source, il devra également partager cette autorisation de l'API source avec le propriétaire de l'API fusionnée afin de disposer d'un accès à l'exécution via le point de terminaison de l'API fusionnée.

AWS AppSyncprend également en charge les autorisations gérées par le client. Lorsque l'une des autorisations AWS gérées fournies ne fonctionne pas, vous pouvez créer votre propre autorisation gérée par le client. Les autorisations gérées par le client sont des autorisations gérées que vous créez et gérez en spécifiant précisément quelles actions peuvent être effectuées dans quelles conditions avec des ressources partagées. AWS RAM AWS AppSyncvous permet de choisir parmi les actions suivantes lors de la création de votre propre autorisation :

1. `appsync:AssociateSourceGraphqlApi`
2. `appsync:AssociateMergedGraphqlApi`
3. `appsync:GetSourceApiAssociation`
4. `appsync:UpdateSourceApiAssociation`
5. `appsync:StartSchemaMerge`
6. `appsync:ListTypesByAssociation`
7. `appsync:SourceGraphQL`

Une fois que vous avez correctement partagé une API source ou une API fusionnée AWS RAM et que, si nécessaire, l'invitation au partage de ressources a été acceptée, elle sera visible dans la AWS AppSync console lorsque vous créez ou mettez à jour les associations d'API source sur votre API fusionnée. Vous pouvez également répertorier toutes les AWS AppSync API qui ont été partagées AWS RAM avec votre compte, quelle que soit l'autorisation définie, en appelant l'`ListGraphqlApis`opération fournie par AWS AppSync et en utilisant le filtre du `OTHER_ACCOUNTS` propriétaire.

> **Note**
>
> Le partage via AWS RAM nécessite que l'AWS RAMappelant soit autorisé à effectuer l'`appsync:PutResourcePolicy`action sur toute API partagée.

Fusion

Gestion des fusions

Les API fusionnées sont destinées à favoriser la collaboration en équipe sur un point de AWS AppSync terminaison unifié. Les équipes peuvent développer indépendamment leurs propres API GraphQL sources isolées dans le backend, tandis que le AWS AppSync service gère l'intégration

des ressources dans le point de terminaison unique de l'API Merged afin de réduire les difficultés de collaboration et de réduire les délais de développement.

Fusions automatiques

Les API source associées à votre API AWS AppSync fusionnée peuvent être configurées pour être automatiquement fusionnées (fusion automatique) dans l'API fusionnée une fois que des modifications ont été apportées à l'API source. Cela garantit que les modifications apportées par l'API source sont toujours propagées au point de terminaison de l'API Merged en arrière-plan. Toute modification du schéma de l'API source sera mise à jour dans l'API fusionnée tant qu'elle n'introduit pas de conflit de fusion avec une définition existante dans l'API fusionnée. Si la mise à jour dans l'API source met à jour un résolveur, une source de données ou une fonction, la ressource importée sera également mise à jour. Lorsqu'un nouveau conflit ne peut pas être résolu automatiquement (résolution automatique), la mise à jour du schéma de l'API fusionnée est rejetée en raison d'un conflit non pris en charge lors de l'opération de fusion. Le message d'erreur est disponible dans la console pour chaque association d'API source dont le statut est`MERGE_FAILED`. Vous pouvez également inspecter le message d'erreur en appelant l'`GetSourceApiAssociation`opération pour une association d'API source donnée à l'aide du AWS SDK ou de la AWS CLI comme suit :

```
aws appsync get-source-api-association --merged-api-identifier <Merged API ARN> --
association-id <SourceApiAssociation id>
```

Cela produira un résultat au format suivant :

```
{
    "sourceApiAssociation": {
        "associationId": "<association id>",
        "associationArn": "<association arn>",
        "sourceApiId": "<source api id>",
        "sourceApiArn": "<source api arn>",
        "mergedApiArn": "<merged api arn>",
        "mergedApiId": "<merged api id>",
        "sourceApiAssociationConfig": {
            "mergeType": "MANUAL_MERGE"
        },
        "sourceApiAssociationStatus": "MERGE_FAILED",
        "sourceApiAssociationStatusDetail": "Unable to resolve conflict on object with
 name title: Merging is not supported for fields with different types."
    }
}
```

Fusions manuelles

Le paramètre par défaut pour une API source est une fusion manuelle. Pour fusionner les modifications apportées aux API sources depuis la dernière mise à jour de l'API fusionnée, le propriétaire de l'API source peut invoquer une fusion manuelle depuis la AWS AppSync console ou via l'`StartSchemaMerge`opération disponible dans le AWS SDK et la AWS CLI.

Support supplémentaire pour les API fusionnées

Configuration des abonnements

Contrairement aux approches basées sur les routeurs pour la composition de schémas GraphQL, les API Merged fournissent un support AWS AppSync intégré pour les abonnements GraphQL. Toutes les opérations d'abonnement définies dans vos API sources associées seront automatiquement fusionnées et fonctionneront dans votre API fusionnée sans modification. Pour en savoir plus sur la prise AWS AppSync en charge des abonnements via une WebSockets connexion sans serveur, consultez la section Données en temps réel.

Configuration de l'observabilité

AWS AppSyncLes API fusionnées fournissent des fonctionnalités intégrées de journalisation, de surveillance et de statistiques via Amazon CloudWatch. AWS AppSyncfournit également un support intégré pour le traçage via AWS X-Ray.

Configuration de domaines personnalisés

AWS AppSyncLes API fusionnées fournissent un support intégré pour l'utilisation de domaines personnalisés avec les points de terminaison GraphQL et en temps réel de votre API fusionnée.

Configuration de la mise en cache

AWS AppSyncLes API fusionnées fournissent un support intégré pour la mise en cache facultative des réponses au niveau de la demande et/ou au niveau du résolveur, ainsi que pour la compression des réponses. Pour en savoir plus, consultez la section Mise en cache et compression.

Configuration d'API privées

AWS AppSyncLes API fusionnées fournissent un support intégré pour les API privées qui limitent l'accès au GraphQL et aux points de terminaison en temps réel de votre API fusionnée au trafic provenant des points de terminaison VPC que vous pouvez configurer.

Configuration des règles de pare-feu

AWS AppSyncLes API fusionnées fournissent un support intégré qui vous permet de protéger vos API en définissant des règles de pare-feu pour les applications Web. AWS WAF

Configuration des journaux d'audit

AWS AppSyncLes API fusionnées fournissent un support intégré qui vous permet de configurer et de gérer les journaux d'audit. AWS CloudTrail

Limites de l'API fusionnée

Lorsque vous développez des API fusionnées, prenez note des règles suivantes :

1. Une API fusionnée ne peut pas être une API source pour une autre API fusionnée.
2. Une API source ne peut pas être associée à plusieurs API fusionnées.
3. La limite de taille par défaut pour un document de schéma d'API fusionnée est de 10 Mo.
4. Le nombre par défaut d'API sources pouvant être associées à une API fusionnée est de 10. Toutefois, vous pouvez demander une augmentation de limite si vous avez besoin de plus de 10 API sources dans votre API fusionnée.

Création d'API fusionnées

Pour créer une API fusionnée dans la console

1. Connectez-vous à AWS Management Console et ouvrez la console AWS AppSync.
   - Dans le tableau de bord, choisissez Create API.
2. Choisissez Merged API, puis Next.
3. Dans la page Spécifier les détails de l'API, entrez les informations suivantes :
   a. Dans Détails de l'API, entrez les informations suivantes :
      i. Spécifiez le nom de l'API de votre API fusionnée. Ce champ permet d'étiqueter votre API GraphQL afin de la distinguer facilement des autres API GraphQL.
      ii. Spécifiez les détails du contact. Ce champ est facultatif et associe un nom ou un groupe à l'API GraphQL. Il n'est pas lié ou généré par d'autres ressources et fonctionne de la même manière que le champ du nom de l'API.

b. Sous Rôle de service, vous devez associer un rôle d'exécution IAM à votre API fusionnée afin de AWS AppSync pouvoir importer et utiliser vos ressources en toute sécurité lors de l'exécution. Vous pouvez choisir de créer et d'utiliser un nouveau rôle de service, qui vous permettra de spécifier les politiques et les AWS AppSync ressources à utiliser. Vous pouvez également importer un rôle IAM existant en choisissant Utiliser un rôle de service existant, puis en sélectionnant le rôle dans la liste déroulante.

c. Dans Configuration de l'API privée, vous pouvez choisir d'activer les fonctionnalités de l'API privée. Notez que ce choix ne peut pas être modifié après la création de l'API fusionnée. Pour plus d'informations sur les API privées, consultez la section Utilisation AWS AppSync d'API privées.

Choisissez Next une fois que vous avez terminé.

4. Ensuite, vous devez ajouter les API GraphQL qui seront utilisées comme base pour votre API fusionnée. Sur la page Sélectionner les API sources, entrez les informations suivantes :

a. Dans le tableau des API de votre AWS compte, sélectionnez Ajouter des API source. Dans la liste des API GraphQL, chaque entrée contiendra les données suivantes :

i. Nom : champ de nom d'API de l'API GraphQL.

ii. ID d'API : valeur d'ID unique de l'API GraphQL.

iii. Mode d'authentification principal : mode d'autorisation par défaut pour l'API GraphQL. Pour plus d'informations sur les modes d'autorisation dansAWS AppSync, consultez la section Autorisation et authentification.

iv. Mode d'authentification supplémentaire : modes d'autorisation secondaires configurés dans l'API GraphQL.

v. Choisissez les API que vous utiliserez dans l'API fusionnée en cochant la case à côté du champ Nom de l'API. Ensuite, choisissez Add Source APIs. Les API GraphQL sélectionnées apparaîtront dans les API de votre tableau des AWS comptes.

b. Dans le tableau APIs from other AWS accounts, sélectionnez Add Source APIs. Les API GraphQL de cette liste proviennent d'autres comptes qui partagent leurs ressources avec les vôtres via AWS Resource Access Manager ()AWS RAM. Le processus de sélection des API GraphQL dans ce tableau est le même que celui de la section précédente. Pour plus d'informations sur le partage de ressources viaAWS RAM, voir Qu'est-ce que c'est AWS Resource Access Manager ? .

Choisissez Next une fois que vous avez terminé.

c. Ajoutez votre mode d'authentification principal. Voir Autorisation et authentification pour plus d'informations. Choisissez Suivant.

d. Passez en revue vos entrées, puis choisissez Create API.

Introspection RDS

AWS AppSyncfacilite la création d'API à partir de bases de données relationnelles existantes. Son utilitaire d'introspection permet de découvrir des modèles à partir de tables de base de données et de proposer des types GraphQL. L'assistant de création d'API de la AWS AppSync console peut générer instantanément une API à partir d'une base de données Aurora MySQL ou PostgreSQL. Il crée automatiquement des types et des JavaScript résolveurs pour lire et écrire des données.

AWS AppSyncfournit une intégration directe aux bases de données Amazon Aurora via l'API de données Amazon RDS. Plutôt que de nécessiter une connexion permanente à la base de données, l'API de données Amazon RDS propose un point de terminaison HTTP sécurisé qui AWS AppSync se connecte pour exécuter SQL des instructions. Vous pouvez l'utiliser pour créer une API de base de données relationnelle pour vos charges de travail MySQL et PostgreSQL sur Aurora.

La création d'une API pour votre base de données relationnelle AWS AppSync présente plusieurs avantages :

- Votre base de données n'est pas directement exposée aux clients, ce qui permet de dissocier le point d'accès de la base de données elle-même.
- Vous pouvez créer des API spécialement conçues pour répondre aux besoins des différentes applications, éliminant ainsi le besoin d'une logique métier personnalisée dans les interfaces. Cela correspond au modèle Backend-For-Frontend (BFF).
- L'autorisation et le contrôle d'accès peuvent être mis en œuvre au niveau de la AWS AppSync couche en utilisant différents modes d'autorisation pour contrôler l'accès. Aucune ressource de calcul supplémentaire n'est requise pour se connecter à la base de données, par exemple pour héberger un serveur Web ou établir des connexions par proxy.
- Des fonctionnalités en temps réel peuvent être ajoutées par le biais d'abonnements, les mutations de données AppSync étant automatiquement transmises aux clients connectés.
- Les clients peuvent se connecter à l'API via HTTPS à l'aide de ports courants tels que 443.

AWS AppSyncfacilite la création d'API à partir de bases de données relationnelles existantes. Son utilitaire d'introspection permet de découvrir des modèles à partir de tables de base de données et

de proposer des types GraphQL. L'assistant de création d'API de la AWS AppSync console peut générer instantanément une API à partir d'une base de données Aurora MySQL ou PostgreSQL. Il crée automatiquement des types et des JavaScript résolveurs pour lire et écrire des données.

AWS AppSyncfournit des JavaScript utilitaires intégrés pour simplifier l'écriture d'instructions SQL dans les résolveurs. Vous pouvez utiliser les modèles AWS AppSync de `sql` balises pour les instructions statiques avec des valeurs dynamiques, ou les utilitaires du `rds` module pour créer des instructions par programmation. Consultez la référence des fonctions de résolution pour les sources de données RDS et les modules intégrés pour en savoir plus.

Utilisation de la fonction d'introspection (console)

Pour un didacticiel détaillé et un guide de démarrage, voir Tutoriel : Aurora PostgreSQL Serverless with Data API.

La AWS AppSync console vous permet de créer une API AWS AppSync GraphQL à partir de votre base de données Aurora existante configurée avec l'API Data en quelques minutes seulement. Cela génère rapidement un schéma opérationnel basé sur la configuration de votre base de données. Vous pouvez utiliser l'API telle quelle ou vous appuyer sur celle-ci pour ajouter des fonctionnalités.

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
   - Dans le Tableau de bord, choisissez Créer une API.
2. Sous Options d'API, choisissez GraphQL APIs, Start with a Amazon Aurora cluster, puis Next.
   a. Entrez un nom d'API. Il sera utilisé comme identifiant pour l'API dans la console.
   b. Pour les coordonnées, vous pouvez saisir un point de contact afin d'identifier un responsable de l'API. Il s'agit d'un champ facultatif.
   c. Dans Configuration de l'API privée, vous pouvez activer les fonctionnalités de l'API privée. Une API privée n'est accessible qu'à partir d'un point de terminaison VPC configuré (VPCE). Pour plus d'informations, consultez la section API privées.

      Nous ne recommandons pas d'activer cette fonctionnalité pour cet exemple. Choisissez Next après avoir examiné vos entrées.
3. Sur la page Base de données, choisissez Sélectionner une base de données.
   a. Vous devez choisir votre base de données dans votre cluster. La première étape consiste à choisir la région dans laquelle se trouve votre cluster.

b. Choisissez le cluster Aurora dans la liste déroulante. Notez que vous devez avoir créé et activé une API de données correspondante avant d'utiliser la ressource.

c. Vous devez ensuite ajouter les informations d'identification de votre base de données au service. Cela se fait principalement à l'aide deAWS Secrets Manager. Choisissez la région dans laquelle se trouve votre secret. Pour plus d'informations sur la façon de récupérer des informations secrètes, voir Rechercher des secrets ou Extraire des secrets.

d. Ajoutez votre secret dans la liste déroulante. Notez que l'utilisateur doit disposer d'autorisations de lecture pour votre base de données.

4. Choisissez Import (Importer).

AWS AppSynccommencera à introspecter votre base de données, en découvrant les tables, les colonnes, les clés primaires et les index. Il vérifie que les tables découvertes peuvent être prises en charge dans une API GraphQL. Notez que pour prendre en charge la création de nouvelles lignes, les tables ont besoin d'une clé primaire, qui peut utiliser plusieurs colonnes. AWS AppSyncmappe les colonnes d'une table aux champs de type comme suit :

Type de données	Type de champ
VARCHAR	String
CHAR	String
BINARY	String
VARBINARY	String
TINYBLOB	String
TINYTEXT	String
TEXT	String
BLOB	String
MEDIUMTEXT	String
MEDIUMBLOB	String
LONGTEXT	String

LONGBLOB	String
BOOL	Boolean
BOOLEAN	Boolean
BIT	Int
TINYINT	Int
SMALLINT	Int
MEDIUMINT	Int
INT	Int
INTEGER	Int
BIGINT	Int
YEAR	Int
FLOAT	Float
DOUBLE	Float
DECIMAL	Float
DEC	Float
NUMERIC	Float
DATE	AWSDate
TIMESTAMP	String
DATETIME	String
TIME	AWSTime
JSON	AWSJson
ENUM	ENUM

5. Une fois la découverte des tables terminée, la section Base de données sera remplie avec vos informations. Dans la nouvelle section Tables de base de données, les données de la table sont peut-être déjà renseignées et converties en un type adapté à votre schéma. Si certaines des données requises ne s'affichent pas, vous pouvez les vérifier en choisissant Ajouter des tables, en cliquant sur les cases à cocher correspondant à ces types dans le modal qui apparaît, puis en choisissant Ajouter.

   Pour supprimer un type de la section des tables de base de données, cochez la case à côté du type que vous souhaitez supprimer, puis choisissez Supprimer. Les types supprimés seront placés dans le modal Ajouter des tables si vous souhaitez les ajouter à nouveau ultérieurement.

   Notez que les noms de table sont AWS AppSync utilisés comme noms de type, mais vous pouvez les renommer, par exemple en remplaçant le nom d'un tableau au pluriel, par exemple films, par le nom de type Movie. Pour renommer un type dans la section Tables de base de données, cochez la case du type que vous souhaitez renommer, puis cliquez sur l'icône en forme de crayon dans la colonne Nom du type.

   Pour prévisualiser le contenu du schéma en fonction de vos sélections, choisissez Aperçu du schéma. Notez que ce schéma ne peut pas être vide, vous devez donc convertir au moins une table en un type. En outre, la taille de ce schéma ne peut pas dépasser 1 Mo.

   - Sous Rôle de service, choisissez de créer un nouveau rôle de service spécifiquement pour cette importation ou d'utiliser un rôle existant.
6. Choisissez Suivant.
7. Choisissez ensuite de créer une API en lecture seule (requêtes uniquement) ou une API pour lire et écrire des données (avec requêtes et mutations). Ce dernier prend également en charge les abonnements en temps réel déclenchés par des mutations.
8. Choisissez Suivant.
9. Passez en revue vos choix, puis choisissez Create API. AWS AppSynccréera l'API et associera des résolveurs aux requêtes et aux mutations. L'API générée est pleinement opérationnelle et peut être étendue selon les besoins.

Utilisation de la fonctionnalité d'introspection (API)

Vous pouvez utiliser l'API d'`StartDataSourceIntrospection`introspection pour découvrir des modèles dans votre base de données par programmation. Pour plus de détails sur la commande, consultez la section Utilisation de l'`StartDataSourceIntrospection`API.

Pour l'utiliser`StartDataSourceIntrospection`, indiquez le nom de ressource Amazon (ARN) de votre cluster Aurora, le nom de la base de données et l'ARN AWS Secrets Manager secret. La commande lance le processus d'introspection. Vous pouvez récupérer les résultats à l'aide de la `GetDataSourceIntrospection` commande. Vous pouvez spécifier si la commande doit renvoyer la chaîne SDL (Storage Definition Language) pour les modèles découverts. Cela est utile pour générer une définition de schéma SDL directement à partir des modèles découverts.

Par exemple, si vous avez l'instruction DDL (Data Definition Language) suivante pour une `Todos` table simple :

```
create table if not exists public.todos
(
id serial constraint todos_pk primary key,
description text,
due timestamp,
"createdAt" timestamp default now()
);
```

Vous commencez l'introspection par ce qui suit.

```
aws appsync start-data-source-introspection \
  --rds-data-api-config resourceArn=<cluster-arn>,secretArn=<secret-
arn>,databaseName=database
```

Ensuite, utilisez la `GetDataSourceIntrospection` commande pour récupérer le résultat.

```
aws appsync get-data-source-introspection \
  --introspection-id a1234567-8910-abcd-efgh-identifier \
  --include-models-sdl
```

Cela renvoie le résultat suivant.

```
{
    "introspectionId": "a1234567-8910-abcd-efgh-identifier",
    "introspectionStatus": "SUCCESS",
```

```
    "introspectionStatusDetail": null,
    "introspectionResult": {
        "models": [
            {
                "name": "todos",
                "fields": [
                    {
                        "name": "description",
                        "type": {
                            "kind": "Scalar",
                            "name": "String",
                            "type": null,
                            "values": null
                        },
                        "length": 0
                    },
                    {
                        "name": "due",
                        "type": {
                            "kind": "Scalar",
                            "name": "AWSDateTime",
                            "type": null,
                            "values": null
                        },
                        "length": 0
                    },
                    {
                        "name": "id",
                        "type": {
                            "kind": "NonNull",
                            "name": null,
                            "type": {
                                "kind": "Scalar",
                                "name": "Int",
                                "type": null,
                                "values": null
                            },
                            "values": null
                        },
                        "length": 0
                    },
                    {
                        "name": "createdAt",
                        "type": {
```

```
                            "kind": "Scalar",
                            "name": "AWSDateTime",
                            "type": null,
                            "values": null
                        },
                        "length": 0
                    }
                ],
                "primaryKey": {
                    "name": "PRIMARY_KEY",
                    "fields": [
                        "id"
                    ]
                },
                "indexes": [],
                "sdl": "type todos\n{\ndescription: String\n\ndue: AWSDateTime\n\nid:
 Int!\n\ncreatedAt: AW
SDateTime\n}\n"
            }
        ],
        "nextToken": null
    }
}
```

Création d'une application client

Vous pouvez vous connecter à votre API AWS AppSync GraphQL à l'aide de n'importe quel client GraphQL, mais nous recommandons vivement le client Amplify. Amplify génère non seulement automatiquement des SDK clients fortement typés pour votre API GraphQL, mais offre également une prise en charge des données en temps réel et des fonctionnalités de requête GraphQL améliorées dans les applications clientes. Pour les applications Web, Amplify peut produire un JavaScript client. Pour ceux qui ciblent des environnements multiplateformes ou mobiles, Amplify s'adresse à Android, iOS et React Native. Pour en savoir plus sur la génération de code client pour ces plateformes, consultez la documentation Amplify. Voici un guide pour démarrer votre voyage avec une application JavaScript React :

> **Note**
>
> Vous devez installer et configurer à la fois npm et l'Amazon CLI avant de commencer. Si vous utilisez le client Amplify v6, suivez ce guide.

Mise en route :

1. Sur votre machine locale, accédez au répertoire de votre projet. Installez la bibliothèque Amplify à l'aide de la commande ci-dessous :

   ```
   npm install aws-amplify
   ```

2. Téléchargez votre fichier de configuration et placez-le dans le dossier de votre projet. Votre fichier de configuration contient généralement une `config` variable dont certains paramètres (point de terminaison, région, mode d'autorisation, etc.) sont définis. Par exemple, cela peut ressembler à ceci :

   ```
   const config =  {
       API: {
           GraphQL: {
             endpoint: 'https://abcdefghijklmnopqrstuvwxyz.appsync-api.us-
   west-2.amazonaws.com/graphql',
             region: 'us-west-2',
             defaultAuthMode: 'apiKey',
             apiKey: ''
           }
   ```

```
    }
};

export default config;
```

3. Dans votre code, importez la bibliothèque Amplify et votre configuration pour configurer Amplify :

```
import { Amplify } from 'aws-amplify';
import config from './aws-exports.js';

Amplify.configure(config);
```

   Vous pouvez également utiliser l'extrait de code dans la configuration de votre API pour configurer Amplify directement :

```
import { Amplify } from 'aws-amplify';

Amplify.configure({
  API: {
    GraphQL: {
      endpoint: 'https://abcdefghijklmnopqrstuvwxyz.appsync-api.us-
west-2.amazonaws.com/graphql',
      region: 'us-west-2',
      defaultAuthMode: 'apiKey',
      apiKey: ''
    }
  }
});
```

4. À l'aide de la chaîne d'outils Amplify, vous avez la possibilité de générer automatiquement des opérations en fonction de votre schéma, ce qui vous évite d'avoir à créer des scripts manuels. Dans le répertoire racine de votre application, utilisez la commande CLI suivante :

```
npx @aws-amplify/cli codegen add --apiId <id goes here> --region <region goes here>
```

   Cela téléchargera le schéma de votre API et, par défaut, générera du code d'assistance client dans le `src/graphql` dossier. Après chaque déploiement d'API, vous pouvez réexécuter la commande suivante pour générer des instructions et des types GraphQL mis à jour :

```
npx @aws-amplify/cli codegen
```

5. Vous pouvez désormais générer des modèles pour Android, Swift, Flutter et JavaScript DataStore. Utilisez la commande suivante pour télécharger votre schéma :

```
aws appsync get-introspection-schema --api-id <id goes here> --region <region goes
 here> --format SDL schema.graphql
```

Exécutez ensuite la commande suivante depuis le répertoire racine de votre application :

```
npx @aws-amplify/cli codegen models \
--model-schema schema.graphql \
--target [android|ios|flutter|javascript|typescript] \
--output-dir ./
```

Tutoriels Resolver () JavaScript

Les sources de données et les résolveurs AWS AppSync traduisent les requêtes GraphQL et récupèrent les informations de vos ressources. AWS AWS AppSync prend en charge le provisionnement automatique et les connexions avec certains types de sources de données. AWS AppSync prend en charge Amazon DynamoDBAWS Lambda, les bases de données relationnelles (Amazon Aurora Serverless), OpenSearch Amazon Service et les points de terminaison HTTP en tant que sources de données. Vous pouvez utiliser une API GraphQL avec vos AWS ressources existantes ou créer des sources de données et des résolveurs. Cette section vous guide à travers ce processus en une série de didacticiels pour mieux comprendre comment les détails fonctionnent et affiner les options.

Rubriques

Tutoriel : résolveurs DynamoDB JavaScript

Dans ce didacticiel, vous allez importer vos tables Amazon DynamoDB et les connecter AWS AppSync pour créer une API GraphQL entièrement fonctionnelle à l' JavaScript aide de résolveurs de pipeline que vous pouvez exploiter dans votre propre application.

Vous utiliserez la AWS AppSync console pour approvisionner vos ressources Amazon DynamoDB, créer vos résolveurs et les connecter à vos sources de données. Vous pourrez également lire et écrire dans votre base de données Amazon DynamoDB via des instructions GraphQL et vous abonner à des données en temps réel.

Certaines étapes spécifiques doivent être effectuées pour que les instructions GraphQL soient traduites en opérations Amazon DynamoDB et pour que les réponses soient retraduites dans GraphQL. Ce didacticiel explique le processus de configuration par le biais de plusieurs scénarios et modèles d'accès aux données concrets.

Création de votre API GraphQL

Pour créer une API GraphQL dans AWS AppSync

1. Ouvrez la AppSync console et choisissez Create API.
2. Sélectionnez Design from scratch et choisissez Next.
3. Donnez un nom à votre API`PostTutorialAPI`, puis choisissez Next. Passez à la page de révision tout en conservant les valeurs par défaut pour les autres options, puis choisissez`Create`.

La AWS AppSync console crée une nouvelle API GraphQL pour vous. Par défaut, il utilise le mode d'authentification par clé API. Vous pouvez utiliser la console pour configurer le reste de l'API GraphQL et exécuter des requêtes sur celle-ci jusqu'à la fin de ce didacticiel.

Définition d'une API de publication de base

Maintenant que vous disposez de votre API GraphQL, vous pouvez configurer un schéma de base qui permet la création, la récupération et la suppression de base des données de publication.

Pour ajouter des données à votre schéma

1. Dans votre API, choisissez l'onglet Schéma.
2. Nous allons créer un schéma qui définit un `Post` type et une opération `addPost` pour ajouter et obtenir `Post` des objets. Dans le volet Schéma, remplacez le contenu par le code suivant :

```
schema {
    query: Query
    mutation: Mutation
}

type Query {
    getPost(id: ID): Post
}

type Mutation {
```

```
    addPost(
        id: ID!
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}

type Post {
    id: ID!
    author: String
    title: String
    content: String
    url: String
    ups: Int!
    downs: Int!
    version: Int!
}
```

3. Choisissez Sauvegarder schéma.

Configuration de votre table Amazon DynamoDB

La AWS AppSync console peut vous aider à fournir les AWS ressources nécessaires pour stocker vos propres ressources dans une table Amazon DynamoDB. Au cours de cette étape, vous allez créer une table Amazon DynamoDB pour stocker vos publications. Vous allez également configurer un index secondaire que nous utiliserons ultérieurement.

Pour créer votre table Amazon DynamoDB

1. Sur la page Schéma, choisissez Create Resources.
2. Choisissez Utiliser le type existant, puis choisissez le `Post` type.
3. Dans la section Index supplémentaires, choisissez Ajouter un index.
4. Donnez un nom à l'index`author-index`.
5. Réglez le `Primary key` point sur `author` et la `Sort` touche sur`None`.
6. Désactivez Générer automatiquement GraphQL. Dans cet exemple, nous allons créer le résolveur nous-mêmes.
7. Sélectionnez Create (Créer).

Vous avez maintenant une nouvelle source de données appeléePostTable, que vous pouvez consulter en accédant à Sources de données dans l'onglet latéral. Vous utiliserez cette source de données pour lier vos requêtes et mutations à votre table Amazon DynamoDB.

Configuration d'un résolveur AddPost (Amazon DynamoDB) PutItem

Maintenant AWS AppSync que vous connaissez la table Amazon DynamoDB, vous pouvez la lier à des requêtes et à des mutations individuelles en définissant des résolveurs. Le premier résolveur que vous créez est le résolveur de addPost pipeline utilisé JavaScript, qui vous permet de créer une publication dans votre table Amazon DynamoDB. Un résolveur de pipeline comporte les composants suivants :

- L'emplacement dans le schéma GraphQL pour joindre le résolveur. Dans ce cas, vous configurez un résolveur sur le champ createPost sur le type Mutation. Ce résolveur sera invoqué lorsque l'appelant appellera une mutation. { addPost(...){...} }
- La source de données à utiliser pour ce résolveur. Dans ce cas, vous souhaitez utiliser la source de données DynamoDB que vous avez définie précédemment afin de pouvoir ajouter des entrées dans la post-table-for-tutorial table DynamoDB.
- Le gestionnaire de demandes. Le gestionnaire de demandes est une fonction qui gère la demande entrante de l'appelant et la traduit en instructions AWS AppSync à exécuter par rapport à DynamoDB.
- Le gestionnaire de réponses. Le rôle du gestionnaire de réponse est de gérer la réponse de DynamoDB et de la retraduire en une réponse attendue par GraphQL. Cela est utile si la forme des données dans DynamoDB est différente du type Post dans GraphQL, mais dans ce cas, ils ont la même forme, de sorte que vous transmettez simplement les données.

Pour configurer votre résolveur

1. Dans votre API, choisissez l'onglet Schéma.
2. Dans le volet Résolveurs, recherchez le addPost champ situé sous le Mutation type, puis choisissez Attacher.
3. Choisissez votre source de données, puis sélectionnez Créer.
4. Dans votre éditeur de code, remplacez le code par cet extrait :

```
import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'
```

```
export function request(ctx) {
 const item = { ...ctx.arguments, ups: 1, downs: 0, version: 1 }
 const key = { id: ctx.args.id ?? util.autoId() }
 return ddb.put({ key, item })
}

export function response(ctx) {
 return ctx.result
}
```

5. Choisissez Save (Enregistrer).

> **Note**
>
> Dans ce code, vous utilisez les utilitaires du module DynamoDB qui vous permettent de créer facilement des requêtes DynamoDB.

AWS AppSyncest livré avec un utilitaire de génération automatique d'identifiant appelé`util.autoId()`, qui est utilisé pour générer un identifiant pour votre nouveau message. Si vous ne spécifiez pas d'identifiant, l'utilitaire le générera automatiquement pour vous.

```
const key = { id: ctx.args.id ?? util.autoId() }
```

Pour plus d'informations sur les utilitaires disponibles pour JavaScript, consultez les fonctionnalités JavaScript d'exécution pour les résolveurs et les fonctions.

Appelez l'API pour ajouter une publication

Maintenant que le résolveur a été configuré, il AWS AppSync peut traduire une `addPost` mutation entrante en une opération Amazon DynamoDB`PutItem`. Vous pouvez désormais exécuter une mutation pour placer quelque chose dans la table.

Pour exécuter l'opération

1. Dans votre API, choisissez l'onglet Requêtes.
2. Dans le volet Requêtes, ajoutez la mutation suivante :

```
mutation addPost {
  addPost(
```

```
    id: 123,
    author: "AUTHORNAME"
    title: "Our first post!"
    content: "This is our first post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`addPost`. Les résultats de la publication nouvellement créée doivent apparaître dans le volet Résultats à droite du volet Requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

L'explication suivante montre ce qui s'est passé :

1. AWS AppSync a reçu une demande de mutation `addPost`.
2. AWS AppSyncexécute le gestionnaire de requêtes du résolveur. La `ddb.put` fonction crée une `PutItem` demande qui ressemble à ceci :

```
{
  operation: 'PutItem',
  key: { id: { S: '123' } },
  attributeValues: {
    downs: { N: 0 },
    author: { S: 'AUTHORNAME' },
    ups: { N: 1 },
    title: { S: 'Our first post!' },
    version: { N: 1 },
    content: { S: 'This is our first post.' },
    url: { S: 'https://aws.amazon.com/appsync/' }
  }
}
```

3. AWS AppSyncutilise cette valeur pour générer et exécuter une demande Amazon `PutItem` DynamoDB.
4. AWS AppSync a pris les résultats de la demande `PutItem` et les a reconvertis en types GraphQL.

```
{
    "id" : "123",
    "author": "AUTHORNAME",
    "title": "Our first post!",
    "content": "This is our first post.",
    "url": "https://aws.amazon.com/appsync/",
    "ups" : 1,
    "downs" : 0,
    "version" : 1
}
```

5. Le gestionnaire de réponses renvoie le résultat immédiatement (`return ctx.result`).
6. Le résultat final est visible dans la réponse GraphQL.

Configuration du résolveur GetPost (Amazon DynamoDB) GetItem

Maintenant que vous pouvez ajouter des données à la table Amazon DynamoDB, vous devez configurer `getPost` la requête afin qu'elle puisse extraire ces données de la table. Pour ce faire, vous configurez un autre résolveur.

Pour ajouter votre résolveur

1. Dans votre API, choisissez l'onglet Schéma.
2. Dans le volet Resolvers sur la droite, recherchez le `getPost` champ correspondant au `Query` type, puis choisissez Attacher.
3. Choisissez votre source de données, puis sélectionnez Créer.
4. Dans l'éditeur de code, remplacez le code par cet extrait :

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 return ddb.get({ key: { id: ctx.args.id } })
}

export const response = (ctx) => ctx.result
```

5. Enregistrez votre résolveur.

> **Note**
>
> Dans ce résolveur, nous utilisons une expression de fonction flèche pour le gestionnaire de réponses.

Appelez l'API pour obtenir un message

Maintenant que le résolveur est configuré, AWS AppSync il sait comment traduire une `getPost` requête entrante en une opération Amazon DynamoDB`GetItem`. Vous pouvez désormais exécuter une requête pour récupérer la publication que vous avez créée précédemment.

Pour exécuter votre requête

1. Dans votre API, choisissez l'onglet Requêtes.
2. Dans le volet Requêtes, ajoutez le code suivant et utilisez l'identifiant que vous avez copié après avoir créé votre publication :

```
query getPost {
  getPost(id: "123") {
    id
    author
    title
```

```
    content
    url
    ups
    downs
    version
  }
}
```

3. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`getPost`. Les résultats de la publication nouvellement créée doivent apparaître dans le volet Résultats à droite du volet Requêtes.
4. La publication extraite d'Amazon DynamoDB doit apparaître dans le volet Résultats à droite du volet Requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "getPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

Vous pouvez également prendre l'exemple suivant :

```
query getPost {
  getPost(id: "123") {
    id
    author
    title
  }
}
```

Si votre `getPost` requête n'a besoin que `duid`, et `authortitle`, vous pouvez modifier votre fonction de demande pour utiliser des expressions de projection afin de spécifier uniquement les

attributs que vous souhaitez voir apparaître dans votre table DynamoDB afin d'éviter tout transfert de données inutile de DynamoDB vers. AWS AppSync Par exemple, la fonction de requête peut ressembler à l'extrait ci-dessous :

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 return ddb.get({
  key: { id: ctx.args.id },
  projection: ['author', 'id', 'title'],
 })
}

export const response = (ctx) => ctx.result
```

Vous pouvez également utiliser un selectionSetListsigne avec `getPost` pour représenter `expression` :

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 const projection = ctx.info.selectionSetList.map((field) => field.replace('/', '.'))
 return ddb.get({ key: { id: ctx.args.id }, projection })
}

export const response = (ctx) => ctx.result
```

Création d'une mutation UpdatePost (Amazon DynamoDB) UpdateItem

Jusqu'à présent, vous pouvez créer et récupérer `Post` des objets dans Amazon DynamoDB. Vous allez ensuite configurer une nouvelle mutation pour mettre à jour un objet. Comparée à la `addPost` mutation qui nécessite que tous les champs soient spécifiés, cette mutation vous permet de ne spécifier que les champs que vous souhaitez modifier. Il a également introduit un nouvel `expectedVersion` argument qui vous permet de spécifier la version que vous souhaitez modifier. Vous allez définir une condition garantissant que vous modifiez la dernière version de l'objet. Pour ce faire, utilisez l'opération `UpdateItem` Amazon DynamoDB .sc

Pour mettre à jour votre résolveur

1. Dans votre API, choisissez l'onglet Schéma.

2. Dans le volet Schéma, modifiez le type `Mutation` pour ajouter une nouvelle mutation `updatePost` :

```
type Mutation {
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post

    addPost(
        id: ID
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}
```

3. Choisissez Sauvegarder schéma.

4. Dans le volet Résolveurs sur la droite, recherchez le `updatePost` champ nouvellement créé sur le `Mutation` type, puis choisissez Joindre. Créez votre nouveau résolveur à l'aide de l'extrait ci-dessous :

```
import { util } from '@aws-appsync/utils';
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { id, expectedVersion, ...rest } = ctx.args;
  const values = Object.entries(rest).reduce((obj, [key, value]) => {
    obj[key] = value ?? ddb.operations.remove();
    return obj;
  }, {});

  return ddb.update({
    key: { id },
    condition: { version: { eq: expectedVersion } },
    update: { ...values, version: ddb.operations.increment(1) },
  });
```

```
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type);
  }
  return result;
```

5. Enregistrez toutes les modifications que vous avez apportées.

Ce résolveur permet `ddb.update` de créer une demande Amazon DynamoDB`UpdateItem`. Au lieu de rédiger l'article dans son intégralité, vous demandez simplement à Amazon DynamoDB de mettre à jour certains attributs. Cela se fait à l'aide des expressions de mise à jour Amazon DynamoDB.

La `ddb.update` fonction prend une clé et un objet de mise à jour comme arguments. Ensuite, vous vérifiez les valeurs des arguments entrants. Lorsqu'une valeur est définie sur`null`, utilisez l'opération `remove` DynamoDB pour signaler que la valeur doit être supprimée de l'élément DynamoDB.

Il y a également une nouvelle `condition` section. Une expression de condition vous permet de dire AWS AppSync à Amazon DynamoDB si la demande doit aboutir ou non en fonction de l'état de l'objet déjà présent dans Amazon DynamoDB avant que l'opération ne soit effectuée. Dans ce cas, vous souhaitez que la `UpdateItem` demande aboutisse uniquement si le `version` champ de l'article actuellement dans Amazon DynamoDB correspond `expectedVersion` exactement à l'argument. Lorsque l'élément est mis à jour, nous voulons augmenter la valeur du`version`. C'est facile à faire avec la fonction d'opération`increment`.

Pour plus d'informations sur les expressions de condition, consultez la documentation sur les expressions de condition.

Pour plus d'informations sur la `UpdateItem` demande, consultez la UpdateItemdocumentation et celle du module DynamoDB.

Pour plus d'informations sur la façon d'écrire des expressions de mise à jour, consultez la documentation UpdateExpressionsDynamoDB.

Appelez l'API pour mettre à jour une publication

Essayons de mettre à jour l'`Post`objet avec le nouveau résolveur.

Pour mettre à jour votre objet

1. Dans votre API, choisissez l'onglet Requêtes.
2. Dans le volet Requêtes, ajoutez la mutation suivante. Vous devrez également mettre à jour l'`id`argument avec la valeur que vous avez notée précédemment :

```
mutation updatePost {
  updatePost(
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`updatePost`.
4. La publication mise à jour dans Amazon DynamoDB doit apparaître dans le volet Résultats à droite du volet Requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "updatePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 2
    }
  }
}
```

Dans cette demande, vous avez demandé à Amazon DynamoDB de mettre à jour uniquement `title` les `content` champs AWS AppSync et. Tous les autres champs ont été laissés de côté (à l'exception de l'incrémentation du `version` champ). Vous avez défini une nouvelle valeur pour `title` l'attribut et vous l'`content`avez supprimé de la publication. Les champs `author`, `url`, `ups` et `downs` sont restés inchangés. Réessayez d'exécuter la demande de mutation en la laissant exactement telle quelle. La réponse devrait être similaire à ce qui suit :

```
{
  "data": {
    "updatePost": null
  },
  "errors": [
    {
      "path": [
        "updatePost"
      ],
      "data": null,
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "The conditional request failed (Service: DynamoDb, Status Code: 400,
 Request ID: 1RR3QN5F35CS8IV5VR4OQO9NNBVV4KQNSO5AEMVJF66Q9ASUAAJG)"
    }
  ]
}
```

La demande échoue car l'expression de condition est évaluée à `false` :

1. La première fois que vous avez exécuté la demande, la valeur du `version` champ de la publication dans Amazon DynamoDB 1 était, ce qui correspondait à l'argument. `expectedVersion` La demande a abouti, ce qui signifie que le `version` champ a été incrémenté dans Amazon DynamoDB à. 2
2. La deuxième fois que vous avez exécuté la demande, la valeur du `version` champ de la publication dans Amazon DynamoDB 2 était, ce qui ne correspondait pas à l'argument. `expectedVersion`

Ce modèle est généralement appelé Verrouillage optimiste.

Création de mutations de vote (Amazon DynamoDB UpdateItem)

Le `Post` type contient `ups` des `downs` champs permettant l'enregistrement des votes positifs et négatifs. Cependant, pour le moment, l'API ne nous permet pas de faire quoi que ce soit avec eux. Ajoutons une mutation pour nous permettre de voter pour et contre les publications.

Pour ajouter votre mutation

1. Dans votre API, choisissez l'onglet Schéma.
2. Dans le volet Schéma, modifiez le `Mutation` type et ajoutez l'`DIRECTION`énumération pour ajouter de nouvelles mutations de vote :

```
type Mutation {
    vote(id: ID!, direction: DIRECTION!): Post
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        id: ID,
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post!
}

enum DIRECTION {
  UP
  DOWN
}
```

3. Choisissez Sauvegarder schéma.
4. Dans le volet Résolveurs sur la droite, recherchez le `vote` champ nouvellement créé sur le `Mutation` type, puis choisissez Joindre. Créez un nouveau résolveur en créant et en remplaçant le code par l'extrait suivant :

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const field = ctx.args.direction === 'UP' ? 'ups' : 'downs';
  return ddb.update({
    key: { id: ctx.args.id },
    update: {
      [field]: ddb.operations.increment(1),
      version: ddb.operations.increment(1),
    },
  });
}

export const response = (ctx) => ctx.result;
```

5. Enregistrez toutes les modifications que vous avez apportées.

Appelez l'API pour voter pour ou contre une publication

Maintenant que les nouveaux résolveurs ont été configurés, AWS AppSync il sait comment traduire une entrée `upvotePost` ou une `downvote` mutation en une opération Amazon DynamoDB`UpdateItem`. Vous pouvez désormais exécuter des mutations pour voter pour ou contre la publication que vous avez créée précédemment.

Pour exécuter votre mutation

1. Dans votre API, choisissez l'onglet Requêtes.
2. Dans le volet Requêtes, ajoutez la mutation suivante. Vous devrez également mettre à jour l'`id`argument avec la valeur que vous avez notée précédemment :

```
mutation votePost {
  vote(id:123, direction: UP) {
    id
    author
    title
    content
    url
    ups
    downs
    version
```

```
  }
}
```

3. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`votePost`.
4. La publication mise à jour dans Amazon DynamoDB doit apparaître dans le volet Résultats à droite du volet Requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "vote": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 0,
      "version": 4
    }
  }
}
```

5. Choisissez Exécuter quelques fois de plus. Vous devriez voir les `version` champs `ups` et augmenter à 1 chaque fois que vous exécutez la requête.
6. Modifiez la requête pour l'appeler avec un nom différent`DIRECTION`.

```
mutation votePost {
  vote(id:123, direction: DOWN) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

7. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`votePost`.

Cette fois, vous devriez voir les `version` champs `downs` et augmenter à 1 chaque fois que vous exécutez la requête.

Configuration d'un résolveur DeletePost (Amazon DynamoDB) DeleteItem

Ensuite, vous devez créer une mutation pour supprimer une publication. Pour ce faire, utilisez l'opération `DeleteItem` Amazon DynamoDB.

Pour ajouter votre mutation

1. Dans votre schéma, choisissez l'onglet Schéma.
2. Dans le volet Schéma, modifiez le `Mutation` type pour ajouter une nouvelle `deletePost` mutation :

```
type Mutation {
    deletePost(id: ID!, expectedVersion: Int): Post
    vote(id: ID!, direction: DIRECTION!): Post
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        id: ID
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post!
}
```

3. Cette fois, vous avez rendu le `expectedVersion` champ facultatif. Ensuite, choisissez Enregistrer le schéma.
4. Dans le volet Résolveurs sur la droite, recherchez le `delete` champ nouvellement créé dans le `Mutation` type, puis choisissez Joindre. Créez un nouveau résolveur à l'aide du code suivant :

```
import { util } from '@aws-appsync/utils'
```

```
import { util } from '@aws-appsync/utils';
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  let condition = null;
  if (ctx.args.expectedVersion) {
    condition = {
      or: [
        { id: { attributeExists: false } },
        { version: { eq: ctx.args.expectedVersion } },
      ],
    };
  }
  return ddb.remove({ key: { id: ctx.args.id }, condition });
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type);
  }
  return result;
}
```

> **Note**
>
> L'`expectedVersion`argument est facultatif. Si l'appelant définit un `expectedVersion` argument dans la demande, le gestionnaire de demandes ajoute une condition qui permet à la `DeleteItem` demande de réussir uniquement si l'article est déjà supprimé ou si l'`version`attribut de la publication dans Amazon DynamoDB correspond exactement au. `expectedVersion` En cas d'omission, aucune expression de condition n'est spécifiée dans la demande `DeleteItem`. Il réussit quelle que soit la valeur de l'élément `version` ou qu'il existe ou non dans Amazon DynamoDB.
>
> Même si vous supprimez un article, vous pouvez le renvoyer s'il ne l'a pas déjà été.

Pour plus d'informations sur la `DeleteItem` demande, consultez la DeleteItemdocumentation.

Appelez l'API pour supprimer une publication

Maintenant que le résolveur est configuré, AWS AppSync il sait comment traduire une `delete` mutation entrante en une opération Amazon DynamoDB`DeleteItem`. Vous pouvez désormais exécuter une mutation pour supprimer quelque chose dans la table.

Pour exécuter votre mutation

1. Dans votre API, choisissez l'onglet Requêtes.
2. Dans le volet Requêtes, ajoutez la mutation suivante. Vous devrez également mettre à jour l'`id`argument avec la valeur que vous avez notée précédemment :

```
mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`deletePost`.
4. La publication est supprimée d'Amazon DynamoDB. Notez que cela AWS AppSync renvoie la valeur de l'élément qui a été supprimé d'Amazon DynamoDB, qui doit apparaître dans le volet Résultats à droite du volet Requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 4,
      "version": 12
```

```
    }
  }
}
```

5. La valeur n'est renvoyée que si cet appel à `deletePost` est celui qui la supprime réellement d'Amazon DynamoDB. Choisissez à nouveau Exécuter.
6. L'appel réussit toujours, mais aucune valeur n'est renvoyée :

```
{
  "data": {
    "deletePost": null
  }
}
```

7. Essayons maintenant de supprimer un message, mais cette fois en spécifiant un`expectedValue`. Tout d'abord, vous devez créer une nouvelle publication, car vous venez de supprimer celle sur laquelle vous avez travaillé jusqu'à présent.
8. Dans le volet Requêtes, ajoutez la mutation suivante :

```
mutation addPost {
  addPost(
    id:123
    author: "AUTHORNAME"
    title: "Our second post!"
    content: "A new post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

9. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`addPost`.

10. Les résultats de la publication nouvellement créée doivent apparaître dans le volet Résultats à droite du volet Requêtes. Enregistrez le `id` nom de l'objet nouvellement créé car vous en aurez besoin dans un instant. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

11. Essayons maintenant de supprimer cette publication avec une valeur illégale pour ExpectedVersion. Dans le volet Requêtes, ajoutez la mutation suivante. Vous devrez également mettre à jour l'`id`argument avec la valeur que vous avez notée précédemment :

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

12. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`deletePost`. Le résultat suivant est renvoyé :

```
{
  "data": {
    "deletePost": null
  },
  "errors": [
    {
      "path": [
        "deletePost"
      ],
      "data": null,
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "The conditional request failed (Service: DynamoDb, Status Code:
 400, Request ID: 7083O037M1FTFRK038A4CI9H43VV4KQNSO5AEMVJF66Q9ASUAAJG)"
    }
  ]
}
```

13La demande a échoué car l'expression de condition est évaluée à`false`. La valeur `version` de la publication dans Amazon DynamoDB ne correspond pas à celle spécifiée dans `expectedValue` les arguments. La valeur actuelle de l'objet est renvoyée dans le champ `data` de la section `errors` de la réponse GraphQL. Réessayez la demande, mais corrigez `expectedVersion` :

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
```

```
    version
  }
}
```

14. Choisissez Exécuter (le bouton de lecture orange), puis choisissez `deletePost`.

    Cette fois, la demande aboutit et la valeur supprimée d'Amazon DynamoDB est renvoyée :

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

15. Choisissez à nouveau Exécuter. L'appel réussit toujours, mais cette fois aucune valeur n'est renvoyée car la publication a déjà été supprimée dans Amazon DynamoDB.

```
{ "data": { "deletePost": null } }
```

Configuration d'un résolveur AllPost (Amazon DynamoDB Scan)

Jusqu'à présent, l'API n'est utile que si vous connaissez le `id` message que vous souhaitez consulter. Ajoutons un nouveau résolveur qui renvoie toutes les publications de la table.

Pour ajouter votre mutation

1. Dans votre API, choisissez l'onglet Schéma.
2. Dans le volet Schéma, modifiez le type `Query` pour ajouter une nouvelle requête `allPost` :

```
type Query {
    allPost(limit: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
```

```
}
```

3. Ajouter un nouveau type `PaginationPosts` :

```
type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}
```

4. Choisissez Sauvegarder schéma.
5. Dans le volet Résolveurs sur la droite, recherchez le `allPost` champ nouvellement créé dans le `Query` type, puis choisissez Joindre. Créez un nouveau résolveur avec le code suivant :

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { limit = 20, nextToken } = ctx.arguments;
  return ddb.scan({ limit, nextToken });
}

export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

   Le gestionnaire de requêtes de ce résolveur attend deux arguments facultatifs :

   - `limit`- Spécifie le nombre maximum d'éléments à renvoyer en un seul appel.
   - `nextToken`- Utilisé pour récupérer la prochaine série de résultats (nous montrerons d'où `nextToken` vient la valeur pour plus tard).
6. Enregistrez toutes les modifications apportées à votre résolveur.

Pour plus d'informations sur la `Scan` demande, consultez la documentation de référence du scan.

Appelez l'API pour scanner tous les posts

Maintenant que le résolveur est configuré, AWS AppSync il sait comment traduire une `allPost` requête entrante en une opération Amazon DynamoDB`Scan`. Vous pouvez désormais analyser la table pour récupérer toutes les publications. Avant de pouvoir essayer, vous avez besoin de remplir la table avec certaines données, car vous avez supprimé tout ce que vous aviez utilisé jusque là.

Pour ajouter et interroger des données

1. Dans votre API, choisissez l'onglet Requêtes.
2. Dans le volet Requêtes, ajoutez la mutation suivante :

```
mutation addPost {
  post1: addPost(id:1 author: "AUTHORNAME" title: "A series of posts, Volume 1"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post2: addPost(id:2 author: "AUTHORNAME" title: "A series of posts, Volume 2"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post3: addPost(id:3 author: "AUTHORNAME" title: "A series of posts, Volume 3"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post4: addPost(id:4 author: "AUTHORNAME" title: "A series of posts, Volume 4"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post5: addPost(id:5 author: "AUTHORNAME" title: "A series of posts, Volume 5"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post6: addPost(id:6 author: "AUTHORNAME" title: "A series of posts, Volume 6"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post7: addPost(id:7 author: "AUTHORNAME" title: "A series of posts, Volume 7"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post8: addPost(id:8 author: "AUTHORNAME" title: "A series of posts, Volume 8"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post9: addPost(id:9 author: "AUTHORNAME" title: "A series of posts, Volume 9"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
}
```

3. Choisissez Exécuter (le bouton de lecture orange).
4. Maintenant, analysons la table, en renvoyant cinq résultats à la fois. Dans le volet Requêtes, ajoutez la requête suivante :

```
query allPost {
  allPost(limit: 5) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

5. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`allPost`.

Les cinq premiers articles doivent apparaître dans le volet Résultats à droite du volet Requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        },
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        }
      ],
      "nextToken": "<token>"
    }
  }
}
```

6. Vous avez reçu cinq résultats et un `nextToken` que vous pouvez utiliser pour obtenir la prochaine série de résultats. Mettez à jour la requête `allPost` pour qu'elle inclue l'élément `nextToken` issu de l'ensemble de résultats précédent :

```
query allPost {
  allPost(
    limit: 5
    nextToken: "<token>"
  ) {
```

```
    posts {
      id
      author
    }
    nextToken
  }
}
```

7. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`allPost`.

   Les quatre publications restantes devraient apparaître dans le volet Résultats à droite du volet Requêtes. Il n'y en a pas `nextToken` dans cette série de résultats parce que vous avez parcouru les neuf articles et qu'il n'en reste aucun. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        }
      ],
      "nextToken": null
    }
  }
}
```

Configuration d'un résolveur d' allPostsByauteur (requête Amazon DynamoDB)

Outre l'analyse de toutes les publications sur Amazon DynamoDB, vous pouvez également interroger Amazon DynamoDB pour récupérer les publications créées par un auteur spécifique. La table Amazon DynamoDB que vous avez créée précédemment possède déjà `GlobalSecondaryIndex` un `author-index` appel que vous pouvez utiliser avec une opération Amazon DynamoDB pour récupérer toutes les `Query` publications créées par un auteur spécifique.

Pour ajouter votre requête

1. Dans votre API, choisissez l'onglet Schéma.
2. Dans le volet Schéma, modifiez le type `Query` pour ajouter une nouvelle requête `allPostsByAuthor` :

   ```
   type Query {
       allPostsByAuthor(author: String!, limit: Int, nextToken: String): PaginatedPosts!
       allPost(limit: Int, nextToken: String): PaginatedPosts!
       getPost(id: ID): Post
   }
   ```

   Notez que cela utilise le même `PaginatedPosts` type que celui que vous avez utilisé pour la `allPost` requête.
3. Choisissez Sauvegarder schéma.
4. Dans le volet Résolveurs sur la droite, recherchez le `allPostsByAuthor` champ nouvellement créé sur le `Query` type, puis choisissez Joindre. Créez un résolveur à l'aide de l'extrait ci-dessous :

   ```
   import * as ddb from '@aws-appsync/utils/dynamodb';

   export function request(ctx) {
     const { limit = 20, nextToken, author } = ctx.arguments;
     return ddb.query({
       index: 'author-index',
       query: { author: { eq: author } },
       limit,
       nextToken,
     });
   }
   ```

```
export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

Comme le `allPost` résolveur, ce résolveur possède deux arguments facultatifs :

- `limit`- Spécifie le nombre maximum d'éléments à renvoyer en un seul appel.
- `nextToken`- Récupère le prochain ensemble de résultats (la valeur de `nextToken` peut être obtenue à partir d'un appel précédent).

5. Enregistrez toutes les modifications apportées à votre résolveur.

Pour plus d'informations sur la `Query` demande, consultez la documentation de référence sur les requêtes.

Appelez l'API pour interroger tous les articles par auteur

Maintenant que le résolveur a été configuré, AWS AppSync il sait comment traduire une `allPostsByAuthor` mutation entrante en une opération `Query` DynamoDB par rapport à l'index. `author-index` Vous pouvez désormais interroger la table pour récupérer toutes les publications d'un auteur spécifique.

Avant cela, cependant, remplissons le tableau avec quelques articles supplémentaires, car tous les articles ont jusqu'à présent le même auteur.

Pour ajouter des données et effectuer une requête

1. Dans votre API, choisissez l'onglet Requêtes.
2. Dans le volet Requêtes, ajoutez la mutation suivante :

```
mutation addPost {
  post1: addPost(id:10 author: "Nadia" title: "The cutest dog in the world" content:
 "So cute. So very, very cute." url: "https://aws.amazon.com/appsync/" ) { author,
 title }
  post2: addPost(id:11 author: "Nadia" title: "Did you know...?" content: "AppSync
 works offline?" url: "https://aws.amazon.com/appsync/" ) { author, title }
  post3: addPost(id:12 author: "Steve" title: "I like GraphQL" content: "It's great"
 url: "https://aws.amazon.com/appsync/" ) { author, title }
}
```

3. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`addPost`.

4. Maintenant, interrogez la table et renvoyez toutes les publications créées par `Nadia`. Dans le volet Requêtes, ajoutez la requête suivante :

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

5. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`allPostsByAuthor`. Tous les articles rédigés par `Nadia` doivent apparaître dans le volet Résultats à droite du volet Requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

6. La pagination fonctionne pour `Query` tout comme elle le fait pour `Scan`. Par exemple, examinons toutes les publications créées par `AUTHORNAME`, et prenons-en cinq à la fois.
7. Dans le volet Requêtes, ajoutez la requête suivante :

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
```

```
    limit: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

8. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`allPostsByAuthor`. Tous les articles rédigés par AUTHORNAME doivent apparaître dans le volet Résultats à droite du volet Requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        }
      ],
      "nextToken": "<token>"
    }
  }
}
```

9. Mettez à jour l'argument `nextToken` avec la valeur renvoyée par la requête précédente :

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

10. Choisissez Exécuter (le bouton de lecture orange), puis choisissez `allPostsByAuthor`. Les autres publications rédigées par `AUTHORNAME` doivent apparaître dans le volet Résultats à droite du volet Requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        },
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        }
      ],
      "nextToken": null
    }
```

```
  }
}
```

Utiliser des ensembles

Jusqu'à présent, le `Post` type était un objet clé/valeur plat. Vous pouvez également modéliser des objets complexes avec votre résolveur, tels que des ensembles, des listes et des cartes. Mettons à jour notre type `Post` pour inclure des balises. Une publication peut comporter zéro ou plusieurs balises, qui sont stockées dans DynamoDB sous forme de jeu de chaînes. Vous allez également configurer certaines mutations pour ajouter et supprimer des balises, et une nouvelle requête pour rechercher des publications avec une balise spécifique.

Pour configurer vos données

1. Dans votre API, choisissez l'onglet Schéma.
2. Dans le volet Schéma, modifiez le type `Post` pour ajouter un nouveau champ `tags` :

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
}
```

3. Dans le volet Schéma, modifiez le type `Query` pour ajouter une nouvelle requête `allPostsByTag` :

```
type Query {
  allPostsByTag(tag: String!, limit: Int, nextToken: String): PaginatedPosts!
  allPostsByAuthor(author: String!, limit: Int, nextToken: String): PaginatedPosts!
  allPost(limit: Int, nextToken: String): PaginatedPosts!
  getPost(id: ID): Post
}
```

4. Dans le volet Schéma, modifiez le type `Mutation` pour ajouter de nouvelles mutations `addTag` et `removeTag` :

```
type Mutation {
  addTag(id: ID!, tag: String!): Post
  removeTag(id: ID!, tag: String!): Post
  deletePost(id: ID!, expectedVersion: Int): Post
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}
```

5. Choisissez Sauvegarder schéma.
6. Dans le volet Résolveurs sur la droite, recherchez le `allPostsByTag` champ nouvellement créé sur le `Query` type, puis choisissez Joindre. Créez votre résolveur à l'aide de l'extrait ci-dessous :

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { limit = 20, nextToken, tag } = ctx.arguments;
  return ddb.scan({ limit, nextToken, filter: { tags: { contains: tag } } });
}

export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

7. Enregistrez toutes les modifications que vous avez apportées à votre résolveur.

8. Maintenant, faites de même pour le `Mutation` champ `addTag` en utilisant l'extrait ci-dessous :

> **Note**
>
> Bien que les utilitaires DynamoDB ne prennent actuellement pas en charge les opérations relatives aux ensembles, vous pouvez toujours interagir avec les ensembles en créant vous-même la requête.

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { id, tag } = ctx.arguments
 const expressionValues = util.dynamodb.toMapValues({ ':plusOne': 1 })
 expressionValues[':tags'] = util.dynamodb.toStringSet([tag])

 return {
  operation: 'UpdateItem',
  key: util.dynamodb.toMapValues({ id }),
  update: {
   expression: `ADD tags :tags, version :plusOne`,
   expressionValues,
  },
 }
}

export const response = (ctx) => ctx.result
```

9. Enregistrez toutes les modifications apportées à votre résolveur.
10. Répétez cette opération une fois de plus pour le `Mutation` champ `removeTag` à l'aide de l'extrait ci-dessous :

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
   const { id, tag } = ctx.arguments;
   const expressionValues = util.dynamodb.toMapValues({ ':plusOne': 1 });
   expressionValues[':tags'] = util.dynamodb.toStringSet([tag]);

   return {
     operation: 'UpdateItem',
```

```
    key: util.dynamodb.toMapValues({ id }),
    update: {
      expression: `DELETE tags :tags ADD version :plusOne`,
      expressionValues,
    },
  };
}

export const response = (ctx) => ctx.resultexport
```

11. Enregistrez toutes les modifications apportées à votre résolveur.

Appel de l'API pour utiliser des balises

Maintenant que vous avez configuré les résolveurs, vous savez comment AWS AppSync traduire les requêtes entrantes et `addTag` les `removeTag` `allPostsByTag` requêtes en `UpdateItem` DynamoDB `Scan` et en opérations. Pour tester cela, sélectionnons l'une des publications que nous avons créées précédemment. Par exemple, utilisons un article créé par `Nadia`.

Pour utiliser des balises

1. Dans votre API, choisissez l'onglet Requêtes.
2. Dans le volet Requêtes, ajoutez la requête suivante :

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

3. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`allPostsByAuthor`.
4. Toutes les publications de Nadia doivent apparaître dans le volet Résultats à droite du volet Requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
```

```
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

5. Utilisons celui qui s'intitule Le chien le plus mignon du monde. Enregistrez-le `id` car vous l'utiliserez plus tard. Essayons maintenant d'ajouter un `dog` tag.
6. Dans le volet Requêtes, ajoutez la mutation suivante. Vous devrez également mettre à jour l'argument `id` pour qu'il ait la valeur que vous avez notée précédemment.

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

7. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`addTag`. Le message est mis à jour avec le nouveau tag :

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
```

```
  }
}
```

8. Vous pouvez ajouter d'autres tags. Mettez à jour la mutation pour remplacer l'`tag`argument par `puppy` :

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

9. Choisissez Exécuter (le bouton de lecture orange), puis choisissez`addTag`. Le message est mis à jour avec le nouveau tag :

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

10. Vous pouvez également supprimer des balises. Dans le volet Requêtes, ajoutez la mutation suivante. Vous devrez également mettre à jour l'`id`argument avec la valeur que vous avez notée précédemment :

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

11. Choisissez Exécuter (le bouton de lecture orange), puis choisissez `removeTag`. La publication est mise à jour et la balise `puppy` est supprimée.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

12. Vous pouvez également rechercher tous les articles comportant un tag. Dans le volet Requêtes, ajoutez la requête suivante :

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

13. Choisissez Exécuter (le bouton de lecture orange), puis choisissez `allPostsByTag`. Toutes les publications qui ont la balise dog sont renvoyées :

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
            "dog",
            "puppy"
```

```
          ]
        }
      ],
      "nextToken": null
    }
  }
}
```

Conclusion

Dans ce didacticiel, vous avez créé une API qui vous permet de manipuler des `Post` objets dans DynamoDB à l'aide de AWS AppSync GraphQL.

Pour nettoyer, vous pouvez supprimer l'API AWS AppSync GraphQL de la console.

Pour supprimer le rôle associé à votre table DynamoDB, sélectionnez votre source de données dans la table Sources de données et cliquez sur Modifier. Notez la valeur du rôle sous Créer ou utiliser un rôle existant. Accédez à la console IAM pour supprimer le rôle.

Pour supprimer votre table DynamoDB, cliquez sur le nom de la table dans la liste des sources de données. Cela vous amène à la console DynamoDB où vous pouvez supprimer la table.

Tutoriel : résolveurs Lambda

Vous pouvez utiliserAWS LambdaavecAWS AppSyncpour résoudre n'importe quel champ GraphQL. Par exemple, une requête GraphQL peut envoyer un appel à une instance Amazon Relational Database Service (Amazon RDS), et une mutation GraphQL peut écrire dans un flux Amazon Kinesis. Dans cette section, nous allons vous montrer comment écrire une fonction Lambda qui exécute une logique métier basée sur l'invocation d'une opération de terrain GraphQL.

Création d'une fonction Lambda

L'exemple suivant montre une fonction Lambda écrite en`Node.js`(runtime : Node.js 18.x) qui effectue différentes opérations sur les articles de blog dans le cadre d'une application de publication de blog. Notez que le code doit être enregistré dans un nom de fichier portant l'extension .mis.

```
export const handler = async (event) => {
console.log('Received event {}', JSON.stringify(event, 3))

  const posts = {
```

```
1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
 content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
 AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
 content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
 '10', },
    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
 ups: null, downs: null },
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }

  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
case 'getPost':
      return posts[event.arguments.id]
    case 'allPosts':
      return Object.values(posts)
    case 'addPost':
      // return the arguments back
return event.arguments
    case 'addPostErrorWithData':
      result = posts[event.arguments.id]
      // attached additional error information to the post
      result.errorMessage = 'Error with the mutation, data has changed'
      result.errorType = 'MUTATION_ERROR'
return result
    case 'relatedPosts':
      return relatedPosts[event.source.id]
    default:
```

```
      throw new Error('Unknown field, unable to resolve ' + event.field)
  }
}
```

Cette fonction Lambda récupère une publication par identifiant, ajoute une publication, récupère une liste de publications et récupère les publications associées à une publication donnée.

> **Note**
>
> La fonction Lambda utilise le`switch`déclaration sur`event.field`pour déterminer quel champ est actuellement en cours de résolution.

Créez cette fonction Lambda à l'aide duAWSConsole de gestion.

Configuration d'une source de données pour Lambda

Après avoir créé la fonction Lambda, accédez à votre API GraphQL dans leAWS AppSyncconsole, puis choisissez leSources de donnéesonglet.

ChoisissezCréation d'une source de données, entrez un message amicalNom de la source de données(par exemple,**`Lambda`**), puis pourType de source de données, choisissezAWS Lambdafonction. PourRégion, choisissez la même région que votre fonction. PourFonction ARN, choisissez le nom de ressource Amazon (ARN) de votre fonction Lambda.

Après avoir choisi votre fonction Lambda, vous pouvez soit créer une nouvelleAWS Identity and Access Managementrôle (IAM) (pour lequelAWS AppSyncattribue les autorisations appropriées) ou choisissez un rôle existant dont la politique intégrée est la suivante :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/LAMBDA_FUNCTION"
        }
    ]
```

```
}
```

Vous devez également établir une relation de confiance avecAWS AppSyncpour le rôle IAM comme suit :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "appsync.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

Création d'un schéma GraphQL

Maintenant que la source de données est connectée à votre fonction Lambda, créez un schéma GraphQL.

Depuis l'éditeur de schéma dans leAWS AppSyncconsole, assurez-vous que votre schéma correspond au schéma suivant :

```
schema {
    query: Query
    mutation: Mutation
}
type Query {
    getPost(id:ID!): Post
    allPosts: [Post]
}
type Mutation {
    addPost(id: ID!, author: String!, title: String, content: String, url: String):
 Post!
}
type Post {
    id: ID!
    author: String!
    title: String
```

```
    content: String
    url: String
    ups: Int
    downs: Int
    relatedPosts: [Post]
}
```

Configuration des résolveurs

Maintenant que vous avez enregistré une source de données Lambda et un schéma GraphQL valide, vous pouvez connecter vos champs GraphQL à votre source de données Lambda à l'aide de résolveurs.

Vous allez créer un résolveur qui utilise leAWS AppSync JavaScript(`APPSYNC_JS`) exécutent et interagissent avec vos fonctions Lambda. Pour en savoir plus sur l'écritureAWS AppSyncrésolveurs et fonctions avecJavaScript, voirJavaScriptfonctionnalités d'exécution pour les résolveurs et les fonctions.

Pour plus d'informations sur les modèles de mappage Lambda, voirJavaScriptréférence de fonction de résolution pour Lambda.

Au cours de cette étape, vous devez associer un résolveur à la fonction Lambda pour les champs suivants :`getPost(id:ID!): Post`,`allPosts: [Post]`,`addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`, et`Post.relatedPosts: [Post]`. À partir duSchémaéditeur dans leAWS AppSyncconsole, dans leRésolveursvolet, choisissezJoindreà côté du`getPost(id:ID!): Post`champ. Choisissez votre source de données Lambda. Entrez ensuite le code suivant :

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  return ctx.result;
}
```

Ce code de résolution transmet le nom du champ, la liste des arguments et le contexte de l'objet source à la fonction Lambda lorsqu'elle l'invoque. Choisissez Save (Enregistrer).

Vous avez joint votre première résolveur avec succès. Répétez cette opération pour les autres champs.

Testez votre API GraphQL

Maintenant que votre fonction Lambda est connectée aux résolveurs GraphQL, vous pouvez exécuter des mutations et des requêtes à l'aide de la console ou d'une application cliente.

Sur le côté gauche duAWS AppSyncconsole, choisissezRequêtes, puis collez le code suivant :

addPost Mutation

```
mutation AddPost {
    addPost(
        id: 6
        author: "Author6"
        title: "Sixth book"
        url: "https://www.amazon.com/"
        content: "This is the book is a tutorial for using GraphQL with AWS AppSync."
    ) {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

getPost Query

```
query GetPost {
    getPost(id: "2") {
        id
        author
        title
        content
        url
```

```
        ups
        downs
    }
}
```

allPosts Query

```
query AllPosts {
    allPosts {
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {
            id
            title
        }
    }
}
```

Erreurs de renvoi

Toute résolution de champ donnée peut entraîner une erreur. AvecAWS AppSync, vous pouvez générer des erreurs à partir des sources suivantes :

- Gestionnaire de réponses Resolver
- Fonction Lambda

À partir du gestionnaire de réponses du résolveur

Pour signaler des erreurs intentionnelles, vous pouvez utiliser le`util.error`méthode utilitaire. Il faut un argument et`errorMessage`, une`errorType`, et une option`data`valeur. L'objet `data` est utile pour renvoyer des données supplémentaires au client, lorsqu'une erreur a été déclenchée. L'objet `data` sera ajouté à `errors` dans la réponse GraphQL finale.

L'exemple suivant montre comment l'utiliser dans le`Post.relatedPosts: [Post]`gestionnaire de réponses du résolveur.

```
// the Post.relatedPosts response handler
export function response(ctx) {
    util.error("Failed to fetch relatedPosts", "LambdaFailure", ctx.result)
    return ctx.result;
}
```

Cela génère une réponse GraphQL similaire à ce qui suit :

```
{
    "data": {
        "allPosts": [
            {
                "id": "2",
                "title": "Second book",
                "relatedPosts": null
            },
            ...
        ]
    },
    "errors": [
        {
            "path": [
                "allPosts",
                0,
                "relatedPosts"
            ],
            "errorType": "LambdaFailure",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ],
            "message": "Failed to fetch relatedPosts",
            "data": [
                {
                  "id": "2",
                  "title": "Second book"
                },
                {
                  "id": "1",
                  "title": "First book"
                }
```

```
            ]
        }
    ]
}
```

où `allPosts[0].relatedPosts` est null du fait de l'erreur et `errorMessage`, `errorType` et `data` sont présents dans l'objet `data.errors[0]`.

À partir de la fonction Lambda

AWS AppSynccomprend également les erreurs générées par la fonction Lambda. Le modèle de programmation Lambda vous permet d'augmentermanipuléerreurs. Si la fonction Lambda génère une erreur,AWS AppSyncne parvient pas à résoudre le champ actuel. Seul le message d'erreur renvoyé par Lambda est défini dans la réponse. Actuellement, vous ne pouvez pas renvoyer de données superflues au client en déclenchant une erreur à partir de la fonction Lambda.

> **Note**
>
> Si votre fonction Lambda déclenche unnon manipuléerreur,AWS AppSyncutilise le message d'erreur défini par Lambda.

La fonction Lambda suivante génère une erreur :

```
export const handler = async (event) => {
  console.log('Received event {}', JSON.stringify(event, 3))
  throw new Error('I always fail.')
}
```

L'erreur est reçue dans votre gestionnaire de réponses. Vous pouvez le renvoyer dans la réponse GraphQL en ajoutant l'erreur à la réponse avec`util.appendError`. Pour ce faire, modifiez votreAWS AppSyncgestionnaire de réponse fonctionnelle à ceci :

```
// the lambdaInvoke response handler
export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  }
  return result;
```

```
}
```

Cela renvoie une réponse GraphQL similaire à ce qui suit :

```
{
  "data": {
    "allPosts": null
  },
  "errors": [
    {
      "path": [
        "allPosts"
      ],
      "data": null,
      "errorType": "Lambda:Unhandled",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "I fail. always"
    }
  ]
}
```

Cas d'utilisation avancé : traitement par lots

Dans cet exemple, la fonction Lambda possède une`relatedPosts`champ qui renvoie une liste de publications associées à une publication donnée. Dans les exemples de requêtes, le`allPosts`l'invocation de champs à partir de la fonction Lambda renvoie cinq messages. Parce que nous avons précisé que nous voulions également résoudre`relatedPosts`pour chaque courrier retourné, le`relatedPosts`une opération de terrain est invoquée cinq fois.

```
query {
    allPosts {   // 1 Lambda invocation - yields 5 Posts
        id
        author
        title
        content
```

```
        url
        ups
        downs
        relatedPosts {   // 5 Lambda invocations - each yields 5 posts
            id
            title
        }
    }
}
```

Bien que cela ne semble pas important dans cet exemple spécifique, ce surchargement aggravé peut rapidement saper l'application.

Si vous souhaitez à nouveau extraire `relatedPosts` sur le `Posts` associé renvoyé dans la même requête, le nombre d'appels augmenterait considérablement.

```
query {
    allPosts {   // 1 Lambda invocation - yields 5 Posts
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {   // 5 Lambda invocations - each yield 5 posts = 5 x 5 Posts
            id
            title
            relatedPosts {  // 5 x 5 Lambda invocations - each yield 5 posts = 25 x 5
 Posts
                id
                title
                author
            }
        }
    }
}
```

Dans cette requête relativement simple,AWS AppSyncinvoquerait la fonction Lambda 1 + 5 + 25 = 31 fois.

Il s'agit d'un défi assez courant, souvent appelé « problème N+1 » (dans ce cas, N = 5) et qui peut entraîner une augmentation de la latence et des coûts de l'application.

L'une des approches possibles pour résoudre ce problème est de regrouper les demandes de résolveur de champ similaires. Dans cet exemple, au lieu de demander à la fonction Lambda de résoudre une liste de publications connexes pour une publication donnée, elle pourrait résoudre une liste de publications connexes pour un lot de publications donné.

Pour le démontrer, mettons à jour le résolveur pour`relatedPosts`pour gérer le traitement par lots.

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: ctx.info.fieldName === 'relatedPosts' ? 'BatchInvoke' : 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  }
  return result;
}
```

Le code modifie désormais l'opération de`Invoke`pour`BatchInvoke`lorsque le`fieldName`être résolu est`relatedPosts`. Activez maintenant le traitement par lots sur la fonction dans leConfigurer le traitement par lotssection. Définissez la taille de lot maximale définie sur`5`. Choisissez Save (Enregistrer).

Avec cette modification, lors de la résolution`relatedPosts`, la fonction Lambda reçoit les informations suivantes en entrée :

```
[
    {
        "field": "relatedPosts",
        "source": {
            "id": 1
        }
    },
    {
```

```
        "field": "relatedPosts",
        "source": {
            "id": 2
        }
    },
    ...
]
```

Quand`BatchInvoke`est spécifiée dans la demande, la fonction Lambda reçoit une liste de demandes et renvoie une liste de résultats.

Plus précisément, la liste des résultats doit correspondre à la taille et à l'ordre des entrées de charge utile de la demande afin queAWS AppSyncpeut faire correspondre les résultats en conséquence.

Dans cet exemple de traitement par lots, la fonction Lambda renvoie un lot de résultats comme suit :

```
[
    [{"id":"2","title":"Second book"}, {"id":"3","title":"Third book"}],   //
 relatedPosts for id=1
    [{"id":"3","title":"Third book"}]                                      //
 relatedPosts for id=2
]
```

Vous pouvez mettre à jour votre code Lambda pour gérer le traitement par lots pour`relatedPosts`:

```
export const handler = async (event) => {
  console.log('Received event {}', JSON.stringify(event, 3))
  //throw new Error('I fail. always')

  const posts = {
    1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
 content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
 AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
 content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
 '10', },
    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
 ups: null, downs: null },
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
```

```
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
    1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }

  if (!event.field && event.length){
    console.log(`Got a BatchInvoke Request. The payload has ${event.length} items to
 resolve.`);
    return event.map(e => relatedPosts[e.source.id])
  }

  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
    case 'getPost':
      return posts[event.arguments.id]
    case 'allPosts':
      return Object.values(posts)
    case 'addPost':
      // return the arguments back
      return event.arguments
    case 'addPostErrorWithData':
      result = posts[event.arguments.id]
      // attached additional error information to the post
      result.errorMessage = 'Error with the mutation, data has changed'
      result.errorType = 'MUTATION_ERROR'
      return result
    case 'relatedPosts':
      return relatedPosts[event.source.id]
    default:
      throw new Error('Unknown field, unable to resolve ' + event.field)
  }
}
```

Renvoi d'erreurs individuelles

Les exemples précédents montrent qu'il est possible de renvoyer une seule erreur à partir de la fonction Lambda ou de générer une erreur à partir de votre gestionnaire de réponses. Pour les appels par lots, le fait de générer une erreur à partir de la fonction Lambda indique qu'un lot entier a échoué. Cela peut être acceptable pour des scénarios spécifiques dans lesquels une erreur irrécupérable se produit, telle qu'un échec de connexion à un magasin de données. Toutefois, dans les cas où certains éléments du lot réussissent et d'autres échouent, il est possible de renvoyer à la fois des erreurs et des données valides. Parce queAWS AppSyncnécessite que la réponse du lot répertorie les éléments correspondant à la taille d'origine du lot. Vous devez définir une structure de données capable de différencier les données valides d'une erreur.

Par exemple, si la fonction Lambda est censée renvoyer un lot de publications connexes, vous pouvez choisir de renvoyer une liste de`Response`objets où chaque objet est facultatifdonnées,Message d'erreur, etType d'erreurchamps. Si le champ errorMessage est présent, cela signifie qu'une erreur s'est produite.

Le code suivant montre comment mettre à jour la fonction Lambda :

```
export const handler = async (event) => {
console.log('Received event {}', JSON.stringify(event, 3))
  // throw new Error('I fail. always')
const posts = {
1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
 content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
 AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
 content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
 '10', },
    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
 ups: null, downs: null },
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
```

```
1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }

  if (!event.field && event.length){
console.log(`Got a BatchInvoke Request. The payload has ${event.length} items to
 resolve.`);
    return event.map(e => {
// return an error for post 2
if (e.source.id === '2') {
return { 'data': null, 'errorMessage': 'Error Happened', 'errorType': 'ERROR' }
      }
      return {data: relatedPosts[e.source.id]}
      })
  }

  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
case 'getPost':
      return posts[event.arguments.id]
    case 'allPosts':
      return Object.values(posts)
    case 'addPost':
      // return the arguments back
return event.arguments
    case 'addPostErrorWithData':
      result = posts[event.arguments.id]
      // attached additional error information to the post
      result.errorMessage = 'Error with the mutation, data has changed'
      result.errorType = 'MUTATION_ERROR'
return result
    case 'relatedPosts':
      return relatedPosts[event.source.id]
    default:
      throw new Error('Unknown field, unable to resolve ' + event.field)
  }
}
```

Mettez à jour le`relatedPosts`code du résolveur :

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: ctx.info.fieldName === 'relatedPosts' ? 'BatchInvoke' : 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  } else if (result.errorMessage) {
    util.appendError(result.errorMessage, result.errorType, result.data)
  } else if (ctx.info.fieldName === 'relatedPosts') {
      return result.data
  } else {
      return result
  }
}
```

Le gestionnaire de réponses vérifie désormais les erreurs renvoyées par la fonction Lambda sur`Invoke`opérations, vérifications des erreurs renvoyées pour des articles individuels pour`BatchInvoke`opérations, et enfin vérifie le`fieldName`. Pour`relatedPosts`, la fonction renvoie`result.data`. Pour tous les autres champs, la fonction renvoie simplement`result`. Par exemple, consultez la requête ci-dessous :

```
query AllPosts {
  allPosts {
    id
    title
    content
    url
    ups
    downs
    relatedPosts {
      id
    }
    author
  }
```

```
}
```

Cette requête renvoie une réponse GraphQL similaire à la suivante :

```
{
  "data": {
    "allPosts": [
      {
        "id": "1",
        "relatedPosts": [
          {
            "id": "4"
          }
        ]
      },
      {
        "id": "2",
        "relatedPosts": null
      },
      {
        "id": "3",
        "relatedPosts": [
          {
            "id": "2"
          },
          {
            "id": "1"
          }
        ]
      },
      {
        "id": "4",
        "relatedPosts": [
          {
            "id": "2"
          },
          {
            "id": "1"
          }
        ]
      },
      {
        "id": "5",
```

```
        "relatedPosts": []
      }
    ]
  },
  "errors": [
    {
      "path": [
        "allPosts",
        1,
        "relatedPosts"
      ],
      "data": null,
      "errorType": "ERROR",
      "errorInfo": null,
      "locations": [
        {
          "line": 4,
          "column": 5,
          "sourceName": null
        }
      ],
      "message": "Error Happened"
    }
  ]
}
```

Configuration de la taille de lot maximale

Pour configurer la taille de traitement par lots maximale sur un résolveur, utilisez la commande suivante dans leAWS Command Line Interface(AWS CLI) :

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
 relatedPosts \
 --code "<code-goes-here>" \
 --runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
 --data-source-name "<lambda-datasource>" \
 --max-batch-size X
```

> **Note**
>
> Lorsque vous fournissez un modèle de mappage de demandes, vous devez utiliser le`BatchInvoke`opération pour utiliser le traitement par lots.

Tutoriel : résolveurs locaux

AWS AppSyncvous permet d'utiliser les sources de données prises en charge (AWS Lambda, Amazon DynamoDB ou AmazonOpenSearchService) pour effectuer diverses opérations. Cependant, dans certains cas, un appel à une source de données prise en charge peut ne pas être nécessaire.

C'est là où le résolveur local se révèle pratique. Au lieu d'appeler une source de données distante, le résolveur local va simplementvers l'avantle résultat du gestionnaire de demandes envoyé au gestionnaire de réponses. La résolution du champ ne partira pasAWS AppSync.

Les résolveurs locaux sont utiles dans une multitude de situations. Le scénario le plus répandu consiste à publier des notifications sans déclencher d'appel de source de données. Pour illustrer ce cas d'utilisation, créons une application pub/sub dans laquelle les utilisateurs peuvent publier des messages et s'y abonner. Comme cet exemple met à profit les Abonnements, si vous n'êtes pas familiarisé avec les Abonnements, vous pouvez suivre le didacticiel Données en temps réel.

Création de l'application pub/sub

Créez d'abord une API GraphQL vide en choisissantDesign à partir de zérooption et configuration des détails facultatifs lors de la création de votre API GraphQL.

Dans notre application pub/sub, les clients peuvent s'abonner à des messages et les publier. Chaque message publié inclut un nom et des données. Ajoutez ceci au schéma :

```
type Channel {
 name: String!
 data: AWSJSON!
}

type Mutation {
 publish(name: String!, data: AWSJSON!): Channel
}

type Query {
```

```
 getChannel: Channel
}

type Subscription {
 subscribe(name: String!): Channel
  @aws_subscribe(mutations: ["publish"])
}
```

Attachons ensuite un résolveur au`Mutation.publish`champ. Dans le`Résolveurs`volet situé à côté du`Schéma`volet, trouvez le`Mutation`ontapez, puis`publish(...): Channel`champ, puis cliquez sur`Joindre`.

Créez un`Aucune`source de données et nommez-la`PageDataSource`. Attachez-le à votre résolveur.

Ajoutez votre implémentation de résolveur à l'aide de l'extrait de code suivant :

```
export function request(ctx) {
  return { payload: ctx.args };
}

export function response(ctx) {
  return ctx.result;
}
```

Assurez-vous de créer le résolveur et d'enregistrer les modifications que vous avez apportées.

Envoyer des messages et s'y abonner

Pour que les clients puissent recevoir des messages, ils doivent d'abord être abonnés à une boîte de réception.

Dans le`Requêtes`volet, exécutez le`SubscribeToData`abonnement :

```
subscription SubscribeToData {
    subscribe(name:"channel") {
        name
        data
    }
}
```

L'abonné recevra des messages chaque fois que`publish`la mutation est invoquée mais uniquement lorsque le message est envoyé au`channel`abonnement. Essayons ceci dans`Requêtes`volet. Pendant

que votre abonnement est toujours en cours d'exécution dans la console, ouvrez une autre console et exécutez la requête suivante dansRequêtesvolet :

> **Note**
>
> Nous utilisons des chaînes JSON valides dans cet exemple.

```
mutation PublishData {
    publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
        data
        name
    }
}
```

Le résultat doit se présenter comme suit :

```
{
  "data": {
    "publish": {
      "data": "{\"msg\":\"hello world!\"}",
      "name": "channel"
    }
  }
}
```

Nous venons de démontrer l'utilisation de résolveurs locaux, en publiant un message et en le recevant sans quitter leAWS AppSyncservice.

Tutoriel : Combiner des résolveurs GraphQL

Les résolveurs et les champs d'un schéma GraphQL possèdent des relations 1:1 avec un haut niveau de flexibilité. Comme une source de données est configurée sur un résolveur indépendamment d'un schéma, vous avez la possibilité de résoudre ou de manipuler vos types GraphQL via différentes sources de données, ce qui vous permet de combiner un schéma pour répondre au mieux à vos besoins.

Les scénarios suivants montrent comment mélanger et associer des sources de données dans votre schéma. Avant de commencer, vous devez être familiarisé avec la configuration des sources

de données et des résolveurs pourAWS Lambda, Amazon DynamoDB et AmazonOpenSearchUn service.

Exemple de schéma

Le schéma suivant a un type de`Post`avec trois`Query`et`Mutation`opérations chacune :

```
type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
}

type Query {
    allPost: [Post]
    getPost(id: ID!): Post
    searchPosts: [Post]
}

type Mutation {
    addPost(
        id: ID!,
        author: String!,
        title: String,
        content: String,
        url: String
    ): Post
    updatePost(
        id: ID!,
        author: String!,
        title: String,
        content: String,
        url: String,
        ups: Int!,
        downs: Int!,
        expectedVersion: Int!
    ): Post
    deletePost(id: ID!): Post
```

```
}
```

Dans cet exemple, vous auriez un total de six résolveurs, chacun ayant besoin d'une source de données. Une façon de résoudre ce problème serait de les connecter à une seule table Amazon DynamoDB, appeléePosts, dans lequelAllPostle champ exécute un scan et lesearchPostsle champ exécute une requête (voirJavaScriptréférence de fonction de résolution pour DynamoDB). Cependant, vous n'êtes pas limité à Amazon DynamoDB ; différentes sources de données telles que Lambda ouOpenSearchLe service existe pour répondre aux besoins de votre entreprise.

Modification des données par le biais de résolveurs

Vous devrez peut-être renvoyer les résultats d'une base de données tierce qui n'est pas directement prise en charge parAWS AppSyncsources de données. Il se peut également que vous deviez effectuer des modifications complexes sur les données avant qu'elles ne soient renvoyées au (x) client (s) de l'API. Cela peut être dû à un formatage incorrect des types de données, tel que des différences d'horodatage sur les clients ou à la gestion de problèmes de rétrocompatibilité. Dans ce cas, la connexionAWS Lambdafonctionne comme une source de données pour votreAWS AppSyncL'API est la solution appropriée. À des fins d'illustration, dans l'exemple suivant, unAWS Lambdafonction manipule les données extraites d'un magasin de données tiers :

```
export const handler = (event, context, callback) => {
    // fetch data
    const result = fetcher()

    // apply complex business logic
    const data = transform(result)

    // return to AppSync
    return data
};
```

Il s'agit d'une fonction Lambda parfaitement valide qui peut être attachée à un champ `AllPost` dans le schéma GraphQL afin que toute requête renvoyant tous les résultats obtienne des nombres aléatoires pour les pour et/ou les contre.

DynamoDB etOpenSearchService

Pour certaines applications, vous pouvez effectuer des mutations ou de simples requêtes de recherche sur DynamoDB et demander à un processus en arrière-plan de transférer des

documents versOpenSearchUn service. Vous pouvez simplement joindre le`searchPosts`résolveur duOpenSearchSource de données de service et retour des résultats de recherche (à partir de données provenant de DynamoDB) à l'aide d'une requête GraphQL. Cela peut être extrêmement puissant lorsque vous ajoutez des opérations de recherche avancées à vos applications, telles que des mots clés, des correspondances de mots flous ou même des recherches géospatiales. Le transfert de données depuis DynamoDB peut être effectué via un processus ETL, ou vous pouvez également diffuser des données depuis DynamoDB à l'aide de Lambda.

Pour commencer à utiliser ces sources de données spécifiques, consultez notreDynamoDBetLambdatutoriels.

Par exemple, en utilisant le schéma de notre précédent didacticiel, la mutation suivante ajoute un élément à DynamoDB :

```
mutation addPost {
  addPost(
    id: 123
    author: "Nadia"
    title: "Our first post!"
    content: "This is our first post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

Cela écrit les données dans DynamoDB, qui les diffuse ensuite via Lambda vers AmazonOpenSearchService, que vous utilisez ensuite pour rechercher des articles par différents champs. Par exemple, étant donné que les données se trouvent dans AmazonOpenSearchService, vous pouvez rechercher l'auteur ou les champs de contenu avec du texte libre, même avec des espaces, comme suit :

```
query searchName{
    searchAuthor(name:"   Nadia   "){
```

```
        id
        title
        content
    }
}

---------- or ----------

query searchContent{
    searchContent(text:"test"){
        id
        title
        content
    }
}
```

Les données étant écrites directement dans DynamoDB, vous pouvez toujours effectuer des opérations efficaces de recherche de listes ou d'éléments par rapport à la table à l'aide du `allPost{...}` et `getPost{...}` requêtes. Cette pile utilise l'exemple de code suivant pour les flux DynamoDB :

> **Note**
>
> Ce code Python est un exemple et n'est pas destiné à être utilisé dans le code de production.

```
import boto3
import requests
from requests_aws4auth import AWS4Auth

region = '' # e.g. us-east-1
service = 'es'
credentials = boto3.Session().get_credentials()
awsauth = AWS4Auth(credentials.access_key, credentials.secret_key, region, service,
 session_token=credentials.token)

host = '' # the OpenSearch Service domain, e.g. https://search-mydomain.us-
west-1.es.amazonaws.com
index = 'lambda-index'
datatype = '_doc'
url = host + '/' + index + '/' + datatype + '/'
```

```
headers = { "Content-Type": "application/json" }

def handler(event, context):
    count = 0
    for record in event['Records']:
        # Get the primary key for use as the OpenSearch ID
        id = record['dynamodb']['Keys']['id']['S']

        if record['eventName'] == 'REMOVE':
            r = requests.delete(url + id, auth=awsauth)
        else:
            document = record['dynamodb']['NewImage']
            r = requests.put(url + id, auth=awsauth, json=document, headers=headers)
        count += 1
    return str(count) + ' records processed.'
```

Vous pouvez ensuite utiliser des flux DynamoDB pour l'associer à une table DynamoDB avec une clé primaire de`id`, et toute modification apportée à la source de DynamoDB sera répercutée dans votreOpenSearchDomaine de service. Pour plus d'informations sur la configuration de ce processus, consultez la Documentation DynamoDB Streams.

Tutoriel : AmazonOpenSearchRésolveurs de services

AWS AppSyncprend en charge l'utilisation d'AmazonOpenSearchService fourni par des domaines que vous avez fournis vous-mêmeAWScompte, à condition qu'ils n'existent pas dans un VPC. Une fois que vos domaines sont mis en service, vous pouvez vous y connecter à l'aide d'une source de données, puis vous pouvez configurer un résolveur dans le schéma afin d'effectuer des opérations GraphQL telles que des requêtes, des mutations et des abonnements. Ce didacticiel vous présente certains exemples courants.

Pour plus d'informations, consultez notreJavaScriptréférence de la fonction de résolution pourOpenSearch.

Créez un nouveauOpenSearchDomaine de service

Pour commencer à utiliser ce didacticiel, vous avez besoin d'unOpenSearchDomaine de service. Si vous n'en avez pas, vous pouvez utiliser l'exemple suivant. Notez que cela peut prendre jusqu'à 15 minutes pourOpenSearchLe domaine de service doit être créé avant de pouvoir passer à son intégration à unAWS AppSyncsource de données.

```
aws cloudformation create-stack --stack-name AppSyncOpenSearch \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/elasticsearch/
ESResolverCFTemplate.yaml \
--parameters ParameterKey=OSDomainName,ParameterValue=ddtestdomain
 ParameterKey=Tier,ParameterValue=development \
--capabilities CAPABILITY_NAMED_IAM
```

Vous pouvez lancer ce qui suitAWS CloudFormationempilez dans la région US-West-2 (Oregon) dans votreAWScompte :

Configurer une source de données pourOpenSearchService

Après leOpenSearchLe domaine de service est créé, accédez à votreAWS AppSyncAPI GraphQL et choisissezSources de donnéesonglet. ChoisissezCréation d'une source de donnéeset entrez un nom convivial pour la source de données, tel que »*perte*». Ensuite, choisissezAmazonOpenSearchdomainepourType de source de données, choisissez la région appropriée, et vous devriez voir votreOpenSearchDomaine de service répertorié. Après l'avoir sélectionné, vous pouvez soit créer un nouveau rôle, soitAWS AppSyncattribuera les autorisations appropriées au rôle, ou vous pouvez choisir un rôle existant, dont la politique intégrée est la suivante :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1234234",
            "Effect": "Allow",
            "Action": [
                "es:ESHttpDelete",
                "es:ESHttpHead",
                "es:ESHttpGet",
                "es:ESHttpPost",
                "es:ESHttpPut"
            ],
            "Resource": [
                "arn:aws:es:REGION:ACCOUNTNUMBER:domain/democluster/*"
            ]
        }
```

```
    ]
}
```

Vous devrez également établir une relation de confiance avecAWS AppSyncpour ce rôle :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "appsync.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

De plus, leOpenSearchLe domaine de service a son proprePolitique d'accèsque vous pouvez modifier via AmazonOpenSearchConsole de service. Vous devez ajouter une politique similaire à celle ci-dessous avec les actions et les ressources appropriées pourOpenSearchDomaine de service. Notez que lePrincipalsera leAWS AppSyncrôle de source de données, qui se trouve dans la console IAM si vous laissez cette console le créer.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Effect": "Allow",
        "Principal": {
            "AWS": "arn:aws:iam::ACCOUNTNUMBER:role/service-role/
APPSYNC_DATASOURCE_ROLE"
        },
        "Action": [
            "es:ESHttpDelete",
            "es:ESHttpHead",
            "es:ESHttpGet",
            "es:ESHttpPost",
            "es:ESHttpPut"
        ],
        "Resource": "arn:aws:es:REGION:ACCOUNTNUMBER:domain/DOMAIN_NAME/*"
        }
```

```
    ]
}
```

Connecter un résolveur

Maintenant que la source de données est connectée à votreOpenSearchDomaine de service, vous pouvez le connecter à votre schéma GraphQL à l'aide d'un résolveur, comme indiqué dans l'exemple suivant :

```
 type Query {
   getPost(id: ID!): Post
   allPosts: [Post]
 }

 type Mutation {
   addPost(id: ID!, author: String, title: String, url: String, ups: Int, downs: Int,
 content: String): AWSJSON
 }

type Post {
  id: ID!
  author: String
  title: String
  url: String
  ups: Int
  downs: Int
  content: String
}
```

Notez qu'il y a un type `Post` défini par l'utilisateur avec un champ `id`. Dans les exemples suivants, nous supposons qu'il existe un processus (qui peut être automatisé) pour introduire ce type dans votreOpenSearchDomaine de service, qui serait mappé à une racine de chemin de`/post/_doc`où`post`est l'indice. À partir de ce chemin racine, vous pouvez effectuer des recherches de documents individuels, des recherches par caractères génériques avec`/id/post*`, ou des recherches dans plusieurs documents avec un chemin de`/post/_search`. Par exemple, si vous avez un autre type appelé`User`, vous pouvez indexer des documents sous un nouvel index appelé`user`, puis effectuez des recherches à l'aide d'uncheminde`/user/_search`.

À partir du**Schéma**éditeur dans le**AWS AppSync**console, modifiez le précédent`Posts`schéma pour inclure un`searchPosts`requête :

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

Enregistrez le schéma. Dans leRésolveursvolet, recherchez`searchPosts`et choisissezJoindre. Choisissez votreOpenSearchSource de données de service et enregistrez le résolveur. Mettez à jour le code de votre résolveur à l'aide de l'extrait ci-dessous :

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by using an input term
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
 return {
  operation: 'GET',
  path: `/post/_search`,
  params: { body: { from: 0, size: 50 } },
 }
}

/**
 * Returns the fetched items
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type)
 }
 return ctx.result.hits.hits.map((hit) => hit._source)
}
```

Cela suppose que le schéma précédent contient des documents qui ont été indexés dansOpenSearchService dans le cadre du`post`champ. Si vous structurez vos données différemment, vous devrez les mettre à jour en conséquence.

Modifier vos recherches

Le gestionnaire de demandes de résolution précédent exécute une requête simple pour tous les enregistrements. Supposons que vous souhaitiez effectuer une recherche en fonction d'un auteur spécifique. Supposons également que vous souhaitiez que cet auteur soit un argument défini dans votre requête GraphQL. Dans leSchémarédacteur en chef duAWS AppSyncconsole, ajoutez un`allPostsByAuthor`requête :

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

Dans leRésolveursvolet, recherche`allPostsByAuthor`et choisissezJoindre. Choisissez leOpenSearchSource de données de service et utilisez le code suivant :

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
 return {
  operation: 'GET',
  path: '/post/_search',
  params: {
   body: {
    from: 0,
    size: 50,
    query: { match: { author: ctx.args.author } },
   },
  },
 }
}

/**
 * Returns the fetched items
 * @param {import('@aws-appsync/utils').Context} ctx the context
```

```
 * @returns {*} the result
 */
export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type)
 }
 return ctx.result.hits.hits.map((hit) => hit._source)
}
```

Notez que l'élément body est renseigné avec une requête terminologique concernant le champ `author`, qui est transmise à partir du client en tant qu'argument. Vous pouvez éventuellement utiliser des informations préremplies, telles que du texte standard.

Ajouter des données àOpenSearchService

Vous souhaiterez peut-être ajouter des données à votreOpenSearchDomaine de service résultant d'une mutation GraphQL. Il s'agit d'un puissant mécanisme de recherche, qui peut également avoir d'autres fonctions. Parce que vous pouvez utiliser des abonnements GraphQL pourtransformez vos données en temps réel, il peut servir de mécanisme pour informer les clients des mises à jour des données de votreOpenSearchDomaine de service.

Retournez auSchémapage dans leAWS AppSyncconsole et sélectionnezJoindrepour le`addPost()`mutation. Sélectionnez leOpenSearchReprenez la source des données du service et utilisez le code suivant :

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
 return {
  operation: 'PUT',
  path: `/post/_doc/${ctx.args.id}`,
  params: { body: ctx.args },
 }
}

/**
 * Returns the inserted post
```

```
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type)
 }
 return ctx.result
}
```

Comme précédemment, il s'agit d'un exemple de la manière dont vos données peuvent être structurées. Si vous avez des noms de champs ou des index différents, vous devez mettre à jour le`path`et`body`. Cet exemple montre également comment utiliser`context.arguments`, qui peut également être écrit comme`ctx.args`, dans votre gestionnaire de demandes.

Récupération d'un seul document

Enfin, si vous souhaitez utiliser`getPost(id:ID)`dans votre schéma pour renvoyer un document individuel, trouvez cette requête dans leSchémarédacteur en chef duAWS AppSyncconsole et choisissezJoindre. Sélectionnez leOpenSearchReprenez la source des données du service et utilisez le code suivant :

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
 return {
  operation: 'GET',
  path: `/post/_doc/${ctx.args.id}`,
 }
}

/**
 * Returns the post
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
```

```
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type)
 }
 return ctx.result._source
}
```

Effectuer des requêtes et des mutations

Vous devriez maintenant être en mesure d'effectuer des opérations GraphQL sur votreOpenSearchDomaine de service. Naviguez vers leRequêtesonglet duAWS AppSyncconsole et ajoutez un nouvel enregistrement :

```
mutation AddPost {
    addPost (
        id:"12345"
        author: "Fred"
        title: "My first book"
        content: "This will be fun to write!"
        url: "publisher website",
        ups: 100,
        downs:20
       )
}
```

Vous verrez le résultat de la mutation sur la droite. De même, vous pouvez désormais exécuter un`searchPosts`requête contre votreOpenSearchDomaine de service :

```
query search {
    searchPosts {
        id
        title
        author
        content
    }
}
```

Bonnes pratiques

- OpenSearchLe service doit être destiné à interroger des données, et non en tant que base de données principale. Vous souhaiterez peut-être utiliserOpenSearchService associé à Amazon DynamoDB, comme indiqué dansCombinaison de résolveurs GraphQL.

- Ne donnez accès à votre domaine qu'en autorisantAWS AppSyncrôle de service pour accéder au cluster.
- Vous pouvez commencer la phase de développement de façon modeste, avec le cluster le moins onéreux, puis passer à un plus grand cluster à haute disponibilité (HA) lorsque vous passerez en production.

Tutoriel : résolveurs de transactions DynamoDB

AWS AppSyncprend en charge l'utilisation des opérations de transaction Amazon DynamoDB sur une ou plusieurs tables d'une même région. Les opérations prises en charge sont `TransactGetItems` et `TransactWriteItems`. En utilisant ces fonctionnalités dansAWS AppSync, vous pouvez effectuer des tâches telles que :

- Transmission d'une liste de clés en une seule requête et renvoi des résultats à partir d'une table
- Lecture d'enregistrements d'une ou de plusieurs tables en une seule requête
- Écrire des enregistrements de transactions dans une ou plusieurs tables d'unall-or-nothingfaçon
- Exécution de transactions lorsque certaines conditions sont satisfaites

Autorisations

Comme les autres résolveurs, vous devez créer une source de données dansAWS AppSyncet créez un rôle ou utilisez un rôle existant. Les opérations de transaction nécessitant des autorisations différentes sur les tables DynamoDB, vous devez accorder au rôle configuré des autorisations pour les actions de lecture ou d'écriture :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Effect": "Allow",
```

```
            "Resource": [
                "arn:aws:dynamodb:region:accountId:table/TABLENAME",
                "arn:aws:dynamodb:region:accountId:table/TABLENAME/*"
            ]
        }
    ]
}
```

> **Note**
>
> Les rôles sont liés aux sources de données dansAWS AppSync, et les résolveurs de champs sont invoqués par rapport à une source de données. Les sources de données configurées pour effectuer des extractions par rapport à DynamoDB n'ont qu'une seule table spécifiée pour simplifier les configurations. Par conséquent, lorsque vous effectuez une opération de transaction sur plusieurs tables dans un seul résolveur (ce qui constitue une tâche plus avancée), vous devez accorder au rôle associé à cette source de données l'accès à toutes les tables avec lesquelles le résolveur devra interagir. Cela doit être effectué dans le champ Resource (Ressource) dans la stratégie IAM ci-dessus. La configuration des appels de transaction par rapport aux tables est effectuée dans le code du résolveur, que nous décrivons ci-dessous.

Source de données

Dans un souci de simplicité, nous allons utiliser la même source de données pour tous les résolveurs utilisés dans ce didacticiel.

Nous aurons deux tables appeléesComptes d'épargneetVérification des comptes, tous deux avec le`accountNumber`en tant que clé de partition, etHistorique des transactionstable avec`transactionId`comme clé de partition. Vous pouvez utiliser les commandes CLI ci-dessous pour créer vos tables. Assurez-vous de remplacer`region`avec votre région.

Avec la CLI

```
aws dynamodb create-table --table-name savingAccounts \
  --attribute-definitions AttributeName=accountNumber,AttributeType=S \
  --key-schema AttributeName=accountNumber,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region
```

```
aws dynamodb create-table --table-name checkingAccounts \
  --attribute-definitions AttributeName=accountNumber,AttributeType=S \
  --key-schema AttributeName=accountNumber,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region

aws dynamodb create-table --table-name transactionHistory \
  --attribute-definitions AttributeName=transactionId,AttributeType=S \
  --key-schema AttributeName=transactionId,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region
```

Dans leAWS AppSyncconsole, enSources de données, créez une nouvelle source de données DynamoDB et nommez-laTransactTutorial. SélectionnezComptes d'épargnecomme table (bien que la table spécifique n'ait pas d'importance lors de l'utilisation de transactions). Choisissez de créer un nouveau rôle et une nouvelle source de données. Vous pouvez consulter la configuration de la source de données pour connaître le nom du rôle généré. Dans la console IAM, vous pouvez ajouter une politique en ligne qui permet à la source de données d'interagir avec toutes les tables.

Remplacer`region`et`accountID`avec votre région et votre numéro de compte :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:region:accountId:table/savingAccounts",
                "arn:aws:dynamodb:region:accountId:table/savingAccounts/*",
                "arn:aws:dynamodb:region:accountId:table/checkingAccounts",
                "arn:aws:dynamodb:region:accountId:table/checkingAccounts/*",
                "arn:aws:dynamodb:region:accountId:table/transactionHistory",
                "arn:aws:dynamodb:region:accountId:table/transactionHistory/*"
            ]
        }
```

```
    ]
}
```

Transactions

Pour cet exemple, le contexte est une transaction bancaire classique, où nous allons utiliser `TransactWriteItems` pour :

- Transférer de l'argent des comptes d'épargne vers des comptes de contrôle
- Générer de nouveaux enregistrements de transaction pour chaque transaction

Ensuite, nous allons utiliser `TransactGetItems` pour récupérer les détails des comptes d'enregistrement et des comptes de vérification.

Nous définissons notre schéma GraphQL comme suit :

```
type SavingAccount {
    accountNumber: String!
    username: String
    balance: Float
}

type CheckingAccount {
    accountNumber: String!
    username: String
    balance: Float
}

type TransactionHistory {
    transactionId: ID!
    from: String
    to: String
    amount: Float
}

type TransactionResult {
    savingAccounts: [SavingAccount]
    checkingAccounts: [CheckingAccount]
    transactionHistory: [TransactionHistory]
}

input SavingAccountInput {
```

```
    accountNumber: String!
    username: String
    balance: Float
}

input CheckingAccountInput {
    accountNumber: String!
    username: String
    balance: Float
}

input TransactionInput {
    savingAccountNumber: String!
    checkingAccountNumber: String!
    amount: Float!
}

type Query {
    getAccounts(savingAccountNumbers: [String], checkingAccountNumbers: [String]):
 TransactionResult
}

type Mutation {
    populateAccounts(savingAccounts: [SavingAccountInput], checkingAccounts:
 [CheckingAccountInput]): TransactionResult
    transferMoney(transactions: [TransactionInput]): TransactionResult
}
```

TransactWriteItems- Remplir les comptes

Afin de transférer de l'argent entre les comptes, nous devons remplir la table avec les détails. Nous allons utiliser l'opération GraphQL `Mutation.populateAccounts` pour le faire.

Dans la section Schéma, cliquez surJoindreà côté du`Mutation.populateAccounts`opération. Choisissez le`TransactTutorial`source de données et choisissezCréez.

Utilisez maintenant le code suivant :

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { savingAccounts, checkingAccounts } = ctx.args
```

```
 const savings = savingAccounts.map(({ accountNumber, ...rest }) => {
  return {
   table: 'savingAccounts',
   operation: 'PutItem',
   key: util.dynamodb.toMapValues({ accountNumber }),
   attributeValues: util.dynamodb.toMapValues(rest),
  }
 })

 const checkings = checkingAccounts.map(({ accountNumber, ...rest }) => {
  return {
   table: 'checkingAccounts',
   operation: 'PutItem',
   key: util.dynamodb.toMapValues({ accountNumber }),
   attributeValues: util.dynamodb.toMapValues(rest),
  }
 })
 return {
  version: '2018-05-29',
  operation: 'TransactWriteItems',
  transactItems: [...savings, ...checkings],
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
 }
 const { savingAccounts: sInput, checkingAccounts: cInput } = ctx.args
 const keys = ctx.result.keys
 const savingAccounts = sInput.map((_, i) => keys[i])
 const sLength = sInput.length
 const checkingAccounts = cInput.map((_, i) => keys[sLength + i])
 return { savingAccounts, checkingAccounts }
}
```

Enregistrez le résolveur et accédez auRequêtessection duAWS AppSyncconsole pour renseigner les comptes.

Exécutez la mutation suivante :

```
mutation populateAccounts {
  populateAccounts (
```

```
    savingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 100},
      {accountNumber: "2", username: "Amy", balance: 90},
      {accountNumber: "3", username: "Lily", balance: 80},
    ]
    checkingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 70},
      {accountNumber: "2", username: "Amy", balance: 60},
      {accountNumber: "3", username: "Lily", balance: 50},
    ]) {
    savingAccounts {
      accountNumber
    }
    checkingAccounts {
      accountNumber
    }
  }
}
```

Nous avons rempli trois comptes d'épargne et trois comptes chèques en une seule mutation.

Utilisez la console DynamoDB pour vérifier que les données apparaissent à la fois dansComptes d'épargneetVérification des comptestables.

TransactWriteItems- Transférer de l'argent

Associez un résolveur au`transferMoney`mutation avec le code suivant. Pour chaque transfert, nous avons besoin d'un modificateur de réussite à la fois pour le compte courant et le compte d'épargne, et nous devons suivre le transfert en termes de transactions.

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const transactions = ctx.args.transactions

 const savings = []
 const checkings = []
 const history = []
 transactions.forEach((t) => {
  const { savingAccountNumber, checkingAccountNumber, amount } = t
  savings.push({
   table: 'savingAccounts',
   operation: 'UpdateItem',
```

```
    key: util.dynamodb.toMapValues({ accountNumber: savingAccountNumber }),
    update: {
     expression: 'SET balance = balance - :amount',
     expressionValues: util.dynamodb.toMapValues({ ':amount': amount }),
    },
   })
   checkings.push({
    table: 'checkingAccounts',
    operation: 'UpdateItem',
    key: util.dynamodb.toMapValues({ accountNumber: checkingAccountNumber }),
    update: {
     expression: 'SET balance = balance + :amount',
     expressionValues: util.dynamodb.toMapValues({ ':amount': amount }),
    },
   })
   history.push({
    table: 'transactionHistory',
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ transactionId: util.autoId() }),
    attributeValues: util.dynamodb.toMapValues({
     from: savingAccountNumber,
     to: checkingAccountNumber,
     amount,
    }),
   })
 })

 return {
  version: '2018-05-29',
  operation: 'TransactWriteItems',
  transactItems: [...savings, ...checkings, ...history],
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
 }
 const tInput = ctx.args.transactions
 const tLength = tInput.length
 const keys = ctx.result.keys
 const savingAccounts = tInput.map((_, i) => keys[tLength * 0 + i])
 const checkingAccounts = tInput.map((_, i) => keys[tLength * 1 + i])
 const transactionHistory = tInput.map((_, i) => keys[tLength * 2 + i])
```

```
 return { savingAccounts, checkingAccounts, transactionHistory }
}
```

Naviguez maintenant vers leRequêtessection duAWS AppSyncconsole et exécutez leTransférer de l'argentmutation comme suit :

```
mutation write {
  transferMoney(
    transactions: [
      {savingAccountNumber: "1", checkingAccountNumber: "1", amount: 7.5},
      {savingAccountNumber: "2", checkingAccountNumber: "2", amount: 6.0},
      {savingAccountNumber: "3", checkingAccountNumber: "3", amount: 3.3}
    ]) {
    savingAccounts {
      accountNumber
    }
    checkingAccounts {
      accountNumber
    }
    transactionHistory {
      transactionId
    }
  }
}
```

Nous avons envoyé trois transactions bancaires en une seule mutation. Utilisez la console DynamoDB pour vérifier que les données apparaissent dansComptes d'épargne,Vérification des comptes, etHistorique des transactionstables.

TransactGetItems- Récupérer des comptes

Afin de récupérer les informations relatives aux comptes d'épargne et aux comptes chèques dans le cadre d'une seule demande transactionnelle, nous associerons un résolveur au`Query.getAccounts`Opération GraphQL sur notre schéma. SélectionnezJoindre, choisissez le même`TransactTutorial`source de données créée au début du didacticiel. Utilisez le code suivant :

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { savingAccountNumbers, checkingAccountNumbers } = ctx.args
```

```
 const savings = savingAccountNumbers.map((accountNumber) => {
  return { table: 'savingAccounts', key: util.dynamodb.toMapValues({ accountNumber }) }
 })
 const checkings = checkingAccountNumbers.map((accountNumber) => {
  return { table: 'checkingAccounts', key:
 util.dynamodb.toMapValues({ accountNumber }) }
 })
 return {
  version: '2018-05-29',
  operation: 'TransactGetItems',
  transactItems: [...savings, ...checkings],
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
 }

 const { savingAccountNumbers: sInput, checkingAccountNumbers: cInput } = ctx.args
 const items = ctx.result.items
 const savingAccounts = sInput.map((_, i) => items[i])
 const sLength = sInput.length
 const checkingAccounts = cInput.map((_, i) => items[sLength + i])
 return { savingAccounts, checkingAccounts }
}
```

Enregistrez le résolveur et accédez auRequêtessections duAWS AppSyncconsole. Pour récupérer les comptes d'épargne et les comptes chèques, exécutez la requête suivante :

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
```

```
      username
      balance
    }
  }
}
```

Nous avons démontré avec succès l'utilisation de transactions DynamoDB en utilisantAWS AppSync.

Tutoriel : résolveurs par lots DynamoDB

AWS AppSyncprend en charge l'utilisation des opérations par lots Amazon DynamoDB sur une ou plusieurs tables d'une même région. Les opérations prises en charge sont `BatchGetItem`, `BatchPutItem` et `BatchDeleteItem`. En utilisant ces fonctionnalités dansAWS AppSync, vous pouvez effectuer des tâches telles que :

- Transmission d'une liste de clés en une seule requête et renvoi des résultats à partir d'une table
- Lecture d'enregistrements d'une ou de plusieurs tables en une seule requête
- Écrire des enregistrements en bloc dans une ou plusieurs tables
- Écrire ou supprimer de manière conditionnelle des enregistrements dans plusieurs tables susceptibles d'avoir une relation

Opérations par lots dansAWS AppSyncprésentent deux différences majeures par rapport aux opérations non groupées :

- Le rôle de source de données doit disposer d'autorisations sur toutes les tables auxquelles le résolveur aura accès.
- La spécification de table pour un résolveur fait partie de l'objet de demande.

Lots à table unique

Pour commencer, créons une nouvelle API GraphQL. Dans leAWS AppSyncconsole, choisissezCréer une API,API GraphQL, etDesign à partir de zéro. Donnez un nom à votre API`BatchTutorial API`, choisissezSuivant, et sur leSpécifier les ressources GraphQLétape, choisissezCréez des ressources GraphQL ultérieurementet cliquezSuivant. Vérifiez vos informations et créez l'API. Accédez auSchémapuis collez le schéma suivant, en notant que pour la requête, nous allons transmettre une liste d'identifiants :

```
type Post {
    id: ID!
    title: String
}

input PostInput {
    id: ID!
    title: String
}

type Query {
    batchGet(ids: [ID]): [Post]
}

type Mutation {
    batchAdd(posts: [PostInput]): [Post]
    batchDelete(ids: [ID]): [Post]
}
```

Enregistrez votre schéma et choisissezCréation de ressourcesen haut de page. ChoisissezUtiliser le type existantet sélectionnez`Post`type. Donnez un nom à votre table`Posts`. Assurez-vous queClé primaireest défini sur`id`, désélectionnezGénérez automatiquement GraphQL(vous fournirez votre propre code), puis sélectionnezCréez. Pour vous aider à démarrer,AWS AppSynccrée une nouvelle table DynamoDB et une source de données connectée à la table avec les rôles appropriés. Cependant, vous devez encore ajouter quelques autorisations au rôle. Accédez auSources de donnéespage et choisissez la nouvelle source de données. SousSélectionnez un rôle existant, vous remarquerez qu'un rôle a été automatiquement créé pour la table. Prenez note du rôle (qui devrait ressembler à`appsync-ds-ddb-aaabbbcccddd-Posts`) puis accédez à la console IAM (https://console.aws.amazon.com/iam/). Dans la console IAM, choisissezRôles, puis choisissez votre rôle dans le tableau. Dans votre rôle, sousPolitiques d'autorisations, cliquez sur le bouton »+« à côté de la politique (doit avoir un nom similaire au nom du rôle). ChoisissezModifieren haut du pliable lorsque la politique apparaît. Vous devez ajouter des autorisations par lots à votre politique, en particulier`dynamodb:BatchGetItem`et`dynamodb:BatchWriteItem`. Cela ressemblera à ceci :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
```

```
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:BatchGetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:…",
                "arn:aws:dynamodb:…"
            ]
        }
    ]
}
```

ChoisissezSuivant, puisEnregistrer les modifications. Votre politique devrait autoriser le traitement par lots dès maintenant.

De retour dans leAWS AppSyncconsole, accédez àSchémapage et sélectionnezJoindreà côté du`Mutation.batchAdd`champ. Créez votre résolveur à l'aide du`Posts`table comme source de données. Dans l'éditeur de code, remplacez les gestionnaires par l'extrait ci-dessous. Cet extrait prend automatiquement chaque élément du GraphQL`input PostInput`tapez et crée une carte, qui est nécessaire pour`BatchPutItem`opération :

```
import { util } from "@aws-appsync/utils";

export function request(ctx) {
  return {
    operation: "BatchPutItem",
    tables: {
      Posts: ctx.args.posts.map((post) => util.dynamodb.toMapValues(post)),
    },
  };
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
```

```
}
```

Naviguez vers leRequêtespage duAWS AppSyncconsole et exécutez ce qui suitbatchAddmutation :

```
mutation add {
    batchAdd(posts:[{
            id: 1 title: "Running in the Park"},{
            id: 2 title: "Playing fetch"
        }]){
            id
            title
    }
}
```

Vous devriez voir les résultats imprimés à l'écran ; cela peut être validé en consultant la console DynamoDB pour rechercher les valeurs écrites dansPoststable.

Ensuite, répétez le processus consistant à joindre un résolveur, sauf pourQuery.batchGetchamp à l'aide duPoststable comme source de données. Remplacez les gestionnaires par le code ci-dessous. Ce processus prend automatiquement chaque élément du type GraphQL `ids:[]` et crée une mappe, qui est nécessaire pour l'opération `BatchGetItem` :

```
import { util } from "@aws-appsync/utils";

export function request(ctx) {
  return {
    operation: "BatchGetItem",
    tables: {
      Posts: {
        keys: ctx.args.ids.map((id) => util.dynamodb.toMapValues({ id })),
        consistentRead: true,
      },
    },
  };
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
}
```

Maintenant, revenez auRequêtespage duAWS AppSyncconsole et exécutez ce qui suitbatchGetrequête :

```
query get {
    batchGet(ids:[1,2,3]){
        id
        title
    }
}
```

Vous devez obtenir les résultats des deux valeurs `id` que vous avez ajoutées précédemment. Notez qu'unnullune valeur a été renvoyée pouridavec une valeur de3. C'est parce qu'il n'y avait aucun enregistrement dans votrePoststable avec cette valeur pour le moment. Notez également queAWS AppSyncrenvoie les résultats dans le même ordre que les clés passées à la requête, ce qui est une fonctionnalité supplémentaire quiAWS AppSyncagit en votre nom. Donc, si vous passez à`batchGet(ids:[1,3,2])`, vous verrez que la commande a changé. Vous saurez également quel `id` a renvoyé une valeur `null`.

Enfin, attachez un autre résolveur au`Mutation.batchDelete`champ à l'aide du`Posts`table comme source de données. Remplacez les gestionnaires par le code ci-dessous. Ce processus prend automatiquement chaque élément du type GraphQL `ids:[]` et crée une mappe, qui est nécessaire pour l'opération `BatchGetItem` :

```
import { util } from "@aws-appsync/utils";

export function request(ctx) {
  return {
    operation: "BatchDeleteItem",
    tables: {
      Posts: ctx.args.ids.map((id) => util.dynamodb.toMapValues({ id })),
    },
  };
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
}
```

Maintenant, revenez auRequêtespage duAWS AppSyncconsole et exécutez ce qui suitbatchDeletemutation :

```
mutation delete {
    batchDelete(ids:[1,2]){ id }
}
```

Les enregistrements contenant les id 1 et 2 doivent désormais avoir été supprimés. Si vous exécutez à nouveau la requête batchGet() à partir de l'état précédent, le résultat renvoyé devrait être null.

Lot multi-tables

AWS AppSyncvous permet également d'effectuer des opérations par lots sur plusieurs tables. Créons un application plus complexe. Imaginez que nous sommes en train de créer une application de santé pour animaux de compagnie dans laquelle des capteurs signalent l'emplacement et la température corporelle de l'animal. Les capteurs sont alimentés par piles et tentent de se connecter au réseau toutes les deux ou trois minutes. Lorsqu'un capteur établit une connexion, il envoie ses relevés à notreAWS AppSyncAPI. Des déclencheurs analysent alors les données afin de pouvoir transmettre un tableau de bord au propriétaire de l'animal. Concentrons-nous sur la représentation des interactions entre le capteur et le magasin de données backend.

Dans leAWS AppSyncconsole, choisissezCréer une API,API GraphQL, etDesign à partir de zéro. Donnez un nom à votre APIMultiBatchTutorial API, choisissezSuivant, et sur leSpécifier les ressources GraphQLétape, choisissezCréez des ressources GraphQL ultérieurementet cliquezSuivant. Vérifiez vos informations et créez l'API. Accédez auSchémapuis collez et enregistrez le schéma suivant :

```
type Mutation {
    # Register a batch of readings
    recordReadings(tempReadings: [TemperatureReadingInput], locReadings:
 [LocationReadingInput]): RecordResult
    # Delete a batch of readings
    deleteReadings(tempReadings: [TemperatureReadingInput], locReadings:
 [LocationReadingInput]): RecordResult
}

type Query {
    # Retrieve all possible readings recorded by a sensor at a specific time
    getReadings(sensorId: ID!, timestamp: String!): [SensorReading]
```

```
}

type RecordResult {
    temperatureReadings: [TemperatureReading]
    locationReadings: [LocationReading]
}

interface SensorReading {
    sensorId: ID!
    timestamp: String!
}

# Sensor reading representing the sensor temperature (in Fahrenheit)
type TemperatureReading implements SensorReading {
    sensorId: ID!
    timestamp: String!
    value: Float
}

# Sensor reading representing the sensor location (lat,long)
type LocationReading implements SensorReading {
    sensorId: ID!
    timestamp: String!
    lat: Float
    long: Float
}

input TemperatureReadingInput {
    sensorId: ID!
    timestamp: String
    value: Float
}

input LocationReadingInput {
    sensorId: ID!
    timestamp: String
    lat: Float
    long: Float
}
```

Nous devons créer deux tables DynamoDB :

- `locationReadings`enregistrera les relevés de position du capteur.

- `temperatureReadings`enregistrera les relevés de température du capteur.

Les deux tables partageront la même structure de clé primaire :`sensorId (String)`comme clé de partition et`timestamp (String)`comme clé de tri.

ChoisissezCréation de ressourcesen haut de page. ChoisissezUtiliser le type existantet sélectionnez`locationReadings`type. Donnez un nom à votre table`locationReadings`. Assurez-vous queClé primaireest défini sur`sensorId`et la clé de tri pour`timestamp`. DésélectionnerGénérez automatiquement GraphQL(vous fournirez votre propre code), puis sélectionnezCréez. Répétez ce processus pour`temperatureReadings`en utilisant le`temperatureReadings`comme type et nom de table. Utilisez les mêmes touches que ci-dessus.

Vos nouvelles tables contiendront les rôles générés automatiquement. Vous devez encore ajouter quelques autorisations à ces rôles. Accédez auSources de donnéespage et choisissez`locationReadings`. SousSélectionnez un rôle existant, vous pouvez voir le rôle. Prenez note du rôle (qui devrait ressembler à`appsync-ds-ddb-aaabbbcccddd-locationReadings`) puis accédez à la console IAM (https://console.aws.amazon.com/iam/). Dans la console IAM, choisissezRôles, puis choisissez votre rôle dans le tableau. Dans votre rôle, sousPolitiques d'autorisations, cliquez sur le bouton »+« à côté de la politique (doit avoir un nom similaire au nom du rôle). ChoisissezModifieren haut du pliable lorsque la politique apparaît. Vous devez ajouter des autorisations à cette politique. Cela ressemblera à ceci :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem",
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:region:account:table/locationReadings",
                "arn:aws:dynamodb:region:account:table/locationReadings/*",
```

```
                "arn:aws:dynamodb:region:account:table/temperatureReadings",
                "arn:aws:dynamodb:region:account:table/temperatureReadings/*"
            ]
        }
    ]
}
```

ChoisissezSuivant, puisEnregistrer les modifications. Répétez cette procédure pour`temperatureReadings`source de données utilisant le même extrait de politique ci-dessus.

BatchPutItem- Enregistrement des lectures des capteurs

Nos capteurs doivent être en mesure d'envoyer leurs relevés une fois qu'ils sont connectés à Internet. Le champ GraphQL `Mutation.recordReadings` est l'API qu'ils utilisent à cet effet. Nous devrons ajouter un résolveur à ce champ.

Dans leAWS AppSyncconsolesSchémapage, sélectionnezJoindreà côté du`Mutation.recordReadings`champ. Sur l'écran suivant, créez votre résolveur à l'aide du`locationReadings`table comme source de données.

Après avoir créé votre résolveur, remplacez les gestionnaires par le code suivant dans l'éditeur. Ce`BatchPutItem`cette opération nous permet de spécifier plusieurs tables :

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { locReadings, tempReadings } = ctx.args
 const locationReadings = locReadings.map((loc) => util.dynamodb.toMapValues(loc))
 const temperatureReadings = tempReadings.map((tmp) => util.dynamodb.toMapValues(tmp))

 return {
  operation: 'BatchPutItem',
  tables: {
   locationReadings,
   temperatureReadings,
  },
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.appendError(ctx.error.message, ctx.error.type)
```

```
 }
 return ctx.result.data
}
```

Les opérations par lots peuvent renvoyer à la fois des erreurs et des résultats à la suite de l'appel. Dans ce cas, nous pouvons effectuer certaines opérations de traitement des erreurs supplémentaires.

> **Note**
>
> L'utilisation de`utils.appendError()`est similaire au`util.error()`, avec la principale différence qu'il n'interrompt pas l'évaluation du gestionnaire de demandes ou de réponses. Il signale plutôt qu'une erreur s'est produite dans le champ, mais permet d'évaluer le gestionnaire et, par conséquent, de renvoyer les données à l'appelant. Nous vous recommandons d'utiliser`utils.appendError()`lorsque votre application doit renvoyer des résultats partiels.

Enregistrez le résolveur et accédez auRequêtespage dans leAWS AppSyncconsole. Nous pouvons maintenant envoyer des relevés de capteurs.

Exécutez la mutation suivante :

```
mutation sendReadings {
  recordReadings(
    tempReadings: [
      {sensorId: 1, value: 85.5, timestamp: "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, value: 85.7, timestamp: "2018-02-01T17:21:06.000+08:00"},
      {sensorId: 1, value: 85.8, timestamp: "2018-02-01T17:21:07.000+08:00"},
      {sensorId: 1, value: 84.2, timestamp: "2018-02-01T17:21:08.000+08:00"},
      {sensorId: 1, value: 81.5, timestamp: "2018-02-01T17:21:09.000+08:00"}
    ]
    locReadings: [
      {sensorId: 1, lat: 47.615063, long: -122.333551, timestamp:
 "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, lat: 47.615163, long: -122.333552, timestamp:
 "2018-02-01T17:21:06.000+08:00"},
      {sensorId: 1, lat: 47.615263, long: -122.333553, timestamp:
 "2018-02-01T17:21:07.000+08:00"},
      {sensorId: 1, lat: 47.615363, long: -122.333554, timestamp:
 "2018-02-01T17:21:08.000+08:00"},
```

```
      {sensorId: 1, lat: 47.615463, long: -122.333555, timestamp:
 "2018-02-01T17:21:09.000+08:00"}
    ]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}
```

Nous avons envoyé dix relevés de capteurs pour une mutation, les résultats étant répartis sur deux tableaux. Utilisez la console DynamoDB pour vérifier que les données apparaissent à la fois dans`locationReadings`et`temperatureReadings`tables.

BatchDeleteItem- Suppression des relevés du capteur

De même, nous devrions également être en mesure de supprimer des lots de relevés de capteurs. Nous allons utiliser le champ GraphQL `Mutation.deleteReadings` à cet effet. Dans leAWS AppSyncconsolesSchémapage, sélectionnezJoindreà côté du`Mutation.deleteReadings`champ. Sur l'écran suivant, créez votre résolveur à l'aide du`locationReadings`table comme source de données.

Après avoir créé votre résolveur, remplacez les gestionnaires de l'éditeur de code par l'extrait ci-dessous. Dans ce résolveur, nous utilisons un mappeur de fonctions auxiliaires qui extrait`sensorId`et le`timestamp`à partir des entrées fournies.

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { locReadings, tempReadings } = ctx.args
 const mapper = ({ sensorId, timestamp }) => util.dynamodb.toMapValues({ sensorId,
 timestamp })

 return {
```

```
  operation: 'BatchDeleteItem',
  tables: {
   locationReadings: locReadings.map(mapper),
   temperatureReadings: tempReadings.map(mapper),
  },
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.appendError(ctx.error.message, ctx.error.type)
 }
 return ctx.result.data
}
```

Enregistrez le résolveur et accédez auRequêtespage dans leAWS AppSyncconsole. Supprimons maintenant quelques mesures du capteur.

Exécutez la mutation suivante :

```
mutation deleteReadings {
  # Let's delete the first two readings we recorded
  deleteReadings(
    tempReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]
    locReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}
```

> **Note**
>
> Contrairement à l'opération `DeleteItem`, l'élément complètement supprimé n'est pas renvoyé dans la réponse. Seule la clé passée est renvoyée. Pour en savoir plus, consultez leBatchDeleteItemdansJavaScriptréférence de fonction de résolution pour DynamoDB.

Vérifiez via la console DynamoDB que ces deux lectures ont été supprimées du`locationReadings`et`temperatureReadings`tables.

BatchGetItem- Récupérez les lectures

Une autre opération courante de notre application consiste à récupérer les mesures d'un capteur à un moment précis. Nous allons joindre un résolveur au champ GraphQL `Query.getReadings` dans notre schéma. Dans leAWS AppSyncconsolesSchémapage, sélectionnezJoindreà côté du`Query.getReadings`champ. Sur l'écran suivant, créez votre résolveur à l'aide du`locationReadings`table comme source de données.

Utilisons le code suivant :

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const keys = [util.dynamodb.toMapValues(ctx.args)]
 const consistentRead = true
 return {
  operation: 'BatchGetItem',
  tables: {
   locationReadings: { keys, consistentRead },
   temperatureReadings: { keys, consistentRead },
  },
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.appendError(ctx.error.message, ctx.error.type)
 }
 const { locationReadings: locs, temperatureReadings: temps } = ctx.result.data

 return [
```

```
  ...locs.map((l) => ({ ...l, __typename: 'LocationReading' })),
  ...temps.map((t) => ({ ...t, __typename: 'TemperatureReading' })),
 ]
}
```

Enregistrez le résolveur et accédez auRequêtespage dans leAWS AppSyncconsole. Maintenant, récupérons les relevés de nos capteurs.

Exécutez la requête suivante :

```
query getReadingsForSensorAndTime {
  # Let's retrieve the very first two readings
  getReadings(sensorId: 1, timestamp: "2018-02-01T17:21:06.000+08:00") {
    sensorId
    timestamp
    ...on TemperatureReading {
      value
    }
    ...on LocationReading {
      lat
      long
    }
  }
}
```

Nous avons démontré avec succès l'utilisation des opérations par lots de DynamoDB en utilisantAWS AppSync.

Gestion des erreurs

DansAWS AppSync, les opérations sur les sources de données peuvent parfois renvoyer des résultats partiels. Le terme résultats partiels est le terme que nous allons utiliser pour désigner une sortie d'opération composée de données et d'une erreur. Dans la mesure où la gestion des erreurs est intrinsèquement spécifique à l'application,AWS AppSyncvous donne la possibilité de gérer les erreurs dans le gestionnaire de réponses. L'erreur d'appel du résolveur, le cas échéant, est disponible depuis le contexte sous la forme `ctx.error`. Les erreurs d'appel incluent toujours un message et un type, accessibles sous la forme des propriétés `ctx.error.message` et `ctx.error.type`. Dans le gestionnaire de réponses, vous pouvez gérer les résultats partiels de trois manières :

1. Avalez l'erreur d'invocation en renvoyant simplement des données.

2. Déclencher une erreur (en utilisant`util.error(...)`) en arrêtant l'évaluation du gestionnaire, qui ne renverra aucune donnée.
3. Ajouter une erreur (en utilisant`util.appendError(...)`) et renvoient également des données.

Démontrons chacun des trois points ci-dessus avec les opérations par lots de DynamoDB.

Opérations par lots DynamoDB

Dans le cas des opérations par lots DynamoDB, il est possible qu'un lot ne soit exécuté que partiellement. En d'autres termes, il est possible que certains des éléments ou des clés demandés ne soient pas traités. SiAWS AppSyncest incapable de terminer un lot, des articles non traités et une erreur d'invocation sera définie dans le contexte.

Nous allons mettre en œuvre la gestion des erreurs à l'aide de la configuration de champ `Query.getReadings` de l'opération `BatchGetItem` provenant de la section précédente de ce didacticiel. Cette fois, nous allons supposer que, lors de l'exécution du champ `Query.getReadings`, la table DynamoDB `temperatureReadings` a dépassé le débit alloué. DynamoDB a généré un`ProvisionedThroughputExceededException`lors de la deuxième tentative par`AWS AppSync`pour traiter les éléments restants du lot.

Le JSON suivant représente le contexte sérialisé après l'appel par lots de DynamoDB mais avant l'appel du gestionnaire de réponses :

```
{
  "arguments": {
    "sensorId": "1",
    "timestamp": "2018-02-01T17:21:05.000+08:00"
  },
  "source": null,
  "result": {
    "data": {
      "temperatureReadings": [
        null
      ],
      "locationReadings": [
        {
          "lat": 47.615063,
          "long": -122.333551,
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
```

```
        }
      ]
    },
    "unprocessedKeys": {
      "temperatureReadings": [
        {
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ],
      "locationReadings": []
    }
  },
  "error": {
    "type": "DynamoDB:ProvisionedThroughputExceededException",
    "message": "You exceeded your maximum allowed provisioned throughput for a table or
 for one or more global secondary indexes. (...)"
  },
  "outErrors": []
}
```

Quelques points à noter concernant le contexte :

- L'erreur d'invocation a été définie dans le contexte à`ctx.error`parAWS AppSync, et le type d'erreur a été défini sur`DynamoDB:ProvisionedThroughputExceededException`.
- Les résultats sont cartographiés par tableau ci-dessous`ctx.result.data`même si une erreur est présente.
- Les clés qui n'ont pas été traitées sont disponibles sur`ctx.result.data.unprocessedKeys`. Ici,AWS AppSyncn'a pas pu récupérer l'élément avec la clé (SensorID:1, Timestamp:2018-02-01T 17:21:05.000 + 08:00) en raison d'un débit de table insuffisant.

> **Note**
>
> Pour `BatchPutItem`, la valeur est `ctx.result.data.unprocessedItems`. Pour `BatchDeleteItem`, la valeur est `ctx.result.data.unprocessedKeys`.

Nous allons traiter cette erreur de trois façons différentes.

1. Digestion de l'erreur d'appel

Le renvoi des données sans gestion de l'erreur d'appel se traduit par une digestion de l'erreur, ce qui permet au résultat du champ GraphQL donné d'être toujours réussi.

Le code que nous écrivons est familier et se concentre uniquement sur les données de résultat.

Gestionnaire de réponses

```
export function response(ctx) {
  return ctx.result.data
}
```

Réponse GraphQL

```
{
  "data": {
    "getReadings": [
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "lat": 47.615063,
        "long": -122.333551
      },
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  }
}
```

Aucune erreur n'est ajoutée à la réponse d'erreur car l'action n'a porté que sur les données.

2. Génération d'une erreur pour annuler l'exécution du gestionnaire de réponses

Lorsque les échecs partiels doivent être traités comme des échecs complets du point de vue du client, vous pouvez interrompre l'exécution du gestionnaire de réponses pour empêcher le renvoi de données. La méthode d'utilitaire `util.error(...)` permet d'obtenir exactement ce comportement.

Code du gestionnaire de réponses

```
export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type, null,
 ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}
```

Réponse GraphQL

```
{
  "data": {
    "getReadings": null
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
        }
      ],
      "message": "You exceeded your maximum allowed provisioned throughput for a table
 or for one or more global secondary indexes. (...)"
    }
  ]
}
```

Même si certains résultats peuvent avoir été renvoyés par l'opération de traitement par lots DynamoDB, nous avons choisi de déclencher une erreur se traduisant par une valeur null pour le champ GraphQL `getReadings` et l'erreur a été ajoutée au bloc d'erreurs de la réponse GraphQL.

3. Ajout d'une erreur pour renvoyer à la fois les données et les erreurs

Dans certains cas, afin d'offrir une meilleure expérience utilisateur, les applications peuvent renvoyer des résultats partiels et informer leurs clients des éléments non traités. Les clients peuvent choisir d'implémenter une nouvelle tentative ou de renvoyer l'erreur à l'utilisateur final. Le`util.appendError(...)`est la méthode utilitaire qui permet ce comportement en permettant au concepteur de l'application d'ajouter des erreurs au contexte sans interférer avec l'évaluation du gestionnaire de réponses. Après avoir évalué le gestionnaire de réponses,AWS AppSynctraitera toutes les erreurs de contexte en les ajoutant au bloc d'erreurs de la réponse GraphQL.

Code du gestionnaire de réponses

```
export function response(ctx) {
  if (ctx.error) {
    util.appendError(ctx.error.message, ctx.error.type, null,
 ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}
```

Nous avons transmis à la fois l'erreur d'invocation etun`processedKeys`élément à l'intérieur du bloc d'erreurs de la réponse GraphQL. Le`getReadings`le champ renvoie également des données partielles provenant du`locationReadings`tableau comme vous pouvez le voir dans la réponse ci-dessous.

Réponse GraphQL

```
{
  "data": {
    "getReadings": [
      null,
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
```

```
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
        }
      ],
      "message": "You exceeded your maximum allowed provisioned throughput for a table
 or for one or more global secondary indexes. (...)"
    }
  ]
}
```

Tutoriel : résolveurs HTTP

AWS AppSyncvous permet d'utiliser des sources de données prises en charge (c'est-à-direAWS Lambda, Amazon DynamoDB, AmazonOpenSearchService, ou Amazon Aurora) pour effectuer diverses opérations, en plus de tout point de terminaison HTTP arbitraire pour résoudre les champs GraphQL. Dès que vos points de terminaison HTTP sont disponibles, vous pouvez vous y connecter à l'aide d'une source de données. Ensuite, vous pouvez configurer un résolveur dans le schéma GraphQL pour effectuer des opérations telles que des requêtes, des mutations et des abonnements. Ce didacticiel vous présente certains exemples courants.

Dans ce didacticiel, vous utilisez une API REST (créée à l'aide d'Amazon API Gateway et de Lambda) avec unAWS AppSyncPoint de terminaison GraphQL.

Création d'une API REST

Vous pouvez utiliser le modèle AWS CloudFormation suivant pour configurer un point de terminaison REST qui fonctionne pour ce didacticiel :

La pile AWS CloudFormation exécute les étapes suivantes :

1. Elle configure une fonction Lambda qui contient la logique métier de votre microservice.
2. Configure une API REST API Gateway avec la combinaison point de terminaison, méthode et type de contenu suivante :

Chemin de ressource API	Méthode HTTP	Type de contenu pris en charge
/v1/users	POST	application/json
/v1/users	GET	application/json
/v1/users/1	GET	application/json
/v1/users/1	PUT	application/json
/v1/users/1	DELETE	application/json

Création de votre API GraphQL

Pour créer l'API GraphQL dansAWS AppSync:

1. Ouvrez leAWS AppSyncconsole et choisissezCréation d'une API.
2. ChoisissezAPI GraphQLpuis choisissezDesign à partir de zéro. Choisissez Suivant.
3. Pour le nom de l'API, saisissez `UserData`. Choisissez Suivant.
4. Sélectionnez `Create GraphQL resources later`. Choisissez Suivant.
5. Passez en revue vos entrées et choisissezCréation d'une API.

LeAWS AppSyncla console crée une nouvelle API GraphQL pour vous en utilisant le mode d'authentification par clé d'API. Vous pouvez utiliser la console pour configurer davantage votre API GraphQL et exécuter des requêtes.

Création d'un schéma GraphQL

Maintenant que vous avez une API GraphQL, nous allons créer un schéma GraphQL. Dans leSchémaéditeur dans leAWS AppSyncconsole, utilisez l'extrait ci-dessous :

```
type Mutation {
    addUser(userInput: UserInput!): User
    deleteUser(id: ID!): User
}

type Query {
    getUser(id: ID): User
    listUser: [User!]!
}

type User {
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
}

input UserInput {
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
}
```

Configuration de votre source de données HTTP

Pour configurer votre source de données HTTP, procédez comme suit :

1. Dans leSources de donnéespage dans votreAWS AppSyncAPI GraphQL, choisissezCréation d'une source de données.
2. Entrez un nom pour la source de données tel que`HTTP_Example`.
3. DansType de source de données, choisissezPoint de terminaison HTTP.
4. Définissez le point de terminaison sur le point de terminaison API Gateway créé au début du didacticiel. Vous pouvez trouver votre point de terminaison généré par une pile si vous accédez à la console Lambda et trouvez votre application sousDemandes. Dans les paramètres de votre application, vous devriez voir unPoint de terminaison APIqui sera votre point de terminaison dansAWS AppSync. Assurez-vous de ne pas inclure le nom de l'étape dans le point de terminaison. Par exemple, si votre point de terminaison était`https://aaabbbcccd.execute-api.us-east-1.amazonaws.com/v1`, vous saisiriez`https://aaabbbcccd.execute-api.us-east-1.amazonaws.com`.

> **Note**
>
> À l'heure actuelle, seuls les points de terminaison publics sont pris en charge parAWS AppSync.
> Pour plus d'informations sur les autorités de certification reconnues par leAWS AppSyncservice, voirAutorités de certification (CA) reconnues parAWS AppSyncpour les points de terminaison HTTPS.

Configuration des résolveurs

Au cours de cette étape, vous allez connecter la source de données HTTP au`getUser`et`addUser`requêtes.

Pour configurer le`getUser`résolveur :

1. Dans votreAWS AppSyncAPI GraphQL, choisissezSchémaonglet.
2. À droite duSchémaéditeur, dans leRésolveursvolet et sous leRequêtetapez, trouvez le`getUser`remplissez le champ et choisissezJoindre.
3. Conservez le type de résolveur à`Unit`et le temps d'exécution pour`APPSYNC_JS`.
4. DansNom de la source de données, choisissez le point de terminaison HTTP que vous avez créé précédemment.

5. Sélectionnez Create (Créer).
6. Dans leRésolveuréditeur de code, ajoutez l'extrait suivant comme gestionnaire de requêtes :

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 return {
  version: '2018-05-29',
  method: 'GET',
  params: {
   headers: {
    'Content-Type': 'application/json',
   },
  },
  resourcePath: `/v1/users/${ctx.args.id}`,
 }
}
```

7. Ajoutez l'extrait suivant en tant que gestionnaire de réponses :

```
export function response(ctx) {
 const { statusCode, body } = ctx.result
 // if response is 200, return the response
 if (statusCode === 200) {
  return JSON.parse(body)
 }
 // if response is not 200, append the response to error block.
 util.appendError(body, statusCode)
}
```

8. Choisissez l'onglet Requête et exécutez la requête suivante :

```
query GetUser{
    getUser(id:1){
        id
        username
    }
}
```

Cela doit renvoyer la réponse suivante :

```
{
```

```
    "data": {
        "getUser": {
            "id": "1",
            "username": "nadia"
        }
    }
}
```

Pour configurer leaddUserrésolveur :

1. Choisissez l'onglet Schéma.
2. À droite duSchémaéditeur, dans leRésolveursvolet et sous leRequêtetapez, trouvez leaddUserremplissez le champ et choisissezJoindre.
3. Conservez le type de résolveur àUnitet le temps d'exécution pourAPPSYNC_JS.
4. DansNom de la source de données, choisissez le point de terminaison HTTP que vous avez créé précédemment.
5. Sélectionnez Create (Créer).
6. Dans leRésolveuréditeur de code, ajoutez l'extrait suivant comme gestionnaire de requêtes :

```
export function request(ctx) {
    return {
        "version": "2018-05-29",
        "method": "POST",
        "resourcePath": "/v1/users",
        "params":{
            "headers":{
                "Content-Type": "application/json"
            },
        "body": ctx.args.userInput
        }
    }
}
```

7. Ajoutez l'extrait suivant en tant que gestionnaire de réponses :

```
export function response(ctx) {
    if(ctx.error) {
        return util.error(ctx.error.message, ctx.error.type)
    }
```

```
    if (ctx.result.statusCode == 200) {
        return ctx.result.body
    } else {
        return util.appendError(ctx.result.body, "ctx.result.statusCode")
    }
}
```

8. Choisissez l'onglet Requête et exécutez la requête suivante :

```
mutation addUser{
    addUser(userInput:{
        id:"2",
        username:"shaggy"
    }){
        id
        username
    }
}
```

Si vous exécutez le`getUser`Encore une fois, elle devrait renvoyer la réponse suivante :

```
{
    "data": {
        "getUser": {
        "id": "2",
        "username": "shaggy"
        }
    }
}
```

InvoquerAWSDes services

Vous pouvez utiliser des résolveurs HTTP pour configurer une interface d'API GraphQL pourAWSservices. Requêtes HTTP destinées àAWSdoit être signé avec leProcessus de signature version 4de sorte queAWSpeut identifier qui les a envoyés.AWS AppSynccalcule la signature en votre nom lorsque vous associez un rôle IAM à la source de données HTTP.

Vous fournissez deux composants supplémentaires à invoquerAWSservices avec résolveurs HTTP :

- Un rôle IAM autorisé à appeler leAWSAPI de service

- La configuration de signature dans la source de données

Par exemple, si vous souhaitez appeler leListGraphqlApisopérationavec les résolveurs HTTP, vous devez d'abordcréer un rôle IAMcetteAWS AppSyncsuppose que la politique suivante est jointe :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "appsync:ListGraphqlApis"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

Créez ensuite la source de données HTTP pourAWS AppSync. Dans cet exemple, vous appelezAWS AppSyncdans la région de l'Ouest des États-Unis (Oregon). Configurez la configuration HTTP suivante dans un fichier nommé `http.json`, qui inclut la région de signature et le nom du service :

```
{
    "endpoint": "https://appsync.us-west-2.amazonaws.com/",
    "authorizationConfig": {
        "authorizationType": "AWS_IAM",
        "awsIamConfig": {
            "signingRegion": "us-west-2",
            "signingServiceName": "appsync"
        }
    }
}
```

Ensuite, utilisez leAWS CLIpour créer la source de données avec un rôle associé comme suit :

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

Lorsque vous attachez un résolveur au champ du schéma, utilisez le modèle de mappage de demandes suivant pour appelerAWS AppSync:

```
{
    "version": "2018-05-29",
    "method": "GET",
    "resourcePath": "/v1/apis"
}
```

Lorsque vous exécutez une requête GraphQL pour cette source de données,AWS AppSyncsigne la demande en utilisant le rôle que vous avez indiqué et inclut la signature dans la demande. La requête renvoie une liste deAWS AppSyncDes API GraphQL dans votre compte dans lesquellesAWSRégion.

Tutoriel : Aurora PostgreSQL avec API de données

AWS AppSyncfournit une source de données pour exécuter des instructions SQL sur des clusters Amazon Aurora activés par une API de données. Vous pouvez utiliser des AWS AppSync résolveurs pour exécuter des instructions SQL sur l'API de données à l'aide de requêtes GraphQL, de mutations et d'abonnements.

> **Note**
>
> Ce didacticiel utilise la région `US-EAST-1`.

Création de clusters

Avant d'ajouter une source de données Amazon RDSAWS AppSync, activez d'abord une API de données sur un cluster Aurora Serverless. Vous devez également configurer un secret à l'aide deAWS Secrets Manager. Pour créer un cluster Aurora Serverless, vous pouvez utiliser : AWS CLI

```
aws rds create-db-cluster \
    --db-cluster-identifier appsync-tutorial \
    --engine aurora-postgresql --engine-version 13.11 \
    --engine-mode serverless \
    --master-username USERNAME \
    --master-user-password COMPLEX_PASSWORD
```

Cela renverra un ARN pour le cluster. Vous pouvez vérifier l'état de votre cluster à l'aide de la commande suivante :

```
aws rds describe-db-clusters \
    --db-cluster-identifier appsync-tutorial \
    --query "DBClusters[0].Status"
```

Créez un secret via la AWS Secrets Manager console ou AWS CLI avec un fichier d'entrée tel que le suivant en utilisant le `USERNAME` et `COMPLEX_PASSWORD` de l'étape précédente :

```
{
    "username": "USERNAME",
    "password": "COMPLEX_PASSWORD"
}
```

Passez ceci en tant que paramètre à la CLI :

```
aws secretsmanager create-secret \
    --name appsync-tutorial-rds-secret \
    --secret-string file://creds.json
```

Cela renverra un ARN pour le secret. Prenez note de l'ARN de votre cluster Aurora Serverless et de Secret pour plus tard lors de la création d'une source de données dans la AWS AppSync console.

Activation de l'API de données

Une fois que le statut de votre cluster est `available` passé à, activez l'API de données en suivant la documentation Amazon RDS. L'API de données doit être activée avant de l'ajouter en tant que source de AWS AppSync données. Vous pouvez également activer l'API de données à l'aide de AWS CLI :

```
aws rds modify-db-cluster \
    --db-cluster-identifier appsync-tutorial \
    --enable-http-endpoint \
    --apply-immediately
```

Création de la base de données et de la table

Après avoir activé votre API de données, validez qu'elle fonctionne à l'aide de la `aws rds-data execute-statement` commande duAWS CLI. Cela garantit que votre cluster Aurora Serverless est

correctement configuré avant de l'ajouter à l'AWS AppSyncAPI. Créez d'abord une base de données TESTDB avec le `--sql` paramètre :

```
aws rds-data execute-statement \
    --resource-arn "arn:aws:rds:us-east-1:123456789012:cluster:appsync-tutorial" \
    --secret-arn "arn:aws:secretsmanager:us-east-1:123456789012:secret:appsync-
tutorial-rds-secret" \
    --sql "create DATABASE \"testdb\""
```

Si cela fonctionne sans erreur, ajoutez deux tables avec la `create table` commande :

```
 aws rds-data execute-statement \
    --resource-arn "arn:aws:rds:us-east-1:123456789012:cluster:appsync-tutorial" \
    --secret-arn "arn:aws:secretsmanager:us-east-1:123456789012:secret:appsync-
tutorial-rds-secret" \
    --database "testdb" \
    --sql 'create table public.todos (id serial constraint todos_pk primary key,
 description text not null, due date not null, "createdAt" timestamp default now());'

aws rds-data execute-statement \
    --resource-arn "arn:aws:rds:us-east-1:123456789012:cluster:appsync-tutorial" \
    --secret-arn "arn:aws:secretsmanager:us-east-1:123456789012:secret:appsync-
tutorial-rds-secret" \
    --database "testdb" \
    --sql 'create table public.tasks (id serial constraint tasks_pk primary key,
 description varchar, "todoId" integer not null constraint tasks_todos_id_fk references
 public.todos);'
```

Si tout fonctionne sans problème, vous pouvez désormais ajouter le cluster en tant que source de données dans votre API.

Création d'un schéma GraphQL

Maintenant que votre API de données Aurora Serverless s'exécute avec des tables configurées, nous allons créer un schéma GraphQL. Vous pouvez le faire manuellement, mais cela vous AWS AppSync permet de démarrer rapidement en important la configuration des tables depuis une base de données existante à l'aide de l'assistant de création d'API.

Pour commencer :

1. Dans la AWS AppSync console, choisissez Create API, puis Start with a Amazon Aurora cluster.

2. Spécifiez les détails de l'API, tels que le nom de l'API, puis sélectionnez votre base de données pour générer l'API.
3. Choisissez votre base de données. Si nécessaire, mettez à jour la région, puis choisissez votre cluster Aurora et votre base de données TESTDB.
4. Choisissez votre secret, puis choisissez Importer.
5. Une fois les tables découvertes, mettez à jour les noms des types. Changez `Todos` vers `Todo` et `Tasks` vers`Task`.
6. Prévisualisez le schéma généré en choisissant Aperçu du schéma. Votre schéma ressemblera à ceci :

```
type Todo {
  id: Int!
  description: String!
  due: AWSDate!
  createdAt: String
}

type Task {
  id: Int!
  todoId: Int!
  description: String
}
```

7. Pour le rôle, vous pouvez soit AWS AppSync créer un nouveau rôle, soit en créer un avec une politique similaire à celle ci-dessous :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "rds-data:ExecuteStatement",
            ],
            "Resource": [
                "arn:aws:rds:us-east-1:123456789012:cluster:appsync-tutorial",
                "arn:aws:rds:us-east-1:123456789012:cluster:appsync-tutorial:*"
            ]
        },
        {
```

```
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": [
                "arn:aws:secretsmanager:us-
east-1:123456789012:secret:your:secret:arn:appsync-tutorial-rds-secret",
                "arn:aws:secretsmanager:us-
east-1:123456789012:secret:your:secret:arn:appsync-tutorial-rds-secret:*"
            ]
        }
    ]
}
```

Notez que cette politique contient deux déclarations auxquelles vous accordez un accès aux rôles. La première ressource est votre cluster Aurora et la seconde est votre AWS Secrets Manager ARN.

Choisissez Next, passez en revue les détails de configuration, puis choisissez Create API. Vous disposez désormais d'une API entièrement opérationnelle. Vous pouvez consulter tous les détails de votre API sur la page Schéma.

Résolveurs pour RDS

Le flux de création d'API a automatiquement créé les résolveurs pour interagir avec nos types. Si vous regardez la page Schéma, vous trouverez les résolveurs nécessaires pour :

- Créez un `todo` via le `Mutation.createTodo` champ.
- Mettez à jour un `todo` via le `Mutation.updateTodo` champ.
- Supprimez un `todo` via le `Mutation.deleteTodo` champ.
- Obtenez-en un `todo` sur le `Query.getTodo` terrain.
- `todos`Répertoriez tout via le `Query.listTodos` champ.

Vous trouverez des champs et des résolveurs similaires attachés au `Task` type. Examinons de plus près certains des résolveurs.

Mutation.CreateToDo

Dans l'éditeur de schéma de la AWS AppSync console, sur le côté droit, choisissez à `testdb` côté de`createTodo(...): Todo`. Le code du résolveur utilise la `insert` fonction du `rds` module pour créer dynamiquement une instruction d'insertion qui ajoute des données à la `todos` table. Comme nous travaillons avec Postgres, nous pouvons tirer parti de `returning` cette instruction pour récupérer les données insérées.

Mettons à jour le résolveur pour spécifier correctement le `DATE` type du `due` champ :

```
import { util } from '@aws-appsync/utils';
import { insert, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/
rds';

export function request(ctx) {
    const { input } = ctx.args;
    // if a due date is provided, cast is as `DATE`
    if (input.due) {
        input.due = typeHint.DATE(input.due)
    }
    const insertStatement = insert({
        table: 'todos',
        values: input,
        returning: '*',
    });
    return createPgStatement(insertStatement)
}

export function response(ctx) {
    const { error, result } = ctx;
    if (error) {
        return util.appendError(
            error.message,
            error.type,
            result
        )
    }
    return toJsonObject(result)[0][0]
}
```

Enregistrez le résolveur. L'indice de type marque le type `due` correct dans notre objet d'entrée en tant que `DATE` type. Cela permet au moteur Postgres d'interpréter correctement la valeur. Ensuite, mettez

à jour votre schéma pour le supprimer `id` de l'`CreateTodo`entrée. Comme notre base de données Postgres peut renvoyer l'identifiant généré, nous pouvons nous y fier pour la création et le renvoi du résultat sous la forme d'une seule requête :

```
input CreateTodoInput {
    due: AWSDate!
    createdAt: String
    description: String!
}
```

Apportez la modification et mettez à jour votre schéma. Accédez à l'éditeur de requêtes pour ajouter un élément à la base de données :

```
mutation CreateTodo {
  createTodo(input: {description: "Hello World!", due: "2023-12-31"}) {
    id
    due
    description
    createdAt
  }
}
```

Vous obtenez le résultat :

```
{
  "data": {
    "createTodo": {
      "id": 1,
      "due": "2023-12-31",
      "description": "Hello World!",
      "createdAt": "2023-11-14 20:47:11.875428"
    }
  }
}
```

Query.ListToDos

Dans l'éditeur de schéma de la console, sur le côté droit, choisissez à `testdb` côté de`listTodos(id: ID!): Todo`. Le gestionnaire de demandes utilise la fonction utilitaire de sélection pour créer une demande de manière dynamique au moment de l'exécution.

```
export function request(ctx) {
    const { filter = {}, limit = 100, nextToken } = ctx.args;
    const offset = nextToken ? +util.base64Decode(nextToken) : 0;
    const statement = select({
        table: 'todos',
        columns: '*',
        limit,
        offset,
        where: filter,
    });
    return createPgStatement(statement)
}
```

Nous voulons filtrer `todos` en fonction de la `due` date. Mettons à jour le résolveur vers lequel convertir `due` `DATE` les valeurs. Mettez à jour la liste des importations et le gestionnaire de demandes :

```
import { util } from '@aws-appsync/utils';
import * as rds from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { filter: where = {}, limit = 100, nextToken } = ctx.args;
  const offset = nextToken ? +util.base64Decode(nextToken) : 0;

  // if `due` is used in a filter, CAST the values to DATE.
  if (where.due) {
    Object.entries(where.due).forEach(([k, v]) => {
      if (k === 'between') {
        where.due[k] = v.map((d) => rds.typeHint.DATE(d));
      } else {
        where.due[k] = rds.typeHint.DATE(v);
      }
    });
  }

  const statement = rds.select({
    table: 'todos',
    columns: '*',
    limit,
    offset,
    where,
  });
```

```
  return rds.createPgStatement(statement);
}

export function response(ctx) {
  const {
    args: { limit = 100, nextToken },
    error,
    result,
  } = ctx;
  if (error) {
    return util.appendError(error.message, error.type, result);
  }
  const offset = nextToken ? +util.base64Decode(nextToken) : 0;
  const items = rds.toJsonObject(result)[0];
  const endOfResults = items?.length < limit;
  const token = endOfResults ? null : util.base64Encode(`${offset + limit}`);
  return { items, nextToken: token };
}
```

Essayons la requête. Dans l'éditeur de requêtes :

```
query LIST {
  listTodos(limit: 10, filter: {due: {between: ["2021-01-01", "2025-01-02"]}}) {
    items {
      id
      due
      description
    }
  }
}
```

Mutation. Mise à jour à faire

Vous pouvez également `update` unTodo. Dans l'éditeur de requêtes, mettons à jour notre premier `Todo` élément de `id1`.

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits"}) {
    description
    due
    id
  }
```

```
}
```

Notez que vous devez spécifier `id` l'élément que vous mettez à jour. Vous pouvez également spécifier une condition pour ne mettre à jour qu'un élément répondant à des conditions spécifiques. Par exemple, il se peut que nous souhaitions modifier l'article uniquement si la description commence par `edits` :

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits: make a change"}, condition:
 {description: {beginsWith: "edits"}}) {
    description
    due
    id
  }
}
```

Tout comme nous avons géré nos `create` `list` opérations, nous pouvons mettre à jour notre résolveur pour transformer le due champ en un`DATE`. Enregistrez ces modifications dans `updateTodo` :

```
import { util } from '@aws-appsync/utils';
import * as rds from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: { id, ...values }, condition = {}, } = ctx.args;
  const where = { ...condition, id: { eq: id } };

  // if `due` is used in a condition, CAST the values to DATE.
  if (condition.due) {
    Object.entries(condition.due).forEach(([k, v]) => {
      if (k === 'between') {
        condition.due[k] = v.map((d) => rds.typeHint.DATE(d));
      } else {
        condition.due[k] = rds.typeHint.DATE(v);
      }
    });
  }

  // if a due date is provided, cast is as `DATE`
  if (values.due) {
    values.due = rds.typeHint.DATE(values.due);
  }
```

```
  const updateStatement = rds.update({
    table: 'todos',
    values,
    where,
    returning: '*',
  });
  return rds.createPgStatement(updateStatement);
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    return util.appendError(error.message, error.type, result);
  }
  return rds.toJsonObject(result)[0][0];
}
```

Maintenant, essayez une mise à jour avec une condition :

```
mutation UPDATE {
  updateTodo(
    input: {
        id: 1, description: "edits: make a change", due: "2023-12-12"},
    condition: {
        description: {beginsWith: "edits"}, due: {ge: "2023-11-08"}})
    {
          description
          due
          id
        }
}
```

Mutation.DeleteToDo

Vous pouvez `delete` `Todo` utiliser la `deleteTodo` mutation. Cela fonctionne comme la `updateTodo` mutation, et vous devez spécifier `id` l'élément que vous souhaitez supprimer :

```
mutation DELETE {
  deleteTodo(input: {id: 1}) {
    description
    due
    id
```

```
  }
}
```

Rédaction de requêtes personnalisées

Nous avons utilisé les utilitaires du `rds` module pour créer nos instructions SQL. Nous pouvons également écrire notre propre déclaration statique personnalisée pour interagir avec notre base de données. Tout d'abord, mettez à jour le schéma pour supprimer le `id` champ de l'`CreateTask`entrée.

```
input CreateTaskInput {
    todoId: Int!
    description: String
}
```

Créez ensuite quelques tâches. Une tâche possède une relation de clé étrangère avec `Todo` :

```
mutation TASKS {
  a: createTask(input: {todoId: 2, description: "my first sub task"}) { id }
  b:createTask(input: {todoId: 2, description: "another sub task"}) { id }
  c: createTask(input: {todoId: 2, description: "a final sub task"}) { id }
}
```

Créez un nouveau champ de votre `Query` type appelé `getTodoAndTasks` :

```
getTodoAndTasks(id: Int!): Todo
```

Ajoutez un `tasks` champ au `Todo` type :

```
type Todo {
    due: AWSDate!
    id: Int!
    createdAt: String
    description: String!
    tasks:TaskConnection
}
```

Enregistrez le schéma. Dans l'éditeur de schéma de la console, sur le côté droit, choisissez Attach Resolver for`getTodosAndTasks(id: Int!): Todo`. Choisissez votre source de données Amazon RDS. Mettez à jour votre résolveur avec le code suivant :

```
import { sql, createPgStatement,toJsonObject } from '@aws-appsync/utils/rds';

export function request(ctx) {
    return createPgStatement(
        sql`SELECT * from todos where id = ${ctx.args.id}`,
        sql`SELECT * from tasks where "todoId" = ${ctx.args.id}`);
}

export function response(ctx) {
    const result = toJsonObject(ctx.result);
    const todo = result[0][0];
    if (!todo) {
        return null;
    }
    todo.tasks = { items: result[1] };
    return todo;
}
```

Dans ce code, nous utilisons le modèle de `sql` balise pour écrire une instruction SQL à laquelle nous pouvons transmettre une valeur dynamique en toute sécurité lors de l'exécution. `createPgStatement`peut traiter jusqu'à deux requêtes SQL à la fois. Nous l'utilisons pour envoyer une requête pour notre `todo` et une autre pour notre`tasks`. Vous auriez pu le faire avec une `JOIN` déclaration ou toute autre méthode d'ailleurs. L'idée est de pouvoir écrire votre propre instruction SQL pour implémenter votre logique métier. Pour utiliser la requête dans l'éditeur de requêtes, nous pouvons essayer ceci :

```
query TodoAndTasks {
  getTodosAndTasks(id: 2) {
    id
    due
    description
    tasks {
      items {
        id
        description
      }
    }
  }
}
```

Supprimer votre cluster

> **Important**
>
> La suppression d'un cluster est définitive. Passez en revue votre projet de manière approfondie avant de réaliser cette action.

Pour supprimer votre cluster :

```
$ aws rds delete-db-cluster \
    --db-cluster-identifier appsync-tutorial \
    --skip-final-snapshot
```

Tutoriels sur le résolveur (VTL)

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Les sources de données et les résolveurs AWS AppSync traduisent les requêtes GraphQL et récupèrent les informations de vos ressources. AWS AWS AppSync prend en charge le provisionnement automatique et les connexions avec certains types de sources de données. AWS AppSync prend en charge Amazon DynamoDBAWS Lambda, les bases de données relationnelles (Amazon Aurora Serverless), OpenSearch Amazon Service et les points de terminaison HTTP en tant que sources de données. Vous pouvez utiliser une API GraphQL avec vos AWS ressources existantes ou créer des sources de données et des résolveurs. Cette section vous guide à travers ce processus en une série de didacticiels pour mieux comprendre comment les détails fonctionnent et affiner les options.

AWS AppSync utilise des modèles de mappage écrits en langage Apache Velocity Template Language (VTL) pour les résolveurs. Pour plus d'informations sur l'utilisation des modèles de mappage, consultez la référence des modèles de mappage Resolver. De plus amples informations sur l'utilisation de VTL sont disponibles dans le guide de programmation du modèle de mappage Resolver.

AWS AppSync prend en charge le provisionnement automatique des tables DynamoDB à partir d'un schéma GraphQL, comme décrit dans Provisionner à partir d'un schéma (facultatif) et Lancer un exemple de schéma. Vous pouvez également importer à partir d'une table DynamoDB existante qui va créer les schémas et connecter les résolveurs. Ceci est décrit dans Importer depuis Amazon DynamoDB (facultatif).

Rubriques

- Tutoriel : Résolveurs par lots DynamoDB
- Tutoriel : résolveurs de transactions DynamoDB
- Tutoriel : résolveurs HTTP
- Tutoriel : Aurora Serverless
- Tutoriel : Pipeline Resolvers
- Tutoriel : Delta Sync

Tutoriel : résolveurs DynamoDB

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Ce didacticiel explique comment importer vos propres tables Amazon DynamoDB et les connecter AWS AppSync à une API GraphQL.

Vous pouvez autoriser le AWS AppSync provisionnement des ressources DynamoDB en votre nom. Ou, si vous préférez, vous pouvez connecter vos tables existantes à un schéma GraphQL en créant une source de données et un résolveur. Dans les deux cas, vous serez en mesure de lire et d'écrire dans votre base de données DynamoDB via les instructions GraphQL, et de vous abonner aux données en temps réel.

Certaines étapes de configuration spécifiques doivent être effectuées pour convertir des instructions GraphQL en opérations DynamoDB, et pour reconvertir les réponses dans GraphQL. Ce didacticiel explique le processus de configuration par le biais de plusieurs scénarios et modèles d'accès aux données concrets.

Configuration de vos tables DynamoDB

Pour commencer ce didacticiel, vous devez d'abord suivre les étapes ci-dessous pour provisionner AWS les ressources.

1. Provisionnez les AWS ressources à l'aide du AWS CloudFormation modèle suivant dans la CLI :

```
aws cloudformation create-stack \
```

```
    --stack-name AWSAppSyncTutorialForAmazonDynamoDB \
    --template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/
dynamodb/AmazonDynamoDBCFTemplate.yaml \
    --capabilities CAPABILITY_NAMED_IAM
```

Vous pouvez également lancer la AWS CloudFormation pile suivante dans la région US-West 2 (Oregon) sur votre AWS compte.

Cela crée les éléments suivants :

- Une table DynamoDB `AppSyncTutorial-Post` appelée qui contiendra les données. `Post`
- Un rôle IAM et la stratégie gérée IAM associée pour autoriser AWS AppSync à interagir avec la table `Post`.

2. Pour plus de détails sur la pile et les ressources créées, exécutez la commande d'interface de ligne de commande suivante :

```
aws cloudformation describe-stacks --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

3. Pour supprimer ultérieurement les ressources, vous pouvez exécuter :

```
aws cloudformation delete-stack --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

Création de votre API GraphQL

Pour créer l'API GraphQL dans AWS AppSync :

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync .
   - Dans le tableau de bord des API, choisissez Create API.
2. Dans la fenêtre Personnalisez votre API ou importez depuis Amazon DynamoDB, choisissez Créer à partir de zéro.
   - Choisissez Démarrer à droite de la même fenêtre.
3. Dans le champ Nom de l'API, définissez le nom de l'API sur`AWSAppSyncTutorial`.
4. Sélectionnez Create (Créer).

La console AWS AppSync crée une nouvelle API GraphQL pour vous à l'aide du mode d'authentification de clé API. Vous pouvez utiliser la console pour configurer le reste de l'API GraphQL et exécuter des requêtes sur celle-ci jusqu'à la fin de ce didacticiel.

Définition d'une API de publication de base

Maintenant que vous avez créé une API AWS AppSync GraphQL, vous pouvez configurer un schéma de base qui permet la création, la récupération et la suppression de base des données de publication.

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync .
   - Dans le tableau de bord des API, choisissez l'API que vous venez de créer.
2. Dans la barre latérale, choisissez Schema.
   - Dans le volet Schéma, remplacez le contenu par le code suivant :

```
schema {
    query: Query
    mutation: Mutation
}

type Query {
    getPost(id: ID): Post
}

type Mutation {
    addPost(
        id: ID!
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}

type Post {
    id: ID!
    author: String
    title: String
    content: String
    url: String
```

```
    ups: Int!
    downs: Int!
    version: Int!
}
```

3. Choisissez Save (Enregistrer).

Ce schéma définit un type `Post` et des opérations pour ajouter et obtenir des objets `Post`.

Configuration de la source de données pour les tables DynamoDB

Liez ensuite les requêtes et les mutations définies dans le schéma à la table `AppSyncTutorial-Post` DynamoDB.

Tout d'abord, AWS AppSync doit être conscient de vos tables. Pour ce faire, vous devez configurer une source de données dans AWS AppSync :

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync .
   a. Dans le tableau de bord des API, choisissez votre API GraphQL.
   b. Dans la barre latérale, sélectionnez Sources de données.
2. Choisissez Create data source.
   a. Dans le champ Nom de la source de données, entrez`PostDynamoDBTable`.
   b. Pour le type de source de données, choisissez la table Amazon DynamoDB.
   c. Pour Région, choisissez US-WEST-2.
   d. Pour Nom de la table, choisissez la table AppSyncTutorialDynamoDB -Post.
   e. Créez un nouveau rôle IAM (recommandé) ou choisissez un rôle existant disposant de l'autorisation `lambda:invokeFunction` IAM. Les rôles existants nécessitent une politique de confiance, comme expliqué dans la section Joindre une source de données.

      Voici un exemple de politique IAM qui dispose des autorisations requises pour effectuer des opérations sur la ressource :

```
{
     "Version": "2012-10-17",
     "Statement": [
         {
             "Effect": "Allow",
```

```
            "Action": [ "lambda:invokeFunction" ],
            "Resource": [
                "arn:aws:lambda:us-west-2:123456789012:function:myFunction",
                "arn:aws:lambda:us-west-2:123456789012:function:myFunction:*"
            ]
        }
    ]
}
```

3. Sélectionnez Create (Créer).

Configuration du résolveur AddPost (DynamoDB) PutItem

Une fois AWS AppSync que vous avez pris connaissance de la table DynamoDB, vous pouvez la lier à des requêtes et à des mutations individuelles en définissant des résolveurs. Le premier résolveur que vous créez est le `addPost` résolveur, qui vous permet de créer une publication dans la table DynamoDB`AppSyncTutorial-Post`.

Un résolveur comprend les éléments suivants :

- L'emplacement dans le schéma GraphQL pour joindre le résolveur. Dans ce cas, vous configurez un résolveur sur le champ `addPost` sur le type `Mutation`. Ce résolveur est appelé lorsque l'appelant appelle `mutation { addPost(...){...} }`.
- La source de données à utiliser pour ce résolveur. Dans ce cas, vous souhaitez utiliser la source de données `PostDynamoDBTable` que vous avez définie précédemment, afin de pouvoir ajouter des entrées dans la table DynamoDB `AppSyncTutorial-Post`.
- Le modèle de mappage de demande. L'objectif du modèle de mappage de demande est de prendre la demande entrante à partir de l'appelant et de la convertir en instructions qu'AWS AppSync exécutera sur DynamoDB.
- Le modèle de mappage de réponse. La tâche du modèle de mappage de réponse est de prendre la réponse provenant de DynamoDB et de la reconvertir en quelque chose que GraphQL attend. Cela est utile si la forme des données dans DynamoDB est différente du type `Post` dans GraphQL, mais dans ce cas, ils ont la même forme, de sorte que vous transmettez simplement les données.

Pour configurer le résolveur :

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync .

a. Dans le tableau de bord des API, choisissez votre API GraphQL.

b. Dans la barre latérale, sélectionnez Sources de données.

2. Choisissez Create data source.

a. Dans le champ Nom de la source de données, entrez`PostDynamoDBTable`.

b. Pour le type de source de données, choisissez la table Amazon DynamoDB.

c. Pour Région, choisissez US-WEST-2.

d. Pour Nom de la table, choisissez la table AppSyncTutorialDynamoDB -Post.

e. Créez un nouveau rôle IAM (recommandé) ou choisissez un rôle existant disposant de l'autorisation `lambda:invokeFunction` IAM. Les rôles existants nécessitent une politique de confiance, comme expliqué dans la section Joindre une source de données.

Voici un exemple de politique IAM qui dispose des autorisations requises pour effectuer des opérations sur la ressource :

```
{
     "Version": "2012-10-17",
     "Statement": [
         {
             "Effect": "Allow",
             "Action": [ "lambda:invokeFunction" ],
             "Resource": [
                 "arn:aws:lambda:us-west-2:123456789012:function:myFunction",
                 "arn:aws:lambda:us-west-2:123456789012:function:myFunction:*"
             ]
         }
     ]
 }
```

3. Sélectionnez Create (Créer).

4. Choisissez l'onglet Schéma.

5. Dans le volet Types de données sur la droite, se trouve le champ addPost sur le type Mutation, choisissez ensuite Joindre.

6. Dans le menu Action, choisissez Update runtime, puis Unit Resolver (VTL uniquement).

7. Dans Nom de la source de données, choisissez PostDynamoDBTable.

8. Collez ce qui suit dans la section Configurer le modèle de mappage de demande :

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "attributeValues" : {
        "author" : $util.dynamodb.toDynamoDBJson($context.arguments.author),
        "title" : $util.dynamodb.toDynamoDBJson($context.arguments.title),
        "content" : $util.dynamodb.toDynamoDBJson($context.arguments.content),
        "url" : $util.dynamodb.toDynamoDBJson($context.arguments.url),
        "ups" : { "N" : 1 },
        "downs" : { "N" : 0 },
        "version" : { "N" : 1 }
    }
}
```

Remarque : un type est spécifié sur toutes les clés et valeurs d'attribut. Par exemple, vous définissez le champ `author` sur `{ "S" : "${context.arguments.author}" }`. La `S` partie indique à DynamoDB AWS AppSync et à DynamoDB que la valeur sera une valeur de chaîne. La valeur réelle est renseignée à partir de l'argument `author`. De la même façon, le champ `version` est un champ numérique, car il utilise `N` comme type. Enfin, vous initialisez également les champs `ups`, `downs` et `version`.

Pour ce didacticiel, vous avez indiqué que le `ID!` type GraphQL, qui indexe le nouvel élément inséré dans DynamoDB, fait partie des arguments du client. AWS AppSyncest livré avec un utilitaire de génération automatique d'identifiants appelé `$utils.autoId()` que vous auriez également pu utiliser sous la forme de`"id" : { "S" : "${$utils.autoId()}" }`. Ensuite, vous pourriez simplement laisser `id: ID!` hors de la définition de schéma de `addPost()` et il serait inséré automatiquement. Vous n'utiliserez pas cette technique dans ce didacticiel, mais vous devez la considérer comme une bonne pratique lorsque vous écrivez dans des tables DynamoDB.

Pour plus d'informations sur les modèles de mappage, consultez la documentation de référence Présentation des modèles de mappage des résolveurs. Pour plus d'informations sur le mappage des GetItem demandes, consultez la documentation de GetItemréférence. Pour plus d'informations sur les types, consultez la documentation de référence Système de types (mappage de demande).

9. Collez ce qui suit dans la section Configurer le modèle de mappage de réponse :

```
$utils.toJson($context.result)
```

Remarque : comme la forme des données de la table `AppSyncTutorial-Post` correspond exactement à la forme du type `Post` dans GraphQL, le modèle de mappage de réponse transmet simplement directement les résultats. Notez également que tous les exemples utilisés dans ce tutoriel utilisent le même modèle de mappage de réponse, si bien que vous ne créez qu'un fichier.

10. Choisissez Save (Enregistrer).

Appel de l'API pour ajouter une publication

Maintenant que le résolveur est configuré, il AWS AppSync peut traduire une `addPost` mutation entrante en une opération DynamoDB PutItem . Vous pouvez désormais exécuter une mutation pour placer quelque chose dans la table.

- Choisissez l'onglet Requêtes.
- Collez la mutation suivante dans le volet Requêtes.

```
mutation addPost {
  addPost(
    id: 123
    author: "AUTHORNAME"
    title: "Our first post!"
    content: "This is our first post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).

- Les résultats de la publication nouvellement créée doivent apparaître dans le volet des résultats à droite du volet de requête. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

Voici ce qui s'est produit :

- AWS AppSync a reçu une demande de `addPost` mutation.
- AWS AppSync a pris la demande et le modèle de mappage des demandes, et a généré un document de mappage des demandes. Celui-ci ressemblait à :

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : { "S" : "123" }
    },
    "attributeValues" : {
        "author": { "S" : "AUTHORNAME" },
        "title": { "S" : "Our first post!" },
        "content": { "S" : "This is our first post." },
        "url": { "S" : "https://aws.amazon.com/appsync/" },
        "ups" : { "N" : 1 },
        "downs" : { "N" : 0 },
        "version" : { "N" : 1 }
    }
}
```

- AWS AppSync a utilisé le document de mappage des demandes pour générer et exécuter une demande `PutItem` DynamoDB.
- AWS AppSync a pris les résultats de la `PutItem` requête et les a reconvertis en types GraphQL.

```
{
    "id" : "123",
    "author": "AUTHORNAME",
    "title": "Our first post!",
    "content": "This is our first post.",
    "url": "https://aws.amazon.com/appsync/",
    "ups" : 1,
    "downs" : 0,
    "version" : 1
}
```

- Ils ont été transmis via le document de mappage de réponse, qui les a transmis sans les changer.
- L'objet nouvellement créé a été renvoyé dans la réponse GraphQL.

Configuration du résolveur GetPost (DynamoDB) GetItem

Maintenant que vous pouvez ajouter des données à la table `AppSyncTutorial-Post` DynamoDB, vous devez configurer `getPost` la requête afin qu'elle puisse récupérer ces données de la table. `AppSyncTutorial-Post` Pour ce faire, vous configurez un autre résolveur.

- Choisissez l'onglet Schéma.
- Dans le volet Types de données sur la droite, cherchez le champ getPost sur le type Requête, puis choisissez Joindre.
- Dans le menu Action, choisissez Update runtime, puis Unit Resolver (VTL uniquement).
- Dans Nom de la source de données, choisissez PostDynamoDBTable.
- Collez ce qui suit dans la section Configurer le modèle de mappage de demande :

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

- Collez ce qui suit dans la section Configurer le modèle de mappage de réponse :

```
$utils.toJson($context.result)
```

- Choisissez Save (Enregistrer).

Appel de l'API pour obtenir une publication

Maintenant que le résolveur est configuré, AWS AppSync il sait comment traduire une `getPost` requête entrante en une opération DynamoDB`GetItem`. Vous pouvez désormais exécuter une requête pour récupérer la publication que vous avez créée précédemment.

- Choisissez l'onglet Requêtes.
- Collez ce qui suit dans le volet Requêtes :

```
query getPost {
  getPost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- La publication extraite de DynamoDB doit apparaître dans le volet des résultats à droite du volet des requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "getPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
```

```
      "downs": 0,
      "version": 1
    }
  }
}
```

Voici ce qui s'est produit :

- AWS AppSync a reçu une demande de `getPost` requête.
- AWS AppSync a pris la demande et le modèle de mappage des demandes, et a généré un document de mappage des demandes. Celui-ci ressemblait à :

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "123" }
    }
}
```

- AWS AppSync a utilisé le document de mappage des demandes pour générer et exécuter une demande GetItem DynamoDB.
- AWS AppSync a pris les résultats de la `GetItem` requête et les a reconvertis en types GraphQL.

```
{
    "id" : "123",
    "author": "AUTHORNAME",
    "title": "Our first post!",
    "content": "This is our first post.",
    "url": "https://aws.amazon.com/appsync/",
    "ups" : 1,
    "downs" : 0,
    "version" : 1
}
```

- Ils ont été transmis via le document de mappage de réponse, qui les a transmis sans les changer.
- L'objet récupéré a été renvoyé dans la réponse.

Vous pouvez également prendre l'exemple suivant :

```
query getPost {
  getPost(id:123) {
    id
    author
    title
  }
}
```

Si votre `getPost` requête n'a besoin que `duid`, et `authortitle`, vous pouvez modifier votre modèle de mappage de demandes pour utiliser des expressions de projection afin de spécifier uniquement les attributs que vous souhaitez voir apparaître dans votre table DynamoDB afin d'éviter tout transfert de données inutile de DynamoDB vers. AWS AppSync Par exemple, le modèle de mappage des demandes peut ressembler à l'extrait ci-dessous :

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "projection" : {
     "expression" : "#author, id, title",
     "expressionNames" : { "#author" : "author"}
    }
}
```

Création d'une mutation UpdatePost (DynamoDB) UpdateItem

Jusqu'à présent, vous pouvez créer et récupérer `Post` des objets dans DynamoDB. Ensuite, vous allez configurer une nouvelle mutation pour pouvoir mettre à jour l'objet. Pour ce faire, utilisez l'opération UpdateItem DynamoDB.

- Choisissez l'onglet Schéma.
- Dans le volet Schéma, modifiez le type `Mutation` pour ajouter une nouvelle mutation `updatePost` :

```
type Mutation {
    updatePost(
        id: ID!,
        author: String!,
```

```
        title: String!,
        content: String!,
        url: String!
    ): Post
    addPost(
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}
```

- Choisissez Save (Enregistrer).
- Dans le volet Types de données sur la droite, se trouve le champ updatePost qui vient d'être créé, sur le type Mutation, choisissez ensuite Joindre.
- Dans le menu Action, choisissez Update runtime, puis Unit Resolver (VTL uniquement).
- Dans Nom de la source de données, choisissez PostDynamoDBTable.
- Collez ce qui suit dans la section Configurer le modèle de mappage de demande :

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "SET author = :author, title = :title, content = :content,
 #url = :url ADD version :one",
        "expressionNames": {
            "#url" : "url"
        },
        "expressionValues": {
            ":author" : $util.dynamodb.toDynamoDBJson($context.arguments.author),
            ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title),
            ":content" : $util.dynamodb.toDynamoDBJson($context.arguments.content),
            ":url" : $util.dynamodb.toDynamoDBJson($context.arguments.url),
            ":one" : { "N": 1 }
        }
    }
}
```

Remarque : Ce résolveur utilise le UpdateItem DynamoDB, qui est très différent de l'opération. PutItem Au lieu d'écrire l'élément dans son intégralité, vous demandez simplement à DynamoDB de mettre à jour certains attributs. Cela se fait à l'aide des expressions de mise à jour DynamoDB. L'expression elle-même est spécifiée dans le champ `expression` de la section `update`. Elle indique de définir les attributs `author`, `title`, `content` et URL, puis d'incrémenter le champ `version`. Les valeurs à utiliser n'apparaissent pas dans l'expression elle-même ; l'expression a des espaces réservés qui ont des noms qui commencent par deux points et qui sont ensuite définis dans le champ `expressionValues`. Enfin, DynamoDB a réservé des mots qui ne peuvent pas apparaître dans le. `expression` Par exemple, `url` est un mot réservé, si bien que pour mettre à jour le champ `url`, vous pouvez utiliser des espaces réservés de nom et les définir dans le champ `expressionNames`.

Pour plus d'informations sur le mappage des `UpdateItem` demandes, consultez la documentation de UpdateItemréférence. Pour plus d'informations sur la façon d'écrire des expressions de mise à jour, consultez la documentation UpdateExpressions DynamoDB.

- Collez ce qui suit dans la section Configurer le modèle de mappage de réponse :

```
$utils.toJson($context.result)
```

Appel de l'API pour mettre à jour une publication

Maintenant que le résolveur est configuré, AWS AppSync il sait comment traduire une `update` mutation entrante en une opération DynamoDB`Update`. Vous pouvez désormais exécuter une mutation pour mettre à jour l'élément que vous avez écrit précédemment.

- Choisissez l'onglet Requêtes.
- Collez la mutation suivante dans le volet Requêtes. Vous devrez également mettre à jour l'argument `id` pour qu'il ait la valeur que vous avez notée précédemment.

```
mutation updatePost {
  updatePost(
    id:"123"
    author: "A new author"
    title: "An updated author!"
    content: "Now with updated content!"
    url: "https://aws.amazon.com/appsync/"
  ) {
```

```
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- La publication mise à jour dans DynamoDB doit apparaître dans le volet des résultats à droite du volet des requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "updatePost": {
      "id": "123",
      "author": "A new author",
      "title": "An updated author!",
      "content": "Now with updated content!",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 2
    }
  }
}
```

Dans cet exemple, les `downs` champs `ups` et n'ont pas été modifiés car le modèle de mappage de demandes n'a pas demandé à AWS AppSync DynamoDB de faire quoi que ce soit avec ces champs. De plus, le `version` champ a été incrémenté de 1 parce que vous avez demandé AWS AppSync à DynamoDB d'ajouter 1 au champ. `version`

Modification du résolveur UpdatePost (DynamoDB) UpdateItem

Il s'agit d'un bon début pour la mutation `updatePost`, mais elle comporte deux problèmes principaux :

- Si vous souhaitez mettre à jour un seul champ, vous devez mettre à jour tous les champs.

- Si deux personnes modifient l'objet, des informations risquent d'être perdues.

Pour résoudre ces problèmes, vous allez modifier la mutation `updatePost` pour modifier uniquement les arguments qui ont été spécifiés dans la demande, puis ajouter une condition à l'opération `UpdateItem`.

1. Choisissez l'onglet Schéma.
2. Dans le volet Schéma, modifiez le champ `updatePost` dans le type `Mutation` pour supprimer les points d'exclamation des arguments `author`, `title`, `content`, et `url`, en veillant à laisser le champ `id` en l'état. Cela rend ces arguments facultatifs. En outre, ajoutez un nouvel argument `expectedVersion` requis.

```
type Mutation {
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}
```

3. Choisissez Save (Enregistrer).
4. Dans le volet Types de données de droite, cherchez le champ updatePost sur le type Mutation.
5. Choisissez PostDynamoDBTable pour ouvrir le résolveur existant.
6. Modifiez le modèle de mappage de demande dans la section Configurer le modèle de mappage de demande :

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
```

```
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },

    ## Set up some space to keep track of things you're updating **
    #set( $expNames  = {} )
    #set( $expValues = {} )
    #set( $expSet = {} )
    #set( $expAdd = {} )
    #set( $expRemove = [] )

    ## Increment "version" by 1 **
    $!{expAdd.put("version", ":one")}
    $!{expValues.put(":one", { "N" : 1 })}

    ## Iterate through each argument, skipping "id" and "expectedVersion" **
    #foreach( $entry in $context.arguments.entrySet() )
        #if( $entry.key != "id" && $entry.key != "expectedVersion" )
            #if( (!$entry.value) && ("$!{entry.value}" == "") )
                ## If the argument is set to "null", then remove that attribute from
the item in DynamoDB **

                #set( $discard = ${expRemove.add("#${entry.key}")} )
                $!{expNames.put("#${entry.key}", "$entry.key")}
            #else
                ## Otherwise set (or update) the attribute on the item in DynamoDB **

                $!{expSet.put("#${entry.key}", ":${entry.key}")}
                $!{expNames.put("#${entry.key}", "$entry.key")}
                $!{expValues.put(":${entry.key}", { "S" : "${entry.value}" })}
            #end
        #end
    #end

    ## Start building the update expression, starting with attributes you're going to
SET **
    #set( $expression = "" )
    #if( !${expSet.isEmpty()} )
        #set( $expression = "SET" )
        #foreach( $entry in $expSet.entrySet() )
            #set( $expression = "${expression} ${entry.key} = ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
```

```
    #end

    ## Continue building the update expression, adding attributes you're going to ADD
**
    #if( !${expAdd.isEmpty()} )
        #set( $expression = "${expression} ADD" )
        #foreach( $entry in $expAdd.entrySet() )
            #set( $expression = "${expression} ${entry.key} ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Continue building the update expression, adding attributes you're going to
REMOVE **
    #if( !${expRemove.isEmpty()} )
        #set( $expression = "${expression} REMOVE" )

        #foreach( $entry in $expRemove )
            #set( $expression = "${expression} ${entry}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Finally, write the update expression into the document, along with any
expressionNames and expressionValues **
    "update" : {
        "expression" : "${expression}"
        #if( !${expNames.isEmpty()} )
            ,"expressionNames" : $utils.toJson($expNames)
        #end
        #if( !${expValues.isEmpty()} )
            ,"expressionValues" : $utils.toJson($expValues)
        #end
    },

    "condition" : {
        "expression"       : "version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" :
$util.dynamodb.toDynamoDBJson($context.arguments.expectedVersion)
```

```
        }
    }
}
```

7. Choisissez Save (Enregistrer).

Ce modèle est l'un des exemples les plus complexes. Il illustre la puissance et la flexibilité des modèles de mappage. Il passe en revue tous les arguments, en faisant l'impasse sur `id` et `expectedVersion`. Si l'argument est défini sur quelque chose, il demande AWS AppSync à DynamoDB de mettre à jour cet attribut sur l'objet dans DynamoDB. Si l'attribut est défini sur null, il demande AWS AppSync à DynamoDB de le supprimer de l'objet de publication. Si un argument n'a pas été spécifié, il laisse l'attribut. Il incrémente également le champ `version`.

Il existe également une nouvelle section `condition`. Une expression de condition vous permet de dire AWS AppSync à DynamoDB si la demande doit aboutir ou non en fonction de l'état de l'objet déjà présent dans DynamoDB avant l'exécution de l'opération. Dans ce cas, vous souhaitez que la `UpdateItem` demande aboutisse uniquement si le `version` champ de l'élément actuellement dans DynamoDB correspond exactement à l'argument. `expectedVersion`

Pour plus d'informations sur les expressions de condition, consultez la documentation de référence Expressions de condition.

Appel de l'API pour mettre à jour une publication

Essayons de mettre à jour l'objet `Post` avec le nouveau résolveur :

- Choisissez l'onglet Requêtes.
- Collez la mutation suivante dans le volet Requêtes. Vous devrez également mettre à jour l'argument `id` pour qu'il ait la valeur que vous avez notée précédemment.

```
mutation updatePost {
  updatePost(
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 2
  ) {
    id
    author
    title
    content
```

```
    url
    ups
    downs
    version
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- La publication mise à jour dans DynamoDB doit apparaître dans le volet des résultats à droite du volet des requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "updatePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 3
    }
  }
}
```

Dans cette demande, vous avez demandé à DynamoDB de mettre à jour `title` le `content` champ AWS AppSync et uniquement. Cela a laissé seuls tous les autres champs (autres que l'incrémentation du champ `version`). Vous avez défini l'attribut `title` sur une nouvelle valeur et avons supprimé l'attribut `content` de la publication. Les champs `author`, `url`, `ups` et `downs` sont restés inchangés.

Essayez d'exécuter une nouvelle fois la demande de mutation en laissant la demande exactement telle quelle. La réponse devrait être similaire à ce qui suit :

```
{
  "data": {
    "updatePost": null
  },
  "errors": [
    {
```

```
      "path": [
        "updatePost"
      ],
      "data": {
        "id": "123",
        "author": "A new author",
        "title": "An empty story",
        "content": null,
        "url": "https://aws.amazon.com/appsync/",
        "ups": 1,
        "downs": 0,
        "version": 3
      },
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
    }
  ]
}
```

La demande échoue, car l'expression de condition prend la valeur false :

- La première fois que vous avez exécuté la demande, la valeur du `version` champ de la publication dans DynamoDB 2 était, qui correspondait à l'argument. `expectedVersion` La demande a abouti, ce qui signifie que le `version` champ a été incrémenté dans DynamoDB à. `3`
- La deuxième fois que vous avez exécuté la demande, la valeur du `version` champ de la publication dans DynamoDB `3` était, ce qui ne correspondait pas à l'argument. `expectedVersion`

Ce modèle est généralement appelé Verrouillage optimiste.

L'une des fonctionnalités d'un résolveur AWS AppSync DynamoDB est qu'il renvoie la valeur actuelle de l'objet de publication dans DynamoDB. Vous pouvez trouver cette valeur dans le champ `data` de la section `errors` de la réponse GraphQL. Votre application peut utiliser ces informations pour déterminer la façon dont elle doit continuer. Dans ce cas, vous pouvez voir que le `version`

champ de l'objet dans DynamoDB est défini sur. Vous pouvez donc simplement mettre `3` à jour `expectedVersion` l'argument `3` pour que la demande aboutisse à nouveau.

Consultez la documentation de référence sur le modèle de mappage Expressions de condition pour obtenir plus d'informations sur la gestion des échecs de vérification de condition.

Création de mutations UpVotePost et DownVotePost (DynamoDB) UpdateItem

Le type `Post` a des champs `ups` et `downs` pour permettre d'enregistrer les votes pour et contre, mais jusqu'à présent l'API ne nous laisse rien faire avec eux. Permet d'ajouter certaines mutations pour nous laisser voter pour ou contre les publications.

- Choisissez l'onglet Schéma.
- Dans le volet Schéma, modifiez le type `Mutation` pour ajouter de nouvelles mutations `upvotePost` et `downvotePost` :

```
type Mutation {
    upvotePost(id: ID!): Post
    downvotePost(id: ID!): Post
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post!
}
```

- Choisissez Save (Enregistrer).
- Dans le volet Types de données sur la droite, se trouve le champ upvotePost qui vient d'être créé, sur le type Mutation, choisissez ensuite Joindre.
- Dans le menu Action, choisissez Update runtime, puis Unit Resolver (VTL uniquement).

- Dans Nom de la source de données, choisissez PostDynamoDBTable.
- Collez ce qui suit dans la section Configurer le modèle de mappage de demande :

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "ADD ups :plusOne, version :plusOne",
        "expressionValues" : {
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- Collez ce qui suit dans la section Configurer le modèle de mappage de réponse :

```
$utils.toJson($context.result)
```

- Choisissez Save (Enregistrer).
- Dans le volet Types de données sur la droite, se trouve le champ `downvotePost` qui vient d'être créé, sur le type Mutation, choisissez ensuite Joindre.
- Dans Nom de la source de données, choisissez PostDynamoDBTable.
- Collez ce qui suit dans la section Configurer le modèle de mappage de demande :

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "ADD downs :plusOne, version :plusOne",
        "expressionValues" : {
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- Collez ce qui suit dans la section Configurer le modèle de mappage de réponse :

```
$utils.toJson($context.result)
```

- Choisissez Save (Enregistrer).

Appel de l'API pour voter pour ou contre une publication

Maintenant que les nouveaux résolveurs ont été configurés, ils AWS AppSync savent comment traduire une opération entrante `upvotePost` ou une `downvote` mutation en opération DynamoDB UpdateItem . Vous pouvez désormais exécuter des mutations pour voter pour ou contre la publication que vous avez créée précédemment.

- Choisissez l'onglet Requêtes.
- Collez la mutation suivante dans le volet Requêtes. Vous devrez également mettre à jour l'argument `id` pour qu'il ait la valeur que vous avez notée précédemment.

```
mutation votePost {
  upvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- La publication est mise à jour dans DynamoDB et doit apparaître dans le volet des résultats à droite du volet des requêtes. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "upvotePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
```

```
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 0,
      "version": 4
    }
  }
}
```

- Choisissez Exécuter une requête quelque fois de plus. Vous devez voir le champ `ups` et `version` être incrémenté de 1 chaque fois que vous exécutez la requête.
- Modifiez la requête pour appeler la mutation `downvotePost` :

```
mutation votePost {
  downvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange). Cette fois, vous devez voir le champ `downs` et `version` être incrémenté de 1 chaque fois que vous exécutez la requête.

```
{
  "data": {
    "downvotePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 4,
      "version": 12
    }
  }
```

```
}
```

Configuration du résolveur DeletePost (DynamoDB) DeleteItem

La prochaine mutation que vous allez configurer permettra de supprimer une publication. Pour ce faire, utilisez l'opération `DeleteItem` DynamoDB.

- Choisissez l'onglet Schéma.
- Dans le volet Schéma, modifiez le type `Mutation` pour ajouter une nouvelle mutation `deletePost` :

```
type Mutation {
    deletePost(id: ID!, expectedVersion: Int): Post
    upvotePost(id: ID!): Post
    downvotePost(id: ID!): Post
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post!
}
```

  Cette fois vous avez rendu le champ `expectedVersion` facultatif, ce qui est expliqué plus loin lorsque vous ajoutez le modèle de mappage de demande.
- Choisissez Save (Enregistrer).
- Dans le volet Types de données sur la droite, se trouve le champ delete qui vient d'être créé, sur le type Mutation, choisissez ensuite Joindre.
- Dans le menu Action, choisissez Update runtime, puis Unit Resolver (VTL uniquement).
- Dans Nom de la source de données, choisissez PostDynamoDBTable.

- Collez ce qui suit dans la section Configurer le modèle de mappage de demande :

```
{
    "version" : "2017-02-28",
    "operation" : "DeleteItem",
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($context.arguments.id)
    }
    #if( $context.arguments.containsKey("expectedVersion") )
        ,"condition" : {
            "expression"       : "attribute_not_exists(id) OR version
 = :expectedVersion",
            "expressionValues" : {
                ":expectedVersion" :
 $util.dynamodb.toDynamoDBJson($context.arguments.expectedVersion)
            }
        }
    #end
}
```

  Remarque : l'argument `expectedVersion` est un argument facultatif. Si l'appelant définit un `expectedVersion` argument dans la demande, le modèle ajoute une condition qui permet à la `DeleteItem` demande de réussir uniquement si l'élément est déjà supprimé ou si l'`version`attribut de la publication dans DynamoDB correspond exactement au. `expectedVersion` En cas d'omission, aucune expression de condition n'est spécifiée dans la demande `DeleteItem`. Il réussit quelle que soit la valeur de l'`version`élément ou qu'il existe ou non dans DynamoDB.

- Collez ce qui suit dans la section Configurer le modèle de mappage de réponse :

```
$utils.toJson($context.result)
```

  Remarque : même si vous supprimez un élément, vous pouvez renvoyer l'élément qui a été supprimé, s'il n'était pas déjà supprimé.

- Choisissez Save (Enregistrer).

Pour plus d'informations sur le mappage des `DeleteItem` demandes, consultez la documentation de DeleteItemréférence.

Appel de l'API pour supprimer une publication

Maintenant que le résolveur est configuré, AWS AppSync il sait comment traduire une `delete` mutation entrante en une opération DynamoDB`DeleteItem`. Vous pouvez désormais exécuter une mutation pour supprimer quelque chose dans la table.

- Choisissez l'onglet Requêtes.
- Collez la mutation suivante dans le volet Requêtes. Vous devrez également mettre à jour l'argument `id` pour qu'il ait la valeur que vous avez notée précédemment.

```
mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- La publication est supprimée de DynamoDB. Notez qu'AWS AppSync renvoie la valeur de l'élément qui a été supprimé de DynamoDB, laquelle doit apparaître dans le volet des résultats à droite du volet de requête. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 4,
      "version": 12
    }
  }
```

```
}
```

La valeur est renvoyée seulement si cet appel à `deletePost` est celui qui l'a réellement supprimée de DynamoDB.

- Choisissez Exécuter une requête à nouveau.
- L'appel réussit quand même, mais aucune valeur n'est renvoyée.

```
{
  "data": {
    "deletePost": null
  }
}
```

Maintenant, essayons de supprimer une publication, mais cette fois en spécifiant `expectedValue`. Tout d'abord, vous devez créer une nouvelle publication, car vous venez de supprimer celle que vous utilisiez jusqu'à présent.

- Collez la mutation suivante dans le volet Requêtes.

```
mutation addPost {
  addPost(
    id:123
    author: "AUTHORNAME"
    title: "Our second post!"
    content: "A new post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).

- Les résultats de la publication nouvellement créée doivent apparaître dans le volet des résultats à droite du volet de requête. Notez l'`id` de l'objet nouvellement créé, car vous en aurez besoin sous peu. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

Maintenant, essayons de supprimer cette publication, mais insérons une valeur incorrecte pour `expectedVersion` :

- Collez la mutation suivante dans le volet Requêtes. Vous devrez également mettre à jour l'argument `id` pour qu'il ait la valeur que vous avez notée précédemment.

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).


```
{
  "data": {
    "deletePost": null
  },
  "errors": [
    {
      "path": [
        "deletePost"
      ],
      "data": {
        "id": "123",
        "author": "AUTHORNAME",
        "title": "Our second post!",
        "content": "A new post.",
        "url": "https://aws.amazon.com/appsync/",
        "ups": 1,
        "downs": 0,
        "version": 1
      },
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
    }
  ]
}
```


La demande a échoué car l'expression de la condition est fausse : la valeur `version` de la publication dans DynamoDB ne correspond pas à celle spécifiée dans `expectedValue` les arguments. La valeur actuelle de l'objet est renvoyée dans le champ `data` de la section `errors` de la réponse GraphQL.

- Réessayez la demande, mais corrigez `expectedVersion` :

```
mutation deletePost {
```

```
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- Cette fois, la demande aboutit et la valeur supprimée de DynamoDB est renvoyée :

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

- Choisissez Exécuter une requête à nouveau.
- L'appel réussit toujours, mais cette fois aucune valeur n'est renvoyée car la publication a déjà été supprimée dans DynamoDB.

```
{
  "data": {
    "deletePost": null
  }
```

```
}
```

Configuration du résolveur allPost (DynamoDB Scan)

Jusqu'à présent, l'API est utile uniquement si vous connaissez l'`id` de chaque publication que vous voulez regarder. Ajoutons un nouveau résolveur qui renvoie toutes les publications de la table.

- Choisissez l'onglet Schéma.
- Dans le volet Schéma, modifiez le type `Query` pour ajouter une nouvelle requête `allPost` :

```
type Query {
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

- Ajouter un nouveau type `PaginationPosts` :

```
type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}
```

- Choisissez Save (Enregistrer).
- Dans le volet Types de données sur la droite, se trouve le champ allPost qui vient d'être créé, sur le type Requête, choisissez ensuite Joindre.
- Dans le menu Action, choisissez Update runtime, puis Unit Resolver (VTL uniquement).
- Dans Nom de la source de données, choisissez PostDynamoDBTable.
- Collez ce qui suit dans la section Configurer le modèle de mappage de demande :

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
    #if( ${context.arguments.count} )
        ,"limit": $util.toJson($context.arguments.count)
    #end
    #if( ${context.arguments.nextToken} )
        ,"nextToken": $util.toJson($context.arguments.nextToken)
    #end
}
```

Ce résolveur possède deux arguments facultatifs : `count`, qui spécifie le nombre maximal d'éléments à renvoyer dans un seul appel, et `nextToken`, qui peut être utilisé pour récupérer l'ensemble de résultats suivant (vous indiquerez ultérieurement d'où provient la valeur de `nextToken`).

- Collez ce qui suit dans la section Configurer le modèle de mappage de réponse :

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

Remarque : ce modèle de mappage de réponse est différent de tous les autres jusqu'à présent. Le résultat de la requête `allPost` est un `PaginatedPosts`, qui contient une liste de publications et un jeton de pagination. La forme de cet objet est différente de ce qui est renvoyé du résolveur DynamoDB d'AWS AppSync : la liste des publications est appelée `items` dans les résultats du résolveur DynamoDB d'AWS AppSync , mais elle est appelée `posts` dans `PaginatedPosts`.

- Choisissez Save (Enregistrer).

Pour en savoir plus sur le mappage des demandes `Scan`, consultez la documentation de référence Scan.

Appel de l'API pour analyser toutes les publications

Maintenant que le résolveur est configuré, AWS AppSync il sait comment traduire une `allPost` requête entrante en une opération DynamoDB`Scan`. Vous pouvez désormais analyser la table pour récupérer toutes les publications.

Avant de pouvoir essayer, vous avez besoin de remplir la table avec certaines données, car vous avez supprimé tout ce que vous aviez utilisé jusque là.

- Choisissez l'onglet Requêtes.
- Collez la mutation suivante dans le volet Requêtes.

```
mutation addPost {
  post1: addPost(id:1 author: "AUTHORNAME" title: "A series of posts, Volume 1"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
```

```
  post2: addPost(id:2 author: "AUTHORNAME" title: "A series of posts, Volume 2"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post3: addPost(id:3 author: "AUTHORNAME" title: "A series of posts, Volume 3"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post4: addPost(id:4 author: "AUTHORNAME" title: "A series of posts, Volume 4"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post5: addPost(id:5 author: "AUTHORNAME" title: "A series of posts, Volume 5"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post6: addPost(id:6 author: "AUTHORNAME" title: "A series of posts, Volume 6"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post7: addPost(id:7 author: "AUTHORNAME" title: "A series of posts, Volume 7"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post8: addPost(id:8 author: "AUTHORNAME" title: "A series of posts, Volume 8"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post9: addPost(id:9 author: "AUTHORNAME" title: "A series of posts, Volume 9"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).

Maintenant, analysons la table, en renvoyant cinq résultats à la fois.

- Collez la requête suivante dans le volet Requêtes :

```
query allPost {
  allPost(count: 5) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- Les cinq premières publications doivent apparaître dans le volet des résultats à droite du volet de requête. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "allPost": {
      "posts": [
```

```
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        },
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        }
      ],
      "nextToken":
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
    }
  }
}
```

Vous avez cinq résultats et un `nextToken` que vous pouvez utiliser pour obtenir l'ensemble de résultats suivant.

- Mettez à jour la requête `allPost` pour qu'elle inclue l'élément `nextToken` issu de l'ensemble de résultats précédent :

```
query allPost {
  allPost(
    count: 5
    nextToken:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
  ) {
    posts {
      id
      author
```

```
    }
    nextToken
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- Les quatre publications restantes doivent apparaître dans le volet des résultats à droite du volet de requête. Il n'y a pas d'élément `nextToken` dans cet ensemble de résultats, car vous avez parcouru les neuf publications, et il n'en reste aucune. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        }
      ],
      "nextToken": null
    }
  }
}
```

Configuration du résolveur d' allPostsByauteur (requête DynamoDB)

En plus de rechercher dans DynamoDB toutes les publications, vous pouvez également interroger DynamoDB pour récupérer les publications créées par un auteur spécifique. La table DynamoDB que vous avez créée précédemment possède déjà `GlobalSecondaryIndex` un `author-index`

appel que vous pouvez utiliser avec une opération DynamoDB pour récupérer toutes les `Query` publications créées par un auteur spécifique.

- Choisissez l'onglet Schéma.
- Dans le volet Schéma, modifiez le type `Query` pour ajouter une nouvelle requête `allPostsByAuthor` :

```
type Query {
    allPostsByAuthor(author: String!, count: Int, nextToken: String): PaginatedPosts!
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

  Remarque : ceci utilise le même type `PaginatedPosts` que nous avons utilisé avec la requête `allPost`.
- Choisissez Save (Enregistrer).
- Dans le volet Types de données sur la droite, recherchez le champ allPostsByAuteur nouvellement créé dans le type de requête, puis choisissez Joindre.
- Dans le menu Action, choisissez Update runtime, puis Unit Resolver (VTL uniquement).
- Dans Nom de la source de données, choisissez PostDynamoDBTable.
- Collez ce qui suit dans la section Configurer le modèle de mappage de demande :

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "index" : "author-index",
    "query" : {
      "expression": "author = :author",
        "expressionValues" : {
          ":author" : $util.dynamodb.toDynamoDBJson($context.arguments.author)
        }
    }
    #if( ${context.arguments.count} )
        ,"limit": $util.toJson($context.arguments.count)
    #end
    #if( ${context.arguments.nextToken} )
        ,"nextToken": "${context.arguments.nextToken}"
    #end
}
```

Comme le résolveur `allPost`, ce résolveur possède deux arguments facultatifs : `count`, qui spécifie le nombre maximal d'éléments à renvoyer dans un seul appel et `nextToken`, qui peut être utilisé pour récupérer l'ensemble de résultats suivant (la valeur de `nextToken` peut être obtenue à partir d'un appel précédent).

- Collez ce qui suit dans la section Configurer le modèle de mappage de réponse :

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

Remarque : il s'agit du même modèle de mappage de réponse que vous avez utilisé dans le résolveur `allPost`.

- Choisissez Save (Enregistrer).

Pour en savoir plus sur le mappage des demandes `Query`, consultez la documentation de référence Requête.

Appel de l'API pour interroger toutes les publications d'un auteur

Maintenant que le résolveur est configuré, AWS AppSync il sait comment traduire une `allPostsByAuthor` mutation entrante en une opération `Query` DynamoDB par rapport à l'index. `author-index` Vous pouvez désormais interroger la table pour récupérer toutes les publications d'un auteur spécifique.

Avant cela, toutefois, remplissez la table avec d'autres publications car, à ce stade, chaque publication est du même auteur.

- Choisissez l'onglet Requêtes.
- Collez la mutation suivante dans le volet Requêtes.

```
mutation addPost {
  post1: addPost(id:10 author: "Nadia" title: "The cutest dog in the world" content:
 "So cute. So very, very cute." url: "https://aws.amazon.com/appsync/" ) { author,
 title }
```

```
  post2: addPost(id:11 author: "Nadia" title: "Did you know...?" content: "AppSync
 works offline?" url: "https://aws.amazon.com/appsync/" ) { author, title }
  post3: addPost(id:12 author: "Steve" title: "I like GraphQL" content: "It's great"
 url: "https://aws.amazon.com/appsync/" ) { author, title }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).

Maintenant, interrogez la table et renvoyez toutes les publications créées par `Nadia`.

- Collez la requête suivante dans le volet Requêtes :

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- Toutes les publications créées par `Nadia` doivent apparaître dans le volet des résultats à droite du volet de requête. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
```

```
}
```

La pagination fonctionne pour `Query` tout comme elle le fait pour `Scan`. Par exemple, examinons toutes les publications créées par `AUTHORNAME`, et prenons-en cinq à la fois.

- Collez la requête suivante dans le volet Requêtes :

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- Toutes les publications créées par `AUTHORNAME` doivent apparaître dans le volet des résultats à droite du volet de requête. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
```

```
          "id": "7",
          "title": "A series of posts, Volume 7"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        }
      ],
      "nextToken":
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
    }
  }
}
```

- Mettez à jour l'argument `nextToken` avec la valeur renvoyée par la requête précédente :

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
    nextToken:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- Les publications créées par AUTHORNAME doivent apparaître dans le volet des résultats à droite du volet de requête. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        },
```

```
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        }
      ],
      "nextToken": null
    }
  }
}
```

Utilisation des ensembles

Jusqu'ici, le type `Post` a été un objet clé/valeur plat. Vous pouvez également modéliser des objets complexes avec le résolveur AWS AppSyncDynamo de base de données, tels que des ensembles, des listes et des cartes.

Mettons à jour notre type `Post` pour inclure des balises. Une publication peut comporter 0 balises ou plus, qui sont stockées dans DynamoDB sous forme de jeu de chaînes. Vous allez également configurer certaines mutations pour ajouter et supprimer des balises, et une nouvelle requête pour rechercher des publications avec une balise spécifique.

- Choisissez l'onglet Schéma.
- Dans le volet Schéma, modifiez le type `Post` pour ajouter un nouveau champ `tags` :

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
```

```
  tags: [String!]
}
```

- Dans le volet Schéma, modifiez le type `Query` pour ajouter une nouvelle requête `allPostsByTag` :

```
type Query {
  allPostsByTag(tag: String!, count: Int, nextToken: String): PaginatedPosts!
  allPostsByAuthor(author: String!, count: Int, nextToken: String): PaginatedPosts!
  allPost(count: Int, nextToken: String): PaginatedPosts!
  getPost(id: ID): Post
}
```

- Dans le volet Schéma, modifiez le type `Mutation` pour ajouter de nouvelles mutations `addTag` et `removeTag` :

```
type Mutation {
  addTag(id: ID!, tag: String!): Post
  removeTag(id: ID!, tag: String!): Post
  deletePost(id: ID!, expectedVersion: Int): Post
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}
```

- Choisissez Save (Enregistrer).
- Dans le volet Types de données sur la droite, recherchez le champ allPostsByTag nouvellement créé dans le type de requête, puis choisissez Joindre.
- Dans Nom de la source de données, choisissez PostDynamoDBTable.

- Collez ce qui suit dans la section Configurer le modèle de mappage de demande :

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "filter": {
      "expression": "contains (tags, :tag)",
        "expressionValues": {
          ":tag": $util.dynamodb.toDynamoDBJson($context.arguments.tag)
        }
    }
    #if( ${context.arguments.count} )
        ,"limit": $util.toJson($context.arguments.count)
    #end
    #if( ${context.arguments.nextToken} )
        ,"nextToken": $util.toJson($context.arguments.nextToken)
    #end
}
```

- Collez ce qui suit dans la section Configurer le modèle de mappage de réponse :

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

- Choisissez Save (Enregistrer).
- Dans le volet Types de données sur la droite, se trouve le champ addTag qui vient d'être créé, sur le type Mutation, choisissez ensuite Joindre.
- Dans Nom de la source de données, choisissez PostDynamoDBTable.
- Collez ce qui suit dans la section Configurer le modèle de mappage de demande :

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "ADD tags :tags, version :plusOne",
```

```
        "expressionValues" : {
            ":tags" : { "SS": [ $util.toJson($context.arguments.tag) ] },
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- Collez ce qui suit dans la section Configurer le modèle de mappage de réponse :

```
$utils.toJson($context.result)
```

- Choisissez Save (Enregistrer).
- Dans le volet Types de données sur la droite, se trouve le champ removeTag qui vient d'être créé, sur le type Mutation, choisissez ensuite Joindre.
- Dans Nom de la source de données, choisissez PostDynamoDBTable.
- Collez ce qui suit dans la section Configurer le modèle de mappage de demande :

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "DELETE tags :tags ADD version :plusOne",
        "expressionValues" : {
            ":tags" : { "SS": [ $util.toJson($context.arguments.tag) ] },
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- Collez ce qui suit dans la section Configurer le modèle de mappage de réponse :

```
$utils.toJson($context.result)
```

- Choisissez Save (Enregistrer).

Appel de l'API pour utiliser des balises

Maintenant que vous avez configuré les résolveurs, vous savez comment AWS AppSync traduire les requêtes entrantes et `addTag` les `removeTag` `allPostsByTag` requêtes en `UpdateItem` DynamoDB `Scan` et en opérations.

Pour tester cela, sélectionnons l'une des publications que nous avons créées précédemment. Par exemple, utilisons un article créé par `Nadia`.

- Choisissez l'onglet Requêtes.
- Collez la requête suivante dans le volet Requêtes :

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- Toutes les publications de Nadia doivent apparaître dans le volet des résultats à droite du volet de requête. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
```

```
    }
  }
}
```

- Utilisons celui avec le titre `"The cutest dog in the world"`. Notez son `id` car vous l'utiliserez ultérieurement.

Essayons maintenant d'ajouter une balise `dog`.

- Collez la mutation suivante dans le volet Requêtes. Vous devrez également mettre à jour l'argument `id` pour qu'il ait la valeur que vous avez notée précédemment.

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- La publication est mise à jour avec la nouvelle balise.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

Vous pouvez ajouter encore d'autres balises :

- Mettez à jour la mutation pour modifier l'argument `tag` en spécifiant `puppy`.

```
mutation addTag {
```

```
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- La publication est mise à jour avec la nouvelle balise.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

Vous pouvez également supprimer des balises :

- Collez la mutation suivante dans le volet Requêtes. Vous devrez également mettre à jour l'argument `id` pour qu'il ait la valeur que vous avez notée précédemment.

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- La publication est mise à jour et la balise `puppy` est supprimée.

```
{
  "data": {
```

```
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

Vous pouvez également rechercher toutes les publications qui ont une balise :

- Collez la requête suivante dans le volet Requêtes :

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- Toutes les publications qui ont la balise dog sont renvoyées :

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
            "dog",
            "puppy"
          ]
        }
      ],
      "nextToken": null
```

```
    }
  }
}
```

Utilisation de listes et de cartes

Outre les ensembles DynamoDB, vous pouvez également utiliser des listes et des cartes DynamoDB pour modéliser des données complexes dans un seul objet.

Ajoutons la possibilité d'ajouter des commentaires aux publications. Cela sera modélisé sous la forme d'une liste d'objets cartographiques sur l'`Post`objet dans DynamoDB.

Remarque : dans une application réelle, vous devez modéliser les commentaires dans leur propre table. Dans le cadre de ce didacticiel, vous devez simplement les ajouter dans la table `Post`.

- Choisissez l'onglet Schéma.
- Dans le volet Schéma, ajoutez un nouveau type `Comment` :

```
type Comment {
    author: String!
    comment: String!
}
```

- Dans le volet Schéma, modifiez le type `Post` pour ajouter un nouveau champ `comments` :

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
  comments: [Comment!]
}
```

- Dans le volet Schéma, modifiez le type `Mutation` pour ajouter une nouvelle mutation `addComment` :

```
type Mutation {
  addComment(id: ID!, author: String!, comment: String!): Post
  addTag(id: ID!, tag: String!): Post
  removeTag(id: ID!, tag: String!): Post
  deletePost(id: ID!, expectedVersion: Int): Post
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}
```

- Choisissez Save (Enregistrer).
- Dans le volet Types de données sur la droite, se trouve le champ addComment qui vient d'être créé, sur le type Mutation, choisissez ensuite Joindre.
- Dans Nom de la source de données, choisissez PostDynamoDBTable.
- Collez ce qui suit dans la section Configurer le modèle de mappage de demande :

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "update" : {
    "expression" : "SET comments =
 list_append(if_not_exists(comments, :emptyList), :newComment) ADD version :plusOne",
    "expressionValues" : {
      ":emptyList": { "L" : [] },
      ":newComment" : { "L" : [
```

```
        { "M": {
          "author": $util.dynamodb.toDynamoDBJson($context.arguments.author),
          "comment": $util.dynamodb.toDynamoDBJson($context.arguments.comment)
          }
        }
      ] },
      ":plusOne" : $util.dynamodb.toDynamoDBJson(1)
    }
  }
}
```

Cette expression de mise à jour ajoute une liste contenant notre nouveau commentaire à la liste `comments` existante. Si la liste n'existe pas encore, elle sera créée.

- Collez ce qui suit dans la section Configurer le modèle de mappage de réponse :

```
$utils.toJson($context.result)
```

- Choisissez Save (Enregistrer).

Appel de l'API pour ajouter un commentaire

Maintenant que vous avez configuré les résolveurs, vous savez comment AWS AppSync traduire les `addComment` demandes entrantes en opérations DynamoDB`UpdateItem`.

Essayons cela en ajoutant un commentaire à la même publication à laquelle vous avez ajouté les balises.

- Choisissez l'onglet Requêtes.
- Collez la requête suivante dans le volet Requêtes :

```
mutation addComment {
  addComment(
    id:10
    author: "Steve"
    comment: "Such a cute dog."
  ) {
    id
    comments {
      author
      comment
    }
```

```
  }
}
```

- Choisissez Exécuter une requête (le bouton de lecture orange).
- Toutes les publications de Nadia doivent apparaître dans le volet des résultats à droite du volet de requête. Il doit ressembler à l'exemple ci-dessous.

```
{
  "data": {
    "addComment": {
      "id": "10",
      "comments": [
        {
          "author": "Steve",
          "comment": "Such a cute dog."
        }
      ]
    }
  }
}
```

Si vous exécutez la demande plusieurs fois, plusieurs commentaires seront ajoutés à la liste.

Conclusion

Dans ce didacticiel, vous avez créé une API qui nous permet de manipuler des objets Post dans DynamoDB à l'aide de AWS AppSync GraphQL. Pour plus d'informations, consultez la Référence du modèle de mappage des résolveurs.

Pour nettoyer, vous pouvez supprimer l'API AppSync GraphQL de la console.

Pour supprimer la table DynamoDB et le rôle IAM que vous avez créés pour ce didacticiel, vous pouvez exécuter les opérations suivantes pour supprimer la pile, ou accéder à `AWSAppSyncTutorialForAmazonDynamoDB` la console et supprimer AWS CloudFormation la pile :

```
aws cloudformation delete-stack \
    --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

Tutoriel : résolveurs Lambda

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Vous pouvez utiliser AWS Lambda with AWS AppSync pour résoudre n'importe quel champ GraphQL. Par exemple, une requête GraphQL peut envoyer un appel à une instance Amazon Relational Database Service (Amazon RDS), et une mutation GraphQL peut écrire dans un flux Amazon Kinesis. Dans cette section, nous allons vous montrer comment écrire une fonction Lambda qui exécute une logique métier basée sur l'invocation d'une opération de terrain GraphQL.

Création d'une fonction Lambda

L'exemple suivant montre une fonction Lambda écrite `Node.js` qui effectue différentes opérations sur des articles de blog dans le cadre d'une application de publication de blog.

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    var posts = {
         "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://
amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR
 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100",
 "downs": "10"},
         "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://
amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups":
 "100", "downs": "10"},
         "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null,
 "content": null, "ups": null, "downs": null },
         "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
         "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} };

    var relatedPosts = {
        "1": [posts['4']],
```

```
        "2": [posts['3'], posts['5']],
        "3": [posts['2'], posts['1']],
        "4": [posts['2'], posts['1']],
        "5": []
    };

    console.log("Got an Invoke Request.");
    switch(event.field) {
        case "getPost":
            var id = event.arguments.id;
            callback(null, posts[id]);
            break;
        case "allPosts":
            var values = [];
            for(var d in posts){
                values.push(posts[d]);
            }
            callback(null, values);
            break;
        case "addPost":
            // return the arguments back
            callback(null, event.arguments);
            break;
        case "addPostErrorWithData":
            var id = event.arguments.id;
            var result = posts[id];
            // attached additional error information to the post
            result.errorMessage = 'Error with the mutation, data has changed';
            result.errorType = 'MUTATION_ERROR';
            callback(null, result);
            break;
        case "relatedPosts":
            var id = event.source.id;
            callback(null, relatedPosts[id]);
            break;
        default:
            callback("Unknown field, unable to resolve" + event.field, null);
            break;
    }
};
```

Cette fonction Lambda récupère une publication par identifiant, ajoute une publication, récupère une liste de publications et récupère les publications associées à une publication donnée.

Remarque : La fonction Lambda utilise l'`switch`instruction on `event.field` pour déterminer le champ en cours de résolution.

Créez cette fonction Lambda à l'aide de la console de AWS gestion ou d'une AWS CloudFormation pile. Pour créer la fonction à partir d'une CloudFormation pile, vous pouvez utiliser la commande suivante AWS Command Line Interface (AWS CLI) :

```
aws cloudformation create-stack --stack-name AppSyncLambdaExample \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/lambda/
LambdaCFTemplate.yaml \
--capabilities CAPABILITY_NAMED_IAM
```

Vous pouvez également lancer le AWS CloudFormation stack dans la AWS région ouest des États-Unis (Oregon) sur votre AWS compte à partir d'ici :

Configuration d'une source de données pour Lambda

Après avoir créé la fonction Lambda, accédez à votre API GraphQL dans la AWS AppSync console, puis choisissez l'onglet Sources de données.

Choisissez Créer une source de données, entrez un nom de source de données convivial (par exemple,**`Lambda`**), puis pour Type de source de données, choisissez AWS Lambdafonction. Pour Région, choisissez la même région que votre fonction. (Si vous avez créé la fonction à partir de la CloudFormation pile fournie, elle se trouve probablement dans US-WEST-2.) Pour Function ARN, choisissez le nom de ressource Amazon (ARN) de votre fonction Lambda.

Après avoir choisi votre fonction Lambda, vous pouvez soit créer un nouveau rôle AWS Identity and Access Management (IAM) (pour lequel les autorisations appropriées sont AWS AppSync attribuées), soit choisir un rôle existant doté de la politique intégrée suivante :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
```

```
            "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/LAMBDA_FUNCTION"
        }
    ]
}
```

Vous devez également établir une relation de confiance avec AWS AppSync le rôle IAM comme suit :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "appsync.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

Création d'un schéma GraphQL

Maintenant que la source de données est connectée à votre fonction Lambda, créez un schéma GraphQL.

Dans l'éditeur de schéma de la AWS AppSync console, assurez-vous que votre schéma correspond au schéma suivant :

```
schema {
    query: Query
    mutation: Mutation
}

type Query {
    getPost(id:ID!): Post
    allPosts: [Post]
}

type Mutation {
    addPost(id: ID!, author: String!, title: String, content: String, url: String):
 Post!
}
```

```
type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    relatedPosts: [Post]
}
```

Configuration des résolveurs

Maintenant que vous avez enregistré une source de données Lambda et un schéma GraphQL valide, vous pouvez connecter vos champs GraphQL à votre source de données Lambda à l'aide de résolveurs.

Pour créer un résolveur, vous aurez besoin de modèles de mappage. Pour en savoir plus sur les modèles de mappage, consultezResolver Mapping Template Overview.

Pour plus d'informations sur les modèles de mappage Lambda, consultez. Resolver mapping template reference for Lambda

Au cours de cette étape, vous associez un résolveur à la fonction Lambda pour les champs suivants `getPost(id:ID!): Post` :`allPosts: [Post]`,`addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`, et. `Post.relatedPosts: [Post]`

Dans l'éditeur de schéma de la AWS AppSync console, sur le côté droit, choisissez Attach Resolver for`getPost(id:ID!): Post`.

Ensuite, dans le menu Action, choisissez Update runtime, puis Unit Resolver (VTL uniquement).

Ensuite, choisissez votre source de données Lambda. Dans la section modèle de mappage de demande, choisissez Invoquer et transférer des arguments.

Modifiez l'objet `payload` pour ajouter le nom du champ. Votre modèle doit ressembler au suivant :

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
```

```
    "payload": {
        "field": "getPost",
        "arguments":  $utils.toJson($context.arguments)
    }
}
```

Dans la section modèle de mappage de réponse section, choisissez Renvoyer le résultat de Lambda.

Dans ce cas, utilisez le modèle de base en l'état. Il devrait se présenter comme suit :

```
$utils.toJson($context.result)
```

Choisissez Save (Enregistrer). Vous avez joint votre première résolveur avec succès. Répétez cette opération pour les autres champs comme suit :

Pour le modèle de mappage de demande `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` :

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "addPost",
        "arguments":  $utils.toJson($context.arguments)
    }
}
```

Pour le modèle de mappage de réponse `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` :

```
$utils.toJson($context.result)
```

Pour le modèle de mappage de demande `allPosts: [Post]` :

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "allPosts"
    }
}
```

Pour le modèle de mappage de réponse `allPosts: [Post]` :

```
$utils.toJson($context.result)
```

Pour le modèle de mappage de demande `Post.relatedPosts: [Post]` :

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "relatedPosts",
        "source":  $utils.toJson($context.source)
    }
}
```

Pour le modèle de mappage de réponse `Post.relatedPosts: [Post]` :

```
$utils.toJson($context.result)
```

Testez votre API GraphQL

Maintenant que votre fonction Lambda est connectée aux résolveurs GraphQL, vous pouvez exécuter des mutations et des requêtes à l'aide de la console ou d'une application cliente.

Sur le côté gauche de la AWS AppSync console, choisissez Requêtes, puis collez le code suivant :

addPost Mutation

```
mutation addPost {
    addPost(
        id: 6
        author: "Author6"
        title: "Sixth book"
        url: "https://www.amazon.com/"
        content: "This is the book is a tutorial for using GraphQL with AWS AppSync."
    ) {
        id
        author
        title
        content
        url
        ups
```

```
        downs
    }
}
```

getPost Query

```
query getPost {
    getPost(id: "2") {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

allPosts Query

```
query allPosts {
    allPosts {
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {
            id
            title
        }
    }
}
```

Erreurs de renvoi

Toute résolution de champ donnée peut entraîner une erreur. Avec AWS AppSync, vous pouvez générer des erreurs provenant des sources suivantes :

- Modèle de mappage de demande ou de réponse

- Fonction Lambda

À partir du modèle de mappage

Pour signaler des erreurs intentionnelles, vous pouvez utiliser la méthode d'`$utils.error`assistance du modèle Velocity Template Language (VTL). Elle prend comme un argument un `errorMessage`, un `errorType` et une valeur `data` facultative. L'objet `data` est utile pour renvoyer des données supplémentaires au client, lorsqu'une erreur a été déclenchée. L'objet `data` sera ajouté à `errors` dans la réponse GraphQL finale.

L'exemple suivant montre comment utiliser le modèle de mappage de réponse dans le `Post.relatedPosts: [Post]` :

```
$utils.error("Failed to fetch relatedPosts", "LambdaFailure", $context.result)
```

Cela génère une réponse GraphQL similaire à ce qui suit :

```
{
    "data": {
        "allPosts": [
            {
                "id": "2",
                "title": "Second book",
                "relatedPosts": null
            },
            ...
        ]
    },
    "errors": [
        {
            "path": [
                "allPosts",
                0,
                "relatedPosts"
            ],
            "errorType": "LambdaFailure",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
```

```
            ],
            "message": "Failed to fetch relatedPosts",
            "data": [
                {
                  "id": "2",
                  "title": "Second book"
                },
                {
                  "id": "1",
                  "title": "First book"
                }
            ]
        }
    ]
}
```

où `allPosts[0].relatedPosts` est null du fait de l'erreur et `errorMessage`, `errorType` et `data` sont présents dans l'objet `data.errors[0]`.

À partir de la fonction Lambda

AWS AppSync comprend également les erreurs générées par la fonction Lambda. Le modèle de programmation Lambda vous permet de signaler les erreurs gérées. Si la fonction Lambda génère une erreur, elle AWS AppSync ne parvient pas à résoudre le champ actuel. Seul le message d'erreur renvoyé par Lambda est défini dans la réponse. Actuellement, vous ne pouvez pas renvoyer de données superflues au client en déclenchant une erreur à partir de la fonction Lambda.

Remarque : Si votre fonction Lambda génère une erreur non gérée, AWS AppSync utilise le message d'erreur défini par Lambda.

La fonction Lambda suivante génère une erreur :

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    callback("I fail. Always.");
};
```

Cela renvoie une réponse GraphQL similaire à ce qui suit :

```
{
    "data": {
        "allPosts": [
```

```
            {
                "id": "2",
                "title": "Second book",
                "relatedPosts": null
            },
            ...
        ]
    },
    "errors": [
        {
            "path": [
                "allPosts",
                0,
                "relatedPosts"
            ],
            "errorType": "Lambda:Handled",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ],
            "message": "I fail. Always."
        }
    ]
}
```

Cas d'utilisation avancé : traitement par lots

Dans cet exemple, la fonction Lambda possède un `relatedPosts` champ qui renvoie une liste de publications connexes pour une publication donnée. Dans les exemples de requêtes, l'invocation du `allPosts` champ par la fonction Lambda renvoie cinq messages. Comme nous avons précisé que nous voulions également résoudre le problème `relatedPosts` pour chaque message renvoyé, l'opération de `relatedPosts` terrain est invoquée cinq fois.

```
query allPosts {
    allPosts {   // 1 Lambda invocation - yields 5 Posts
        id
        author
        title
        content
        url
```

```
        ups
        downs
        relatedPosts {   // 5 Lambda invocations - each yields 5 posts
            id
            title
        }
    }
}
```

Bien que cela ne semble pas important dans cet exemple spécifique, ce surchargement aggravé peut rapidement saper l'application.

Si vous souhaitez à nouveau extraire `relatedPosts` sur le `Posts` associé renvoyé dans la même requête, le nombre d'appels augmenterait considérablement.

```
query allPosts {
    allPosts {   // 1 Lambda invocation - yields 5 Posts
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {   // 5 Lambda invocations - each yield 5 posts = 5 x 5 Posts
            id
            title
            relatedPosts {  // 5 x 5 Lambda invocations - each yield 5 posts = 25 x 5
 Posts
                id
                title
                author
            }
        }
    }
}
```

Dans cette requête relativement simple, AWS AppSync invoquerait la fonction Lambda 1 + 5 + 25 = 31 fois.

Il s'agit d'un défi assez courant, souvent appelé « problème N+1 » (dans ce cas, N = 5) et qui peut entraîner une augmentation de la latence et des coûts de l'application.

L'une des approches possibles pour résoudre ce problème est de regrouper les demandes de résolveur de champ similaires. Dans cet exemple, au lieu de demander à la fonction Lambda de résoudre une liste de publications connexes pour une publication donnée, elle pourrait résoudre une liste de publications connexes pour un lot de publications donné.

Pour démontrer cela, nous allons passer le résolveur `Post.relatedPosts: [Post]` à un résolveur capable d'effectuer des traitements par lots.

Dans la console AWS AppSync , sur le côté droit, choisissez le résolveur `Post.relatedPosts: [Post]` existant. Modifiez le modèle de mappage de demande comme suit :

```
{
    "version": "2017-02-28",
    "operation": "BatchInvoke",
    "payload": {
        "field": "relatedPosts",
        "source":  $utils.toJson($context.source)
    }
}
```

Seul le champ `operation` est passé de `Invoke` à `BatchInvoke`. Le champ de charge utile devient désormais un tableau de tout ce qui est spécifié dans le modèle. Dans cet exemple, la fonction Lambda reçoit les informations suivantes en entrée :

```
[
    {
        "field": "relatedPosts",
        "source": {
            "id": 1
        }
    },
    {
        "field": "relatedPosts",
        "source": {
            "id": 2
        }
    },
    ...
]
```

Lorsqu'elle `BatchInvoke` est spécifiée dans le modèle de mappage des demandes, la fonction Lambda reçoit une liste de demandes et renvoie une liste de résultats.

Plus précisément, la liste des résultats doit correspondre à la taille et à l'ordre des entrées de charge utile de la demande afin de AWS AppSync pouvoir correspondre aux résultats en conséquence.

Dans cet exemple de traitement par lots, la fonction Lambda renvoie un lot de résultats comme suit :

```
[
    [{"id":"2","title":"Second book"}, {"id":"3","title":"Third book"}],   //
 relatedPosts for id=1
    [{"id":"3","title":"Third book"}]
           // relatedPosts for id=2
]
```

La fonction Lambda suivante dans Node.js illustre cette fonctionnalité de traitement par lots pour le `Post.relatedPosts` champ comme suit :

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    var posts = {
         "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://
amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR
 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100",
 "downs": "10"},
         "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://
amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups":
 "100", "downs": "10"},
         "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null,
 "content": null, "ups": null, "downs": null },
         "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
         "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} };

    var relatedPosts = {
        "1": [posts['4']],
        "2": [posts['3'], posts['5']],
        "3": [posts['2'], posts['1']],
        "4": [posts['2'], posts['1']],
```

```
        "5": []
    };

    console.log("Got a BatchInvoke Request. The payload has %d items to resolve.",
 event.length);
    // event is now an array
    var field = event[0].field;
    switch(field) {
        case "relatedPosts":
            var results = [];
            // the response MUST contain the same number
            // of entries as the payload array
            for (var i=0; i< event.length; i++) {
                console.log("post {}", JSON.stringify(event[i].source));
                results.push(relatedPosts[event[i].source.id]);
            }
            console.log("results {}", JSON.stringify(results));
            callback(null, results);
            break;
        default:
            callback("Unknown field, unable to resolve" + field, null);
            break;
    }
};
```

Renvoi d'erreurs individuelles

Les exemples précédents montrent qu'il est possible de renvoyer une seule erreur à partir de la fonction Lambda ou de générer une erreur à partir des modèles de mappage. Pour les appels par lots, le fait de générer une erreur à partir de la fonction Lambda indique qu'un lot entier a échoué. Cela peut être acceptable pour des scénarios spécifiques dans lesquels une erreur irrécupérable se produit, telle qu'un échec de connexion à un magasin de données. Toutefois, dans les cas où certains éléments du lot réussissent et d'autres échouent, il est possible de renvoyer à la fois des erreurs et des données valides. Étant donné que la réponse par lots AWS AppSync nécessite de répertorier les éléments correspondant à la taille d'origine du lot, vous devez définir une structure de données capable de différencier les données valides d'une erreur.

Par exemple, si la fonction Lambda est censée renvoyer un lot de publications connexes, vous pouvez choisir de renvoyer une liste d'*Response*objets dans laquelle chaque objet comporte des données facultatives, des champs ErrorMessage et ErrorType. Si le champ errorMessage est présent, cela signifie qu'une erreur s'est produite.

Le code suivant montre comment mettre à jour la fonction Lambda :

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    var posts = {
         "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://
amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR
 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100",
 "downs": "10"},
         "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://
amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups":
 "100", "downs": "10"},
         "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null,
 "content": null, "ups": null, "downs": null },
         "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
         "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} };

    var relatedPosts = {
        "1": [posts['4']],
        "2": [posts['3'], posts['5']],
        "3": [posts['2'], posts['1']],
        "4": [posts['2'], posts['1']],
        "5": []
    };

    console.log("Got a BatchInvoke Request. The payload has %d items to resolve.",
 event.length);
    // event is now an array
    var field = event[0].field;
    switch(field) {
        case "relatedPosts":
            var results = [];
            results.push({ 'data': relatedPosts['1'] });
            results.push({ 'data': relatedPosts['2'] });
            results.push({ 'data': null, 'errorMessage': 'Error Happened', 'errorType':
 'ERROR' });
            results.push(null);
            results.push({ 'data': relatedPosts['3'], 'errorMessage': 'Error Happened
 with last result', 'errorType': 'ERROR' });
```

```
            callback(null, results);
            break;
        default:
            callback("Unknown field, unable to resolve" + field, null);
            break;
    }
};
```

Dans cet exemple, le modèle de mappage des réponses suivant analyse chaque élément de la fonction Lambda et signale les erreurs éventuelles :

```
#if( $context.result && $context.result.errorMessage )
    $utils.error($context.result.errorMessage, $context.result.errorType,
 $context.result.data)
#else
    $utils.toJson($context.result.data)
#end
```

Cet exemple renvoie une réponse GraphQL similaire à ce qui suit :

```
{
  "data": {
    "allPosts": [
      {
        "id": "1",
        "relatedPostsPartialErrors": [
          {
            "id": "4",
            "title": "Fourth book"
          }
        ]
      },
      {
        "id": "2",
        "relatedPostsPartialErrors": [
          {
            "id": "3",
            "title": "Third book"
          },
          {
            "id": "5",
            "title": "Fifth book"
          }
```

```
        ]
      },
      {
        "id": "3",
        "relatedPostsPartialErrors": null
      },
      {
        "id": "4",
        "relatedPostsPartialErrors": null
      },
      {
        "id": "5",
        "relatedPostsPartialErrors": null
      }
    ]
  },
  "errors": [
    {
      "path": [
        "allPosts",
        2,
        "relatedPostsPartialErrors"
      ],
      "errorType": "ERROR",
      "locations": [
        {
          "line": 4,
          "column": 9
        }
      ],
      "message": "Error Happened"
    },
    {
      "path": [
        "allPosts",
        4,
        "relatedPostsPartialErrors"
      ],
      "data": [
        {
          "id": "2",
          "title": "Second book"
        },
        {
```

```
          "id": "1",
          "title": "First book"
        }
      ],
      "errorType": "ERROR",
      "locations": [
        {
          "line": 4,
          "column": 9
        }
      ],
      "message": "Error Happened with last result"
    }
  ]
}
```

Configuration de la taille de lot maximale

Par défaut, lors de l'utilisation`BatchInvoke`, AWS AppSync envoie des requêtes à votre fonction Lambda par lots de cinq éléments maximum. Vous pouvez configurer la taille de lot maximale de vos résolveurs Lambda.

Pour configurer la taille de lot maximale sur un résolveur, utilisez la commande suivante dans le AWS Command Line Interface () AWS CLI :

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
 relatedPosts \
 --request-mapping-template "<template>" --response-mapping-template "<template>" --
data-source-name "<lambda-datasource>" \
 --max-batch-size X
```

> **Note**
>
> Lorsque vous fournissez un modèle de mappage de demandes, vous devez utiliser l'`BatchInvoke`opération pour utiliser le traitement par lots.

Vous pouvez également utiliser la commande suivante pour activer et configurer le traitement par lots sur les résolveurs Lambda directs :

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
 relatedPosts \
 --data-source-name "<lambda-datasource>" \
 --max-batch-size X
```

Configuration de la taille de lot maximale avec des modèles VTL

Pour les résolveurs Lambda dotés de modèles VTL intégrés à la demande, la taille de lot maximale n'aura aucun effet à moins qu'ils ne l'aient directement spécifiée en tant qu'opération dans VTL. `BatchInvoke` De même, si vous effectuez une mutation de haut niveau, le traitement par lots n'est pas effectué pour les mutations car la spécification GraphQL exige que les mutations parallèles soient exécutées de manière séquentielle.

Par exemple, prenons les mutations suivantes :

```
type Mutation {
    putItem(input: Item): Item
    putItems(inputs: [Item]): [Item]
}
```

En utilisant la première mutation, nous pouvons créer 10 `Items` comme indiqué dans l'extrait ci-dessous :

```
mutation MyMutation {
    v1: putItem($someItem1) {
        id,
        name
    }
    v2: putItem($someItem2) {
        id,
        name
    }
    v3: putItem($someItem3) {
        id,
        name
    }
    v4: putItem($someItem4) {
        id,
        name
    }
    v5: putItem($someItem5) {
```

```
        id,
        name
    }
    v6: putItem($someItem6) {
        id,
        name
    }
    v7: putItem($someItem7) {
        id,
        name
    }
    v8: putItem($someItem8) {
        id,
        name
    }
    v9: putItem($someItem9) {
        id,
        name
    }
    v10: putItem($someItem10) {
        id,
        name
    }
}
```

Dans cet exemple, le ne `Items` sera pas regroupé par lots de 10 même si la taille de lot maximale est définie sur 10 dans le Lambda Resolver. Ils s'exécuteront plutôt de manière séquentielle conformément à la spécification GraphQL.

Pour effectuer une véritable mutation par lots, vous pouvez suivre l'exemple ci-dessous en utilisant la deuxième mutation :

```
mutation MyMutation {
    putItems([$someItem1, $someItem2, $someItem3,$someItem4, $someItem5, $someItem6,
    $someItem7, $someItem8, $someItem9, $someItem10]) {
    id,
    name
    }
}
```

Pour plus d'informations sur l'utilisation du traitement par lots avec les résolveurs Lambda directs, consultez. Résolveurs Lambda directs

Tutoriel : Amazon OpenSearch Service Resolvers

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

AWS AppSync prend en charge l'utilisation d'Amazon OpenSearch Service à partir de domaines que vous avez provisionnés dans votre propre AWS compte, à condition qu'ils n'existent pas au sein d'un VPC. Une fois que vos domaines sont mis en service, vous pouvez vous y connecter à l'aide d'une source de données, puis vous pouvez configurer un résolveur dans le schéma afin d'effectuer des opérations GraphQL telles que des requêtes, des mutations et des abonnements. Ce didacticiel vous présente certains exemples courants.

Pour plus d'informations, consultez la référence du modèle de mappage du résolveur pour OpenSearch.

Configuration en un clic

Pour configurer automatiquement un point de terminaison GraphQL avec AWS AppSync Amazon OpenSearch Service configuré, vous pouvez utiliser ce AWS CloudFormation modèle :

Une fois que le déploiement d'AWS CloudFormation est terminé, vous pouvez passer directement à l'exécution de requêtes et de mutations GraphQL.

Création d'un nouveau domaine OpenSearch de service

Pour commencer à utiliser ce didacticiel, vous avez besoin d'un domaine OpenSearch de service existant. Si vous n'en avez pas, vous pouvez utiliser l'exemple suivant. Notez que la création d'un domaine de OpenSearch service peut prendre jusqu'à 15 minutes avant que vous puissiez passer à son intégration à une source de AWS AppSync données.

```
aws cloudformation create-stack --stack-name AppSyncOpenSearch \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/elasticsearch/
ESResolverCFTemplate.yaml \
```

```
--parameters ParameterKey=OSDomainName,ParameterValue=ddtestdomain
 ParameterKey=Tier,ParameterValue=development \
--capabilities CAPABILITY_NAMED_IAM
```

Vous pouvez lancer le AWS CloudFormation stack suivant dans la région USA West 2 (Oregon) sur votre AWS compte :

Configuration de la source de données pour le OpenSearch service

Une fois le domaine de OpenSearch service créé, accédez à votre API AWS AppSync GraphQL et choisissez l'onglet Sources de données. Choisissez Nouveau et entrez un nom convivial pour la source de données, tel que « oss ». Choisissez ensuite le OpenSearch domaine Amazon pour le type de source de données, choisissez la région appropriée et vous devriez voir votre domaine OpenSearch de service répertorié. Après l'avoir sélectionné, vous pouvez créer un nouveau rôle auquel AWS AppSync attribuera les autorisations appropriées ou choisir un rôle existant, qui dispose de la stratégie en ligne suivante :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1234234",
            "Effect": "Allow",
            "Action": [
                "es:ESHttpDelete",
                "es:ESHttpHead",
                "es:ESHttpGet",
                "es:ESHttpPost",
                "es:ESHttpPut"
            ],
            "Resource": [
                "arn:aws:es:REGION:ACCOUNTNUMBER:domain/democluster/*"
            ]
        }
    ]
}
```

Vous devez également configurer une relation d'approbation avec AWS AppSync pour ce rôle :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "appsync.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

En outre, le domaine de OpenSearch service possède sa propre politique d'accès que vous pouvez modifier via la console Amazon OpenSearch Service. Vous devrez ajouter une politique similaire à la suivante, avec les actions et les ressources appropriées pour le domaine OpenSearch de service. Notez que le principal sera le rôle de source de AppSync données qui, si vous laissez la console le créer, se trouve dans la console IAM.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Effect": "Allow",
        "Principal": {
            "AWS": "arn:aws:iam::ACCOUNTNUMBER:role/service-role/
APPSYNC_DATASOURCE_ROLE"
        },
        "Action": [
            "es:ESHttpDelete",
            "es:ESHttpHead",
            "es:ESHttpGet",
            "es:ESHttpPost",
            "es:ESHttpPut"
        ],
        "Resource": "arn:aws:es:REGION:ACCOUNTNUMBER:domain/DOMAIN_NAME/*"
        }
    ]
}
```

Connexion d'un résolveur

Maintenant que la source de données est connectée à votre domaine de OpenSearch service, vous pouvez la connecter à votre schéma GraphQL à l'aide d'un résolveur, comme illustré dans l'exemple suivant :

```
 schema {
   query: Query
   mutation: Mutation
 }

 type Query {
   getPost(id: ID!): Post
   allPosts: [Post]
 }

 type Mutation {
   addPost(id: ID!, author: String, title: String, url: String, ups: Int, downs: Int,
 content: String): AWSJSON
 }

type Post {
  id: ID!
  author: String
  title: String
  url: String
  ups: Int
  downs: Int
  content: String
}
...
```

Notez qu'il y a un type `Post` défini par l'utilisateur avec un champ `id`. Dans les exemples suivants, nous supposons qu'il existe un processus (qui peut être automatisé) permettant de placer ce type dans votre domaine de OpenSearch service, qui correspondrait à une racine de chemin de`/post/_doc`, où se `post` trouve l'index. À partir de ce chemin racine, vous pouvez effectuer des recherches de documents individuels, des recherches par `/id/post*` caractères génériques ou des recherches multidocuments avec un chemin de. `/post/_search` Par exemple, si un autre type est appelé`User`, vous pouvez indexer les documents sous un nouvel index appelé`user`, puis effectuer des recherches avec un chemin de`/user/_search`.

Dans l'éditeur de schéma de la console AWS AppSync , modifiez le schéma `Posts` précédent en y incluant une requête `searchPosts` :

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

Enregistrez le schéma. Sur le côté droit, pour `searchPosts`, choisissez Attach resolver (Attacher un résolveur). Dans le menu Action, choisissez Update runtime, puis Unit Resolver (VTL uniquement). Choisissez ensuite votre source OpenSearch de données de service. Sous la section request mapping template (modèle de mappage de demande), sélectionnez dans la liste déroulante les Query posts (billets de requête) pour obtenir un modèle de base. Modifiez l'élément `path` en lui donnant la valeur `/post/_search`. Il devrait se présenter comme suit :

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/post/_search",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "from":0,
            "size":50
        }
    }
}
```

Cela suppose que le schéma précédent contient des documents indexés dans OpenSearch Service sous le `post` champ. Si vous structurez vos données différemment, vous devrez faire la mise à jour correspondante.

Dans la section du modèle de mappage des réponses, vous devez spécifier le `_source` filtre approprié si vous souhaitez récupérer les résultats des données d'une requête de OpenSearch service et les traduire vers GraphQL. Utilisez le modèle suivant :

```
[
    #foreach($entry in $context.result.hits.hits)
```

```
    #if( $velocityCount > 1 ) , #end
    $utils.toJson($entry.get("_source"))
    #end
]
```

Modification de vos recherches

Le modèle de mappage de demande précédent exécute une requête simple pour tous les enregistrements. Supposons que vous souhaitiez effectuer une recherche en fonction d'un auteur spécifique. Supposons également que vous vouliez que l'auteur soit un argument défini dans votre requête GraphQL. Dans l'éditeur de schéma de la console AWS AppSync, ajoutez une requête `allPostsByAuthor` :

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

Choisissez maintenant Attach resolver et sélectionnez la source de données du OpenSearch service, mais utilisez l'exemple suivant dans le modèle de mappage des réponses :

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/post/_search",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "from":0,
            "size":50,
            "query":{
                "match" :{
                    "author": $util.toJson($context.arguments.author)
                }
            }
        }
    }
}
```

Notez que l'élément `body` est renseigné avec une requête terminologique concernant le champ `author`, qui est transmise à partir du client en tant qu'argument. Vous pouvez, le cas échéant, avoir certaines informations pré-renseignées, telles que le texte standard, ou encore utiliser d'autres utilitaires.

Si vous utilisez ce résolveur, renseignez la section response mapping template (modèle de mappage de réponse) avec les mêmes informations que dans l'exemple précédent.

Ajouter des données au OpenSearch service

Vous souhaiterez peut-être ajouter des données à votre domaine OpenSearch de service à la suite d'une mutation GraphQL. Il s'agit d'un puissant mécanisme de recherche, qui peut également avoir d'autres fonctions. Comme vous pouvez utiliser les abonnements GraphQL pour générer vos données en temps réel, il s'agit d'un mécanisme permettant d'informer les clients des mises à jour des données de votre OpenSearch domaine de service.

Revenez à la page Schéma de la console AWS AppSync , puis sélectionnez Joindre un résolveur pour la mutation `addPost()`. Sélectionnez à nouveau la source de données du OpenSearch service et utilisez le modèle de mappage des réponses suivant pour le `Posts` schéma :

```
{
    "version":"2017-02-28",
    "operation":"PUT",
    "path": $util.toJson("/post/_doc/$context.arguments.id"),
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "id": $util.toJson($context.arguments.id),
            "author": $util.toJson($context.arguments.author),
            "ups": $util.toJson($context.arguments.ups),
            "downs": $util.toJson($context.arguments.downs),
            "url": $util.toJson($context.arguments.url),
            "content": $util.toJson($context.arguments.content),
            "title": $util.toJson($context.arguments.title)
        }
    }
}
```

Comme auparavant, il s'agit d'un exemple de la façon dont vos données peuvent être structurées. Si vous avez des noms de champs ou des index différents, vous devez mettre à jour les éléments `path`

et body en conséquence. Cet exemple montre également comment utiliser `$context.arguments` pour renseigner le modèle à partir de vos arguments de mutation GraphQL.

Avant de poursuivre, utilisez le modèle de mappage des réponses suivant, qui renverra le résultat de l'opération de mutation ou les informations d'erreur en sortie :

```
#if($context.error)
    $util.toJson($ctx.error)
#else
    $util.toJson($context.result)
#end
```

Récupération d'un document unique

Enfin, si vous souhaitez utiliser la requête `getPost(id:ID)` dans votre schéma pour renvoyer un document individuel, recherchez cette requête dans l'éditeur de schéma de la console AWS AppSync et choisissez Joindre un résolveur. Sélectionnez à nouveau la source de données du OpenSearch service et utilisez le modèle de mappage suivant :

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path": $util.toJson("post/_doc/$context.arguments.id"),
    "params":{
        "headers":{},
        "queryString":{},
        "body":{}
    }
}
```

Étant donné que l'élément `path` ci-dessus utilise l'argument `id` avec un corps vide, c'est un document unique qui est renvoyé. Cependant, vous devez utiliser le modèle de mappage de réponse suivant, car vous souhaitez renvoyer un seul élément et non une liste :

```
$utils.toJson($context.result.get("_source"))
```

Effectuer des requêtes et des mutations

Vous devriez maintenant être en mesure d'effectuer des opérations GraphQL sur votre domaine de OpenSearch service. Accédez à l'onglet Requêtes de la console AWS AppSync et ajoutez un nouvel enregistrement :

```
mutation addPost {
    addPost (
        id:"12345"
        author: "Fred"
        title: "My first book"
        content: "This will be fun to write!"
        url: "publisher website",
        ups: 100,
        downs:20
       )
}
```

Vous verrez le résultat de la mutation sur la droite. De même, vous pouvez désormais exécuter une `searchPosts` requête sur votre domaine OpenSearch de service :

```
query searchPosts {
    searchPosts {
        id
        title
        author
        content
    }
}
```

Bonnes pratiques

- OpenSearch Le service doit être destiné à interroger des données, et non en tant que base de données principale. Vous souhaiterez peut-être utiliser le OpenSearch Service conjointement avec Amazon DynamoDB, comme indiqué dans Combining GraphQL Resolvers.
- Vous devez uniquement accorder l'accès à votre domaine en autorisant le rôle de service AWS AppSync à accéder au cluster.
- Vous pouvez commencer la phase de développement de façon modeste, avec le cluster le moins onéreux, puis passer à un plus grand cluster à haute disponibilité (HA) lorsque vous passerez en production.

Didacticiel : Résolveurs locaux

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

AWS AppSync vous permet d'utiliser des sources de données prises en charge (AWS LambdaAmazon DynamoDB ou OpenSearch Amazon Service) pour effectuer diverses opérations. Cependant, dans certains cas, un appel à une source de données prise en charge peut ne pas être nécessaire.

C'est là où le résolveur local se révèle pratique. Au lieu d'appeler une source de données distante, le résolveur local achemine simplement le résultat du modèle de mappage de la demande vers le modèle de mappage de la réponse. La résolution du champ ne quitte pas AWS AppSync.

Les résolveurs locaux sont utiles dans plusieurs scénarios. Le scénario le plus répandu consiste à publier des notifications sans déclencher d'appel de source de données. Pour illustrer ce cas d'utilisation, nous allons créer une application de pagination, où les utilisateurs peuvent paginer entre eux. Comme cet exemple met à profit les Abonnements, si vous n'êtes pas familiarisé avec les Abonnements, vous pouvez suivre le didacticiel Données en temps réel.

Création de l'application de pagination

Dans notre application de pagination, les clients peuvent s'abonner à une boîte de réception et envoyer des pages à d'autres clients. Chaque page comprend un message. Voici le schéma :

```
schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
}

type Subscription {
    inbox(to: String!): Page
    @aws_subscribe(mutations: ["page"])
}

type Mutation {
```

```
    page(body: String!, to: String!): Page!
}

type Page {
    from: String
    to: String!
    body: String!
    sentAt: String!
}

type Query {
    me: String
}
```

Attachons un résolveur au champ `Mutation.page`. Dans le volet Schema (Schéma), cliquez sur Attach Resolver (Attacher un résolveur) en regard de la définition du champ sur le volet droit. Créez une nouvelle source de données de type None et nommez-la PageDataSource.

Pour le modèle de mappage de la demande, saisissez :

```
{
  "version": "2017-02-28",
  "payload": {
    "body": $util.toJson($context.arguments.body),
    "from": $util.toJson($context.identity.username),
    "to":  $util.toJson($context.arguments.to),
    "sentAt": "$util.time.nowISO8601()"
  }
}
```

Et pour le modèle de mappage de la réponse, sélectionnez la valeur par défaut Forward the result (Transmettre le résultat). Enregistrez votre résolveur. Votre application est désormais prête, commençons à paginer !

Envoi et abonnement aux pages

Pour que les clients reçoivent des pages, ils doivent d'abord s'abonner à une boîte de réception.

Dans le volet Requêtes, exécutons l'abonnement `inbox` :

```
subscription Inbox {
    inbox(to: "Nadia") {
```

```
        body
        to
        from
        sentAt
    }
}
```

Nadia recevra les pages chaque fois que la mutation `Mutation.page` est appelée. Appelons la mutation en l'exécutant :

```
mutation Page {
    page(to: "Nadia", body: "Hello, World!") {
        body
        to
        from
        sentAt
    }
}
```

Nous venons juste d'illustrer l'utilisation des résolveurs locaux, en envoyant une page et en la recevant sans quitter AWS AppSync.

Didacticiel : Association de résolveurs GraphQL

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Les résolveurs et les champs d'un schéma GraphQL possèdent des relations 1:1 avec un haut niveau de flexibilité. Étant donné qu'une source de données est configurée sur un résolveur indépendamment d'un schéma, les types GraphQL peuvent être résolus ou manipulés via différentes sources de données, en les combinant dans un schéma afin de répondre au mieux à vos besoins.

Les exemples de scénarios suivants montrent comment mélanger et associer des sources de données dans votre schéma. Avant de commencer, nous vous recommandons de vous familiariser avec la configuration des sources de données et des résolveurs pour AWS Lambda Amazon DynamoDB OpenSearch et Amazon Service, comme décrit dans les didacticiels précédents.

Exemple de schéma

Le schéma suivant est de type `Post` avec 3 `Query` opérations et 3 `Mutation` opérations définies :

```
type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
}

type Query {
    allPost: [Post]
    getPost(id: ID!): Post
    searchPosts: [Post]
}

type Mutation {
    addPost(
        id: ID!,
        author: String!,
        title: String,
        content: String,
        url: String
    ): Post
    updatePost(
        id: ID!,
        author: String!,
        title: String,
        content: String,
        url: String,
        ups: Int!,
        downs: Int!,
        expectedVersion: Int!
    ): Post
    deletePost(id: ID!): Post
}
```

Dans cet exemple, vous disposez d'un total de 6 résolveurs à joindre. Une solution possible serait de faire en sorte que tous ces éléments proviennent d'une table Amazon DynamoDB, `Posts` appelée, `AllPosts` où exécute un scan `searchPosts` et exécute une requête, comme indiqué dans la référence du modèle de mappage du résolveur DynamoDB. Cependant, il existe des alternatives pour répondre aux besoins de votre entreprise, comme la résolution de ces requêtes GraphQL à partir de Lambda ou de Service. OpenSearch

Modifier les données via les résolveurs

Il se peut que vous deviez renvoyer les résultats d'une base de données telle que DynamoDB (ou Amazon Aurora) à des clients dont certains attributs ont été modifiés. Cela peut être dû à la mise en forme des types de données (par exemple, différences d'horodatage entre les clients) ou à des problèmes de rétrocompatibilité. À titre d'illustration, dans l'exemple suivant, une AWS Lambda fonction manipule les votes positifs et négatifs pour les articles de blog en leur attribuant des nombres aléatoires chaque fois que le résolveur GraphQL est invoqué :

```
'use strict';
const doc = require('dynamodb-doc');
const dynamo = new doc.DynamoDB();

exports.handler = (event, context, callback) => {
    const payload = {
        TableName: 'Posts',
        Limit: 50,
        Select: 'ALL_ATTRIBUTES',
    };

    dynamo.scan(payload, (err, data) => {
        const result = { data: data.Items.map(item =>{
            item.ups = parseInt(Math.random() * (50 - 10) + 10, 10);
            item.downs = parseInt(Math.random() * (20 - 0) + 0, 10);
            return item;
        }) };
        callback(err, result.data);
    });
};
```

Il s'agit d'une fonction Lambda parfaitement valide qui peut être attachée à un champ `AllPosts` dans le schéma GraphQL afin que toute requête renvoyant tous les résultats obtienne des nombres aléatoires pour les pour et/ou les contre.

DynamoDB et service OpenSearch

Pour certaines applications, vous pouvez effectuer des mutations ou de simples requêtes de recherche sur DynamoDB et demander à un processus en arrière-plan de transférer des documents vers Service. OpenSearch Vous pouvez ensuite simplement associer le `searchPosts` résolveur à la source de données du OpenSearch service et renvoyer les résultats de recherche (à partir de données provenant de DynamoDB) à l'aide d'une requête GraphQL. Cela peut être extrêmement intéressant lorsque vous ajoutez des opérations de recherche avancée à vos applications, telles que des mots-clés, des recherches de correspondance partielle ou même des recherches géospatiales. Le transfert de données depuis DynamoDB peut être effectué via un processus ETL ou vous pouvez également diffuser des données depuis DynamoDB à l'aide de Lambda. Vous pouvez en lancer un exemple complet en utilisant la AWS CloudFormation pile suivante dans la région USA West 2 (Oregon) sur votre AWS compte :

Le schéma de cet exemple vous permet d'ajouter des publications à l'aide d'un résolveur DynamoDB comme suit :

```
mutation add {
    putPost(author:"Nadia"
        title:"My first post"
        content:"This is some test content"
        url:"https://aws.amazon.com/appsync/"
    ){
        id
        title
    }
}
```

Cela écrit des données dans DynamoDB, qui les diffuse ensuite via Lambda vers OpenSearch Amazon Service, dans lequel vous pouvez rechercher tous les articles par différents champs. Par exemple, comme les données se trouvent dans Amazon OpenSearch Service, vous pouvez effectuer une recherche dans les champs d'auteur ou de contenu avec du texte libre, même avec des espaces, comme suit :

```
query searchName{
    searchAuthor(name:"   Nadia   "){
        id
```

```
        title
        content
    }
}

query searchContent{
    searchContent(text:"test"){
        id
        title
        content
    }
}
```

Les données étant écrites directement dans DynamoDB, vous pouvez toujours effectuer des opérations efficaces de recherche de listes ou d'éléments par rapport à la table contenant `allPosts{...}` les requêtes et. `singlePost{...}` Cette pile utilise l'exemple de code suivant pour les flux DynamoDB :

Remarque : ce code est fourni à titre d'exemple uniquement.

```
var AWS = require('aws-sdk');
var path = require('path');
var stream = require('stream');

var esDomain = {
    endpoint: 'https://opensearch-domain-name.REGION.es.amazonaws.com',
    region: 'REGION',
    index: 'id',
    doctype: 'post'
};

var endpoint = new AWS.Endpoint(esDomain.endpoint)
var creds = new AWS.EnvironmentCredentials('AWS');

function postDocumentToES(doc, context) {
    var req = new AWS.HttpRequest(endpoint);

    req.method = 'POST';
    req.path = '/_bulk';
    req.region = esDomain.region;
    req.body = doc;
    req.headers['presigned-expires'] = false;
    req.headers['Host'] = endpoint.host;
```

```
    // Sign the request (Sigv4)
    var signer = new AWS.Signers.V4(req, 'es');
    signer.addAuthorization(creds, new Date());

    // Post document to ES
    var send = new AWS.NodeHttpClient();
    send.handleRequest(req, null, function (httpResp) {
        var body = '';
        httpResp.on('data', function (chunk) {
            body += chunk;
        });
        httpResp.on('end', function (chunk) {
            console.log('Successful', body);
            context.succeed();
        });
    }, function (err) {
        console.log('Error: ' + err);
        context.fail();
    });
}

exports.handler = (event, context, callback) => {
    console.log("event => " + JSON.stringify(event));
    var posts = '';

    for (var i = 0; i < event.Records.length; i++) {
        var eventName = event.Records[i].eventName;
        var actionType = '';
        var image;
        var noDoc = false;
        switch (eventName) {
            case 'INSERT':
                actionType = 'create';
                image = event.Records[i].dynamodb.NewImage;
                break;
            case 'MODIFY':
                actionType = 'update';
                image = event.Records[i].dynamodb.NewImage;
                break;
            case 'REMOVE':
            actionType = 'delete';
                image = event.Records[i].dynamodb.OldImage;
                noDoc = true;
```

```
                break;
        }

        if (typeof image !== "undefined") {
            var postData = {};
            for (var key in image) {
                if (image.hasOwnProperty(key)) {
                    if (key === 'postId') {
                        postData['id'] = image[key].S;
                    } else {
                        var val = image[key];
                        if (val.hasOwnProperty('S')) {
                            postData[key] = val.S;
                        } else if (val.hasOwnProperty('N')) {
                            postData[key] = val.N;
                        }
                    }
                }
            }

            var action = {};
            action[actionType] = {};
            action[actionType]._index = 'id';
            action[actionType]._type = 'post';
            action[actionType]._id = postData['id'];
            posts += [
                JSON.stringify(action),
            ].concat(noDoc?[]:[JSON.stringify(postData)]).join('\n') + '\n';
        }
    }
    console.log('posts:',posts);
    postDocumentToES(posts, context);
};
```

Vous pouvez ensuite utiliser des flux DynamoDB pour l'associer à une table DynamoDB avec une clé primaire de, et toute modification apportée à la source `id` de DynamoDB sera répercutée dans votre domaine de service. OpenSearch Pour plus d'informations sur la configuration de ce processus, consultez la Documentation DynamoDB Streams.

Tutoriel : Résolveurs par lots DynamoDB

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

AWS AppSync prend en charge l'utilisation des opérations par lots Amazon DynamoDB sur une ou plusieurs tables d'une même région. Les opérations prises en charge sont `BatchGetItem`, `BatchPutItem` et `BatchDeleteItem`. L'utilisation de ces fonctions dans AWS AppSync vous permet d'exécuter des tâches telles que les suivantes :

- Transmission d'une liste de clés dans une seule requête et renvoi des résultats à partir d'une table
- Lecture des enregistrements à partir d'une ou plusieurs tables dans une seule requête
- Écriture des enregistrements en bloc dans une ou plusieurs tables
- Écriture ou suppression des enregistrements sous condition dans plusieurs tables pouvant avoir une relation

L'utilisation d'opérations par lots avec AWS AppSync DynamoDB est une technique avancée qui nécessite un peu plus de réflexion et de connaissance des opérations de votre backend et des structures de tables. En outre, les opérations par lots dans AWS AppSync présentent deux différences clés par rapport aux opérations ne s'effectuant pas par lots :

- Le rôle de la source de données doit disposer d'autorisations sur toutes les tables auxquelles le résolveur doit accéder.
- La spécification de table d'un résolveur fait partie du modèle de mappage.

Autorisations

Comme pour les autres résolveurs, vous devez créer une source de données dans AWS AppSync et créer un rôle ou utiliser un rôle existant. Les opérations par lots nécessitant des autorisations différentes sur les tables DynamoDB, vous devez accorder au rôle configuré des autorisations pour les actions de lecture ou d'écriture :

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:region:account:table/TABLENAME",
                "arn:aws:dynamodb:region:account:table/TABLENAME/*"
            ]
        }
    ]
}
```

Remarque : Les rôles sont liés aux sources de données dans AWS AppSync et les résolveurs sur les champs sont appelés par rapport à une source de données. Pour simplifier la configuration, une seule table est spécifiée pour les sources de données configurées pour effectuer une extraction par rapport à DynamoDB. Par conséquent, lorsque vous effectuez une opération de traitement par lots sur plusieurs tables dans un seul résolveur (ce qui constitue une tâche plus avancée), vous devez accorder au rôle associé à cette source de données l'accès à toutes les tables avec lesquelles le résolveur devra interagir. Cela doit être effectué dans le champ Resource (Ressource) dans la stratégie IAM ci-dessus. La configuration des tables sur lesquelles les appels par lots doivent être effectués doit être définie dans le modèle de résolveur, dont vous trouverez la description ci-dessous.

Source de données

Dans un souci de simplicité, nous allons utiliser la même source de données pour tous les résolveurs utilisés dans ce didacticiel. Dans l'onglet Sources de données, créez une nouvelle source de données DynamoDB et nommez-la. BatchTutorial Le nom de la table peut être quelconque, car les noms de table sont spécifiés dans le cadre du modèle de mappage de demande pour les opérations par lots. Nous allons nommer la table `empty`.

Dans le cadre de ce didacticiel, n'importe quel rôle avec la stratégie en ligne suivante fonctionne :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
```

```
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:region:account:table/Posts",
                "arn:aws:dynamodb:region:account:table/Posts/*",
                "arn:aws:dynamodb:region:account:table/locationReadings",
                "arn:aws:dynamodb:region:account:table/locationReadings/*",
                "arn:aws:dynamodb:region:account:table/temperatureReadings",
                "arn:aws:dynamodb:region:account:table/temperatureReadings/*"
            ]
        }
    ]
}
```

Traitement par lots sur une table unique

Pour cet exemple, supposons que vous ayez une seule table nommée Posts dans laquelle vous souhaitez ajouter et supprimer des éléments via des opérations par lots. Utilisez le schéma suivant, en notant que pour la requête, nous allons transmettre une liste d'ID :

```
type Post {
    id: ID!
    title: String
}

input PostInput {
    id: ID!
    title: String
}

type Query {
    batchGet(ids: [ID]): [Post]
}

type Mutation {
    batchAdd(posts: [PostInput]): [Post]
    batchDelete(ids: [ID]): [Post]
}

schema {
    query: Query
```

```
    mutation: Mutation
}
```

Attachez un résolveur au champ `batchAdd()` avec le modèle de mappage de demande suivant. Ce processus prend automatiquement chaque élément du type GraphQL `input PostInput` et crée une mappe, qui est nécessaire pour l'opération `BatchPutItem` :

```
#set($postsdata = [])
#foreach($item in ${ctx.args.posts})
    $util.qr($postsdata.add($util.dynamodb.toMapValues($item)))
#end

{
    "version" : "2018-05-29",
    "operation" : "BatchPutItem",
    "tables" : {
        "Posts": $utils.toJson($postsdata)
    }
}
```

Dans cet exemple, le modèle de mappage de réponse est une simple transmission, mais le nom de la table est ajouté sous la forme `..data.Posts` à l'objet de contexte :

```
$util.toJson($ctx.result.data.Posts)
```

À présent, accédez à la page Requêtes de la console AWS AppSync et exécutez la mutation batchAdd suivante :

```
mutation add {
    batchAdd(posts:[{
            id: 1 title: "Running in the Park"},{
            id: 2 title: "Playing fetch"
        }]){
            id
            title
    }
}
```

Vous devriez voir les résultats imprimés à l'écran et vous pouvez valider indépendamment via la console DynamoDB que les deux valeurs ont été écrites dans la table Posts.

Attachez ensuite un résolveur au champ `batchGet()` avec le modèle de mappage de demande suivant. Ce processus prend automatiquement chaque élément du type GraphQL `ids:[]` et crée une mappe, qui est nécessaire pour l'opération `BatchGetItem` :

```
#set($ids = [])
#foreach($id in ${ctx.args.ids})
    #set($map = {})
    $util.qr($map.put("id", $util.dynamodb.toString($id)))
    $util.qr($ids.add($map))
#end

{
    "version" : "2018-05-29",
    "operation" : "BatchGetItem",
    "tables" : {
        "Posts": {
            "keys": $util.toJson($ids),
            "consistentRead": true,
            "projection" : {
                "expression" : "#id, title",
                "expressionNames" : { "#id" : "id"}
                }
        }
    }
}
```

Le modèle de mappage de réponse est à nouveau une simple transmission, avec une fois encore le nom de la table ajouté sous la forme `..data.Posts` à l'objet de contexte :

```
$util.toJson($ctx.result.data.Posts)
```

À présent, revenez à la page Requêtes de la console AWS AppSync et exécutez la requête batchGet suivante :

```
query get {
    batchGet(ids:[1,2,3]){
        id
        title
    }
}
```

Vous devez obtenir les résultats des deux valeurs `id` que vous avez ajoutées précédemment. Notez la valeur `null` renvoyée pour l'élément `id` de valeur `3`. Cela s'explique par le fait qu'il n'y avait encore aucun enregistrement de votre table Posts ayant cette valeur. Notez également qu'AWS AppSync renvoie les résultats dans le même ordre que celui des clés transmises à la requête, ce qui constitue une autre fonction exécutée automatiquement par AWS AppSync . Par conséquent, si vous basculez vers `batchGet(ids:[1,3,2)`, vous verrez l'ordre modifié. Vous saurez également quel `id` a renvoyé une valeur `null`.

Enfin, attachez un résolveur au champ `batchDelete()` avec le modèle de mappage de demande suivant. Ce processus prend automatiquement chaque élément du type GraphQL `ids:[]` et crée une mappe, qui est nécessaire pour l'opération `BatchGetItem` :

```
#set($ids = [])
#foreach($id in ${ctx.args.ids})
    #set($map = {})
    $util.qr($map.put("id", $util.dynamodb.toString($id)))
    $util.qr($ids.add($map))
#end

{
    "version" : "2018-05-29",
    "operation" : "BatchDeleteItem",
    "tables" : {
        "Posts": $util.toJson($ids)
    }
}
```

Le modèle de mappage de réponse est à nouveau une simple transmission, avec une fois encore le nom de la table ajouté sous la forme `..data.Posts` à l'objet de contexte :

```
$util.toJson($ctx.result.data.Posts)
```

À présent, revenez à la page Requêtes de la console AWS AppSync et exécutez la mutation batchDelete suivante :

```
mutation delete {
    batchDelete(ids:[1,2]){ id }
}
```

Les enregistrements contenant les `id` 1 et 2 doivent désormais avoir été supprimés. Si vous exécutez à nouveau la requête `batchGet()` à partir de l'état précédent, le résultat renvoyé devrait être `null`.

Traitement par lots sur plusieurs tables

AWS AppSync vous permet également d'effectuer des opérations par lots sur plusieurs tables. Créons un application plus complexe. Imaginez que nous construisions une application traitant de la santé des animaux domestiques (nommée Pet Health), où des capteurs signalent l'emplacement et la température corporelle des animaux. Les capteurs sont alimentés par piles et tentent de se connecter au réseau toutes les deux ou trois minutes. Lorsqu'un capteur établit une connexion, il envoie ses relevés à notre API AWS AppSync . Des déclencheurs analysent alors les données afin de pouvoir transmettre un tableau de bord au propriétaire de l'animal. Concentrons-nous sur la représentation des interactions entre le capteur et le magasin de données backend.

Comme condition préalable, créons d'abord deux tables DynamoDB : LocationReadings stockera les relevés de position des capteurs et TemperatureReadings stockera les relevés de température des capteurs. Les deux tables partagent la même structure de clé primaire : `sensorId (String)` étant la clé de partition et `timestamp (String)` étant la clé de tri.

Nous allons utiliser le schéma GraphQL suivant :

```
type Mutation {
    # Register a batch of readings
    recordReadings(tempReadings: [TemperatureReadingInput], locReadings:
 [LocationReadingInput]): RecordResult
    # Delete a batch of readings
    deleteReadings(tempReadings: [TemperatureReadingInput], locReadings:
 [LocationReadingInput]): RecordResult
}

type Query {
    # Retrieve all possible readings recorded by a sensor at a specific time
    getReadings(sensorId: ID!, timestamp: String!): [SensorReading]
}

type RecordResult {
    temperatureReadings: [TemperatureReading]
    locationReadings: [LocationReading]
}
```

```
interface SensorReading {
    sensorId: ID!
    timestamp: String!
}

# Sensor reading representing the sensor temperature (in Fahrenheit)
type TemperatureReading implements SensorReading {
    sensorId: ID!
    timestamp: String!
    value: Float
}

# Sensor reading representing the sensor location (lat,long)
type LocationReading implements SensorReading {
    sensorId: ID!
    timestamp: String!
    lat: Float
    long: Float
}

input TemperatureReadingInput {
    sensorId: ID!
    timestamp: String
    value: Float
}

input LocationReadingInput {
    sensorId: ID!
    timestamp: String
    lat: Float
    long: Float
}
```

BatchPutItem - Enregistrement des lectures des capteurs

Nos capteurs doivent être en mesure d'envoyer leurs relevés une fois qu'ils sont connectés à Internet. Le champ GraphQL `Mutation.recordReadings` est l'API qu'ils utilisent à cet effet. Nous allons joindre un résolveur pour donner vie à notre API.

Sélectionnez Attach (Attacher) en regard du champ `Mutation.recordReadings`. Sur l'écran suivant, sélectionnez la source de données `BatchTutorial` créée au début du didacticiel.

Nous ajoutons ensuite le modèle de mappage de demande suivant :

Modèle de mappage de demande

```
## Convert tempReadings arguments to DynamoDB objects
#set($tempReadings = [])
#foreach($reading in ${ctx.args.tempReadings})
    $util.qr($tempReadings.add($util.dynamodb.toMapValues($reading)))
#end

## Convert locReadings arguments to DynamoDB objects
#set($locReadings = [])
#foreach($reading in ${ctx.args.locReadings})
    $util.qr($locReadings.add($util.dynamodb.toMapValues($reading)))
#end

{
    "version" : "2018-05-29",
    "operation" : "BatchPutItem",
    "tables" : {
        "locationReadings": $utils.toJson($locReadings),
        "temperatureReadings": $utils.toJson($tempReadings)
    }
}
```

Comme vous pouvez le voir, l'opération `BatchPutItem` nous permet de spécifier plusieurs tables.

Nous utilisons ensuite le modèle de mappage de réponse suivant :

Modèle de mappage de réponse

```
## If there was an error with the invocation
## there might have been partial results
#if($ctx.error)
    ## Append a GraphQL error for that field in the GraphQL response
    $utils.appendError($ctx.error.message, $ctx.error.message)
#end
## Also returns data for the field in the GraphQL response
$utils.toJson($ctx.result.data)
```

Les opérations par lots peuvent renvoyer à la fois des erreurs et des résultats à la suite de l'appel. Dans ce cas, nous pouvons effectuer certaines opérations de traitement des erreurs supplémentaires.

Remarque : L'utilisation de `$utils.appendError()` est similaire à celle de `$util.error()`, la principale différence résidant dans le fait qu'il n'interrompt pas l'évaluation du modèle de mappage. Au lieu de cela, il signale qu'une erreur s'est produite avec le champ, mais autorise l'évaluation du modèle et renvoie les données à l'appelant. Nous vous recommandons d'utiliser `$utils.appendError()` lorsque votre application doit renvoyer des résultats partiels.

Enregistrez le résolveur et accédez à la page Requêtes de la console AWS AppSync . Nous allons maintenant envoyer quelques relevés des capteurs.

Exécutez la mutation suivante :

```
mutation sendReadings {
  recordReadings(
    tempReadings: [
      {sensorId: 1, value: 85.5, timestamp: "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, value: 85.7, timestamp: "2018-02-01T17:21:06.000+08:00"},
      {sensorId: 1, value: 85.8, timestamp: "2018-02-01T17:21:07.000+08:00"},
      {sensorId: 1, value: 84.2, timestamp: "2018-02-01T17:21:08.000+08:00"},
      {sensorId: 1, value: 81.5, timestamp: "2018-02-01T17:21:09.000+08:00"}
    ]
    locReadings: [
      {sensorId: 1, lat: 47.615063, long: -122.333551, timestamp:
 "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, lat: 47.615163, long: -122.333552, timestamp:
 "2018-02-01T17:21:06.000+08:00"}
      {sensorId: 1, lat: 47.615263, long: -122.333553, timestamp:
 "2018-02-01T17:21:07.000+08:00"}
      {sensorId: 1, lat: 47.615363, long: -122.333554, timestamp:
 "2018-02-01T17:21:08.000+08:00"}
      {sensorId: 1, lat: 47.615463, long: -122.333555, timestamp:
 "2018-02-01T17:21:09.000+08:00"}
    ]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
```

```
  }
}
```

Nous avons envoyé 10 relevés de capteurs dans une mutation, les relevés étant répartis entre deux tables. Utilisez la console DynamoDB pour vérifier que les données apparaissent à la fois dans les tables LocationReadings et TemperatureReadings.

BatchDeleteItem - Suppression des relevés du capteur

Nous pouvons aussi avoir besoin de supprimer des lots de relevés de capteurs. Nous allons utiliser le champ GraphQL `Mutation.deleteReadings` à cet effet. Sélectionnez Attach (Attacher) en regard du champ `Mutation.recordReadings`. Sur l'écran suivant, sélectionnez la source de données `BatchTutorial` créée au début du didacticiel.

Utilisons ensuite le modèle de mappage de demande suivant :

Modèle de mappage de demande

```
## Convert tempReadings arguments to DynamoDB primary keys
#set($tempReadings = [])
#foreach($reading in ${ctx.args.tempReadings})
    #set($pkey = {})
    $util.qr($pkey.put("sensorId", $reading.sensorId))
    $util.qr($pkey.put("timestamp", $reading.timestamp))
    $util.qr($tempReadings.add($util.dynamodb.toMapValues($pkey)))
#end

## Convert locReadings arguments to DynamoDB primary keys
#set($locReadings = [])
#foreach($reading in ${ctx.args.locReadings})
    #set($pkey = {})
    $util.qr($pkey.put("sensorId", $reading.sensorId))
    $util.qr($pkey.put("timestamp", $reading.timestamp))
    $util.qr($locReadings.add($util.dynamodb.toMapValues($pkey)))
#end

{
    "version" : "2018-05-29",
    "operation" : "BatchDeleteItem",
    "tables" : {
        "locationReadings": $utils.toJson($locReadings),
        "temperatureReadings": $utils.toJson($tempReadings)
    }
```

```
}
```

Le modèle de mappage de réponse est le même que celui utilisé pour `Mutation.recordReadings`.

Modèle de mappage de réponse

```
## If there was an error with the invocation
## there might have been partial results
#if($ctx.error)
    ## Append a GraphQL error for that field in the GraphQL response
    $utils.appendError($ctx.error.message, $ctx.error.message)
#end
## Also return data for the field in the GraphQL response
$utils.toJson($ctx.result.data)
```

Enregistrez le résolveur et accédez à la page Requêtes de la console AWS AppSync . Maintenant, nous allons supprimer un ou deux relevés de capteurs.

Exécutez la mutation suivante :

```
mutation deleteReadings {
  # Let's delete the first two readings we recorded
  deleteReadings(
    tempReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]
    locReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}
```

Vérifiez via la console DynamoDB que ces deux relevés ont été supprimés des tables LocationReadings et TemperatureReadings.

BatchGetItem - Récupérez les lectures

Une autre opération courante pour l'application Pet Health consiste à récupérer les relevés d'un capteur à un instant précis. Nous allons joindre un résolveur au champ GraphQL `Query.getReadings` dans notre schéma. Sélectionnez Attach (Attacher) et, sur l'écran suivant, sélectionnez la source de données `BatchTutorial` créée au début du didacticiel.

Nous ajoutons ensuite le modèle de mappage de demande suivant :

Modèle de mappage de demande

```
## Build a single DynamoDB primary key,
## as both locationReadings and tempReadings tables
## share the same primary key structure
#set($pkey = {})
$util.qr($pkey.put("sensorId", $ctx.args.sensorId))
$util.qr($pkey.put("timestamp", $ctx.args.timestamp))

{
    "version" : "2018-05-29",
    "operation" : "BatchGetItem",
    "tables" : {
        "locationReadings": {
            "keys": [$util.dynamodb.toMapValuesJson($pkey)],
            "consistentRead": true
        },
        "temperatureReadings": {
            "keys": [$util.dynamodb.toMapValuesJson($pkey)],
            "consistentRead": true
        }
    }
}
```

Notez que nous utilisons maintenant l'BatchGetItemopération.

Notre modèle de mappage de réponse va être légèrement différent, car nous avons choisi de renvoyer une liste `SensorReading`. Nous allons mapper le résultat de l'appel au format souhaité.

Modèle de mappage de réponse

```
## Merge locationReadings and temperatureReadings
## into a single list
## __typename needed as schema uses an interface
```

```
#set($sensorReadings = [])

#foreach($locReading in $ctx.result.data.locationReadings)
    $util.qr($locReading.put("__typename", "LocationReading"))
    $util.qr($sensorReadings.add($locReading))
#end

#foreach($tempReading in $ctx.result.data.temperatureReadings)
    $util.qr($tempReading.put("__typename", "TemperatureReading"))
    $util.qr($sensorReadings.add($tempReading))
#end

$util.toJson($sensorReadings)
```

Enregistrez le résolveur et accédez à la page Requêtes de la console AWS AppSync . Maintenant, nous allons récupérer les relevés des capteurs.

Exécutez la requête suivante :

```
query getReadingsForSensorAndTime {
  # Let's retrieve the very first two readings
  getReadings(sensorId: 1, timestamp: "2018-02-01T17:21:06.000+08:00") {
    sensorId
    timestamp
    ...on TemperatureReading {
      value
    }
    ...on LocationReading {
      lat
      long
    }
  }
}
```

Nous avons démontré avec succès l'utilisation des opérations par lots DynamoDB à l'aide de. AWS AppSync

Gestion des erreurs

Dans AWS AppSync, les opérations des sources de données peuvent parfois renvoyer des résultats partiels. Le terme résultats partiels est le terme que nous allons utiliser pour désigner une sortie d'opération composée de données et d'une erreur. Étant donné que la gestion des erreurs est par

nature spécifique à l'application, AWS AppSync vous donne la possibilité de gérer les erreurs dans le modèle de mappage de réponse. L'erreur d'appel du résolveur, le cas échéant, est disponible depuis le contexte sous la forme `$ctx.error`. Les erreurs d'appel incluent toujours un message et un type, accessibles sous la forme des propriétés `$ctx.error.message` et `$ctx.error.type`. Pendant l'appel du modèle de mappage de réponse, vous pouvez gérer les résultats partiels de trois manières :

1. Par la digestion de l'erreur d'appel en renvoyant simplement les données.
2. Par le déclenchement d'une erreur (en utilisant `$util.error(...)`) en arrêtant l'évaluation du modèle de mappage de réponse, qui ne renvoie alors aucune donnée.
3. Par l'ajout d'une erreur (en utilisant `$util.appendError(...)`) tout en renvoyant les données.

Nous allons présenter chacun des trois points ci-dessus avec les opérations par lots DynamoDB.

Opérations par lots DynamoDB

Dans le cas des opérations par lots DynamoDB, il est possible qu'un lot ne soit exécuté que partiellement. En d'autres termes, il est possible que certains des éléments ou des clés demandés ne soient pas traités. Si AWS AppSync n'est pas en mesure de terminer l'exécution d'un lot, les éléments non traités et une erreur d'appel sont définis dans le contexte.

Nous allons mettre en œuvre la gestion des erreurs à l'aide de la configuration de champ `Query.getReadings` de l'opération `BatchGetItem` provenant de la section précédente de ce didacticiel. Cette fois, nous allons supposer que, lors de l'exécution du champ `Query.getReadings`, la table DynamoDB `temperatureReadings` a dépassé le débit alloué. DynamoDB a généré ProvisionedThroughputExceededExceptionun à la deuxième tentative pour traiter AWS AppSync les éléments restants du lot.

Le code JSON suivant représente le contexte sérialisé après l'appel de lot DynamoDB, mais avant l'évaluation du modèle de mappage de réponse.

```
{
  "arguments": {
    "sensorId": "1",
    "timestamp": "2018-02-01T17:21:05.000+08:00"
  },
  "source": null,
  "result": {
    "data": {
```

```
      "temperatureReadings": [
        null
      ],
      "locationReadings": [
        {
          "lat": 47.615063,
          "long": -122.333551,
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ]
    },
    "unprocessedKeys": {
      "temperatureReadings": [
        {
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ],
      "locationReadings": []
    }
  },
  "error": {
    "type": "DynamoDB:ProvisionedThroughputExceededException",
    "message": "You exceeded your maximum allowed provisioned throughput for a table or
 for one or more global secondary indexes. (...)"
  },
  "outErrors": []
}
```

Quelques points à noter concernant le contexte :

- l'erreur d'appel a été définie sur le contexte à `$ctx.error` by AWS AppSync, et le type d'erreur a été défini sur DynamoDB :. ProvisionedThroughputExceededException
- Les résultats sont mappés par table sous `$ctx.result.data`, même si une erreur est présente.
- Les clés qui n'ont pas été traitées sont disponibles à l'adresse `$ctx.result.data.unprocessedKeys`. Ici, AWS AppSync n’a pas pu récupérer l'élément avec la clé (sensorId:1, timestamp:2018-02-01T 17:21:05 .000+ 08:00) en raison du débit de table insuffisant.

Remarque : Pour `BatchPutItem`, la valeur est `$ctx.result.data.unprocessedItems`. Pour `BatchDeleteItem`, la valeur est `$ctx.result.data.unprocessedKeys`.

Nous allons traiter cette erreur de trois façons différentes.

1. Digestion de l'erreur d'appel

Le renvoi des données sans gestion de l'erreur d'appel se traduit par une digestion de l'erreur, ce qui permet au résultat du champ GraphQL donné d'être toujours réussi.

Le modèle de mappage de réponse que nous écrivons est courant et ne se concentre que sur les données de résultat.

Modèle de mappage de réponse :

```
$util.toJson($ctx.result.data)
```

Réponse GraphQL :

```
{
  "data": {
    "getReadings": [
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "lat": 47.615063,
        "long": -122.333551
      },
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  }
}
```

Aucune erreur n'est ajoutée à la réponse d'erreur car l'action n'a porté que sur les données.

2. Déclenchement d'une erreur pour interrompre l'exécution du modèle

Lorsque les défaillances partielles doivent être traitées comme des défaillances complètes du point de vue du client, vous pouvez abandonner l'exécution du modèle pour empêcher le renvoi

des données. La méthode d'utilitaire `$util.error(...)` permet d'obtenir exactement ce comportement.

Modèle de mappage de réponse :

```
## there was an error let's mark the entire field
## as failed and do not return any data back in the response
#if ($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type, null,
 $ctx.result.data.unprocessedKeys)
#end

$util.toJson($ctx.result.data)
```

Réponse GraphQL :

```
{
  "data": {
    "getReadings": null
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
        }
      ],
```

```
      "message": "You exceeded your maximum allowed provisioned throughput for a table
 or for one or more global secondary indexes. (...)"
    }
  ]
}
```

Même si certains résultats peuvent avoir été renvoyés par l'opération de traitement par lots DynamoDB, nous avons choisi de déclencher une erreur se traduisant par une valeur null pour le champ GraphQL `getReadings` et l'erreur a été ajoutée au bloc d'erreurs de la réponse GraphQL.

3. Ajout d'une erreur pour renvoyer à la fois les données et les erreurs

Dans certains cas, afin d'offrir une meilleure expérience utilisateur, les applications peuvent renvoyer des résultats partiels et informer leurs clients des éléments non traités. Les clients peuvent choisir d'implémenter une nouvelle tentative ou de renvoyer l'erreur à l'utilisateur final. C'est la méthode d'utilitaire `$util.appendError(...)` qui permet d'obtenir ce comportement en laissant le concepteur de l'application ajouter les erreurs au contexte sans interférer avec l'évaluation du modèle. Une fois le modèle évalué, AWS AppSync traite les erreurs de contexte en les ajoutant au bloc d'erreurs de la réponse GraphQL.

Modèle de mappage de réponse :

```
#if ($ctx.error)
    ## pass the unprocessed keys back to the caller via the `errorInfo` field
    $util.appendError($ctx.error.message, $ctx.error.type, null,
 $ctx.result.data.unprocessedKeys)
#end

$util.toJson($ctx.result.data)
```

Nous avons transmis à la fois l'erreur d'appel et l'élément unprocessedKeys dans le bloc d'erreurs de la réponse GraphQL. Le champ `getReadings` renvoie également les données partielles à partir de la table locationReadings comme vous pouvez le voir dans la réponse ci-dessous.

Réponse GraphQL :

```
{
  "data": {
    "getReadings": [
      null,
```

```
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
        }
      ],
      "message": "You exceeded your maximum allowed provisioned throughput for a table
 or for one or more global secondary indexes. (...)"
    }
  ]
}
```

Tutoriel : résolveurs de transactions DynamoDB

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

AWS AppSync prend en charge l'utilisation des opérations de transaction Amazon DynamoDB sur une ou plusieurs tables d'une même région. Les opérations prises en charge sont `TransactGetItems` et `TransactWriteItems`. L'utilisation de ces fonctions dans AWS AppSync vous permet d'exécuter des tâches telles que les suivantes :

- Transmission d'une liste de clés dans une seule requête et renvoi des résultats à partir d'une table
- Lecture des enregistrements à partir d'une ou plusieurs tables dans une seule requête
- Écrire les enregistrements d'une transaction sur une ou plusieurs tables d'une all-or-nothing manière ou d'une autre
- Exécuter des transactions lorsque certaines conditions sont remplies

Autorisations

Comme pour les autres résolveurs, vous devez créer une source de données dans AWS AppSync et créer un rôle ou utiliser un rôle existant. Les opérations de transaction nécessitant des autorisations différentes sur les tables DynamoDB, vous devez accorder au rôle configuré des autorisations pour les actions de lecture ou d'écriture :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:region:accountId:table/TABLENAME",
                "arn:aws:dynamodb:region:accountId:table/TABLENAME/*"
            ]
        }
    ]
}
```

Remarque : Les rôles sont liés aux sources de données dans AWS AppSync et les résolveurs sur les champs sont appelés par rapport à une source de données. Pour simplifier la configuration, une seule table est spécifiée pour les sources de données configurées pour effectuer une extraction par rapport à DynamoDB. Par conséquent, lorsque vous effectuez une opération de transaction sur plusieurs tables dans un seul résolveur (ce qui constitue une tâche plus avancée), vous devez accorder au rôle associé à cette source de données l'accès à toutes les tables avec lesquelles le résolveur devra interagir. Cela doit être effectué dans le champ Resource (Ressource) dans la stratégie IAM ci-dessus. La configuration des appels de transactions sur les tables doit être effectuée dans le modèle de résolveur, dont vous trouverez la description ci-dessous.

Source de données

Dans un souci de simplicité, nous allons utiliser la même source de données pour tous les résolveurs utilisés dans ce didacticiel. Dans l'onglet Sources de données, créez une nouvelle source de données DynamoDB et nommez-la. TransactTutorial Le nom de la table peut être quelconque, car les noms de table sont spécifiés dans le cadre du modèle de mappage de demande pour les opérations de transaction. Nous allons nommer la table `empty`.

Nous aurons deux tables appelées SavingAccounts et CheckingAccounts, les deux avec `accountNumber` comme clé de partition, et une table TransactionHistory avec `transactionId` comme clé de partition.

Dans le cadre de ce didacticiel, n'importe quel rôle avec la stratégie en ligne suivante fonctionne. Remplacez `region` et `accountId` par votre région et votre numéro de compte :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:region:accountId:table/savingAccounts",
                "arn:aws:dynamodb:region:accountId:table/savingAccounts/*",
```

```
                "arn:aws:dynamodb:region:accountId:table/checkingAccounts",
                "arn:aws:dynamodb:region:accountId:table/checkingAccounts/*",
                "arn:aws:dynamodb:region:accountId:table/transactionHistory",
                "arn:aws:dynamodb:region:accountId:table/transactionHistory/*"
            ]
        }
    ]
}
```

Transactions

Pour cet exemple, le contexte est une transaction bancaire classique, où nous allons utiliser `TransactWriteItems` pour :

- Transférer de l'argent des comptes d'épargne vers des comptes de contrôle
- Générer de nouveaux enregistrements de transaction pour chaque transaction

Ensuite, nous allons utiliser `TransactGetItems` pour récupérer les détails des comptes d'enregistrement et des comptes de vérification.

Nous définissons notre schéma GraphQL comme suit :

```
type SavingAccount {
    accountNumber: String!
    username: String
    balance: Float
}

type CheckingAccount {
    accountNumber: String!
    username: String
    balance: Float
}

type TransactionHistory {
    transactionId: ID!
    from: String
    to: String
    amount: Float
}

type TransactionResult {
```

```
    savingAccounts: [SavingAccount]
    checkingAccounts: [CheckingAccount]
    transactionHistory: [TransactionHistory]
}

input SavingAccountInput {
    accountNumber: String!
    username: String
    balance: Float
}

input CheckingAccountInput {
    accountNumber: String!
    username: String
    balance: Float
}

input TransactionInput {
    savingAccountNumber: String!
    checkingAccountNumber: String!
    amount: Float!
}

type Query {
    getAccounts(savingAccountNumbers: [String], checkingAccountNumbers: [String]):
 TransactionResult
}

type Mutation {
    populateAccounts(savingAccounts: [SavingAccountInput], checkingAccounts:
 [CheckingAccountInput]): TransactionResult
    transferMoney(transactions: [TransactionInput]): TransactionResult
}

schema {
    query: Query
    mutation: Mutation
}
```

TransactWriteItems - Remplir les comptes

Afin de transférer de l'argent entre les comptes, nous devons remplir la table avec les détails. Nous allons utiliser l'opération GraphQL `Mutation.populateAccounts` pour le faire.

Dans la section Schéma, cliquez sur Joindre à côté de l'`Mutation.populateAccounts`opération. Accédez à VTL Unit Resolvers, puis choisissez la même source de `TransactTutorial` données.

Maintenant, utilisez le modèle de mappage de demande suivant :

Modèle de mappage de demande

```
#set($savingAccountTransactPutItems = [])
#set($index = 0)
#foreach($savingAccount in ${ctx.args.savingAccounts})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
 $util.dynamodb.toString($savingAccount.accountNumber)))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("username",
 $util.dynamodb.toString($savingAccount.username)))
    $util.qr($attributeValues.put("balance",
 $util.dynamodb.toNumber($savingAccount.balance)))
    #set($index = $index + 1)
    #set($savingAccountTransactPutItem = {"table": "savingAccounts",
        "operation": "PutItem",
        "key": $keyMap,
        "attributeValues": $attributeValues})
    $util.qr($savingAccountTransactPutItems.add($savingAccountTransactPutItem))
#end

#set($checkingAccountTransactPutItems = [])
#set($index = 0)
#foreach($checkingAccount in ${ctx.args.checkingAccounts})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
 $util.dynamodb.toString($checkingAccount.accountNumber)))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("username",
 $util.dynamodb.toString($checkingAccount.username)))
    $util.qr($attributeValues.put("balance",
 $util.dynamodb.toNumber($checkingAccount.balance)))
    #set($index = $index + 1)
    #set($checkingAccountTransactPutItem = {"table": "checkingAccounts",
        "operation": "PutItem",
        "key": $keyMap,
        "attributeValues": $attributeValues})
    $util.qr($checkingAccountTransactPutItems.add($checkingAccountTransactPutItem))
#end
```

```
#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountTransactPutItems))
$util.qr($transactItems.addAll($checkingAccountTransactPutItems))

{
    "version" : "2018-05-29",
    "operation" : "TransactWriteItems",
    "transactItems" : $util.toJson($transactItems)
}
```

Selon le modèle de mappage de réponse suivant :

Modèle de mappage de réponse

```
#if ($ctx.error)
    $util.appendError($ctx.error.message, $ctx.error.type, null,
 $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
    $util.qr($savingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($checkingAccounts = [])
#foreach($index in [3..5])
    $util.qr($checkingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))

$util.toJson($transactionResult)
```

Enregistrez le résolveur et accédez à la section Requêtes de la console AWS AppSync pour remplir les comptes.

Exécutez la mutation suivante :

```
mutation populateAccounts {
  populateAccounts (
```

```
    savingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 100},
      {accountNumber: "2", username: "Amy", balance: 90},
      {accountNumber: "3", username: "Lily", balance: 80},
    ]
    checkingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 70},
      {accountNumber: "2", username: "Amy", balance: 60},
      {accountNumber: "3", username: "Lily", balance: 50},
    ]) {
    savingAccounts {
      accountNumber
    }
    checkingAccounts {
      accountNumber
    }
  }
}
```

Nous avons rempli 3 comptes d'épargne et 3 comptes de vérification en une seule mutation.

Utilisez la console DynamoDB pour vérifier que les données apparaissent à la fois dans les tables SavingAccounts et CheckingAccounts.

TransactWriteItems - Transférer de l'argent

Attachez un résolveur au champ `transferMoney` avec le modèle de mappage de demande suivant. Notez que les valeurs de `amounts`, `savingAccountNumbers` et `checkingAccountNumbers` sont les mêmes.

```
#set($amounts = [])
#foreach($transaction in ${ctx.args.transactions})
    #set($attributeValueMap = {})
    $util.qr($attributeValueMap.put(":amount",
 $util.dynamodb.toNumber($transaction.amount)))
    $util.qr($amounts.add($attributeValueMap))
#end

#set($savingAccountTransactUpdateItems = [])
#set($index = 0)
#foreach($transaction in ${ctx.args.transactions})
    #set($keyMap = {})
```

```
    $util.qr($keyMap.put("accountNumber",
 $util.dynamodb.toString($transaction.savingAccountNumber)))
    #set($update = {})
    $util.qr($update.put("expression", "SET balance = balance - :amount"))
    $util.qr($update.put("expressionValues", $amounts[$index]))
    #set($index = $index + 1)
    #set($savingAccountTransactUpdateItem = {"table": "savingAccounts",
        "operation": "UpdateItem",
        "key": $keyMap,
        "update": $update})
    $util.qr($savingAccountTransactUpdateItems.add($savingAccountTransactUpdateItem))
#end

#set($checkingAccountTransactUpdateItems = [])
#set($index = 0)
#foreach($transaction in ${ctx.args.transactions})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
 $util.dynamodb.toString($transaction.checkingAccountNumber)))
    #set($update = {})
    $util.qr($update.put("expression", "SET balance = balance + :amount"))
    $util.qr($update.put("expressionValues", $amounts[$index]))
    #set($index = $index + 1)
    #set($checkingAccountTransactUpdateItem = {"table": "checkingAccounts",
        "operation": "UpdateItem",
        "key": $keyMap,
        "update": $update})

 $util.qr($checkingAccountTransactUpdateItems.add($checkingAccountTransactUpdateItem))
#end

#set($transactionHistoryTransactPutItems = [])
#foreach($transaction in ${ctx.args.transactions})
    #set($keyMap = {})
    $util.qr($keyMap.put("transactionId", $util.dynamodb.toString(${utils.autoId()})))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("from",
 $util.dynamodb.toString($transaction.savingAccountNumber)))
    $util.qr($attributeValues.put("to",
 $util.dynamodb.toString($transaction.checkingAccountNumber)))
    $util.qr($attributeValues.put("amount",
 $util.dynamodb.toNumber($transaction.amount)))
    #set($transactionHistoryTransactPutItem = {"table": "transactionHistory",
        "operation": "PutItem",
```

```
        "key": $keyMap,
        "attributeValues": $attributeValues})

 $util.qr($transactionHistoryTransactPutItems.add($transactionHistoryTransactPutItem))
#end

#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountTransactUpdateItems))
$util.qr($transactItems.addAll($checkingAccountTransactUpdateItems))
$util.qr($transactItems.addAll($transactionHistoryTransactPutItems))

{
    "version" : "2018-05-29",
    "operation" : "TransactWriteItems",
    "transactItems" : $util.toJson($transactItems)
}
```

Nous aurons 3 transactions bancaires en une seule opération `TransactWriteItems`. Utilisez le modèle de mappage de réponse suivant :

```
#if ($ctx.error)
    $util.appendError($ctx.error.message, $ctx.error.type, null,
 $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
    $util.qr($savingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($checkingAccounts = [])
#foreach($index in [3..5])
    $util.qr($checkingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($transactionHistory = [])
#foreach($index in [6..8])
    $util.qr($transactionHistory.add(${ctx.result.keys[$index]}))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))
```

```
$util.qr($transactionResult.put('transactionHistory', $transactionHistory))

$util.toJson($transactionResult)
```

Accédez maintenant à la section Requêtes de la console AWS AppSync et exécutez la mutation TransferMoney comme suit :

```
mutation write {
  transferMoney(
    transactions: [
      {savingAccountNumber: "1", checkingAccountNumber: "1", amount: 7.5},
      {savingAccountNumber: "2", checkingAccountNumber: "2", amount: 6.0},
      {savingAccountNumber: "3", checkingAccountNumber: "3", amount: 3.3}
    ]) {
    savingAccounts {
      accountNumber
    }
    checkingAccounts {
      accountNumber
    }
    transactionHistory {
      transactionId
    }
  }
}
```

Nous avons envoyé 2 transactions bancaires en une seule mutation. Utilisez la console DynamoDB pour vérifier que les données apparaissent dans les tables SavingAccounts, CheckingAccounts et TransactionHistory.

TransactGetItems - Récupérer des comptes

Afin de récupérer les détails des comptes d'enregistrement et de vérification dans une seule requête transactionnelle, nous attacherons un résolveur à l'opération `Query.getAccounts` GraphQL sur notre schéma. Sélectionnez Joindre, accédez à VTL Unit Resolvers, puis sur l'écran suivant, sélectionnez la même source de `TransactTutorial` données créée au début du didacticiel. Configurez les modèles comme suit :

Modèle de mappage de demande

```
#set($savingAccountsTransactGets = [])
```

```
#foreach($savingAccountNumber in ${ctx.args.savingAccountNumbers})
    #set($savingAccountKey = {})
    $util.qr($savingAccountKey.put("accountNumber",
 $util.dynamodb.toString($savingAccountNumber)))
    #set($savingAccountTransactGet = {"table": "savingAccounts", "key":
 $savingAccountKey})
    $util.qr($savingAccountsTransactGets.add($savingAccountTransactGet))
#end

#set($checkingAccountsTransactGets = [])
#foreach($checkingAccountNumber in ${ctx.args.checkingAccountNumbers})
    #set($checkingAccountKey = {})
    $util.qr($checkingAccountKey.put("accountNumber",
 $util.dynamodb.toString($checkingAccountNumber)))
    #set($checkingAccountTransactGet = {"table": "checkingAccounts", "key":
 $checkingAccountKey})
    $util.qr($checkingAccountsTransactGets.add($checkingAccountTransactGet))
#end

#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountsTransactGets))
$util.qr($transactItems.addAll($checkingAccountsTransactGets))

{
    "version" : "2018-05-29",
    "operation" : "TransactGetItems",
    "transactItems" : $util.toJson($transactItems)
}
```

Modèle de mappage de réponse

```
#if ($ctx.error)
    $util.appendError($ctx.error.message, $ctx.error.type, null,
 $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
    $util.qr($savingAccounts.add(${ctx.result.items[$index]}))
#end

#set($checkingAccounts = [])
#foreach($index in [3..4])
```

```
    $util.qr($checkingAccounts.add($ctx.result.items[$index]))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))

$util.toJson($transactionResult)
```

Enregistrez le résolveur et accédez aux sections Requêtes de la console AWS AppSync . Pour récupérer les comptes d'enregistrement et de vérification, exécutez la requête suivante :

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
      username
      balance
    }
  }
}
```

Nous avons démontré avec succès l'utilisation des transactions DynamoDB en utilisant. AWS AppSync

Tutoriel : résolveurs HTTP

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

AWS AppSync vous permet d'utiliser des sources de données prises en charge (Amazon DynamoDBAWS Lambda, Amazon Service ou OpenSearch Amazon Aurora) pour effectuer diverses opérations, en plus des points de terminaison HTTP arbitraires pour résoudre les champs GraphQL. Dès que vos points de terminaison HTTP sont disponibles, vous pouvez vous y connecter à l'aide d'une source de données. Ensuite, vous pouvez configurer un résolveur dans le schéma GraphQL pour effectuer des opérations telles que des requêtes, des mutations et des abonnements. Ce didacticiel vous présente certains exemples courants.

Dans ce didacticiel, vous utiliserez une API REST (créée à l'aide d'Amazon API Gateway et Lambda) avec un point de terminaison GraphQL AWS AppSync .

Configuration en un clic

Si vous souhaitez configurer automatiquement un point de terminaison GraphQL AWS AppSync avec un point de terminaison HTTP configuré (à l'aide d'Amazon API Gateway et Lambda), vous pouvez utiliser le modèle suivant : AWS CloudFormation

Création d'une API REST

Vous pouvez utiliser le modèle AWS CloudFormation suivant pour configurer un point de terminaison REST qui fonctionne pour ce didacticiel :

La pile AWS CloudFormation exécute les étapes suivantes :

1. Elle configure une fonction Lambda qui contient la logique métier de votre microservice.
2. Configure une API REST API Gateway avec la combinaison point de terminaison, méthode et type de contenu suivante :

Chemin de ressource API	Méthode HTTP	Type de contenu pris en charge
/v1/users	POST	application/json

Chemin de ressource API	Méthode HTTP	Type de contenu pris en charge
/v1/users	GET	application/json
/v1/users/1	GET	application/json
/v1/users/1	PUT	application/json
/v1/users/1	DELETE	application/json

Création de votre API GraphQL

Pour créer l'API GraphQL dans AWS AppSync :

- Ouvrez la console AWS AppSync et choisissez Créer une API.
- Pour le nom de l'API, saisissez `UserData`.
- Choisissez Schéma personnalisé.
- Sélectionnez Create (Créer).

La console AWS AppSync crée une nouvelle API GraphQL pour vous à l'aide du mode d'authentification de clé API. Vous pouvez utiliser la console pour configurer le reste de l'API GraphQL et exécuter des requêtes sur celle-ci jusqu'à la fin de ce didacticiel.

Création d'un schéma GraphQL

Maintenant que vous avez une API GraphQL, nous allons créer un schéma GraphQL. Dans l'éditeur de schéma de la console AWS AppSync , assurez-vous que votre schéma correspond au schéma ci-dessous :

```
schema {
    query: Query
    mutation: Mutation
}

type Mutation {
    addUser(userInput: UserInput!): User
    deleteUser(id: ID!): User
```

```
}

type Query {
    getUser(id: ID): User
    listUser: [User!]!
}

type User {
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
}

input UserInput {
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
}
```

Configurez votre source de données HTTP

Pour configurer votre source de données HTTP, procédez comme suit :

- DataSourcesDans l'onglet, choisissez Nouveau, puis tapez un nom convivial pour la source de données (par exemple,HTTP).
- Dans Type de source de données, choisissez HTTP.
- Définissez le point de terminaison sur le point de terminaison API Gateway créé. Assurez-vous de ne pas inclure le nom de l'étape dans le point de terminaison.

Remarque : à l'heure actuelle, seuls les points de terminaison publics sont pris en charge par AWS AppSync.

Remarque : Pour plus d'informations sur les autorités de certification reconnues par le AWS AppSync service, voir Autorités de certification (CA) reconnues par AWS AppSync pour les points de terminaison HTTPS.

Configuration des résolveurs

Au cours de cette étape, vous allez connecter la source de données http à la requête getUser.

Pour configurer le résolveur :

- Choisissez l'onglet Schéma.
- Dans le volet Types de données à droite sous le champ Requête, recherchez le champ getUser et choisissez Joindre.
- Dans Nom de la source de données, choisissez HTTP.
- Collez le code suivant dans la section Configurer le modèle de mappage de demande :

```
{
    "version": "2018-05-29",
    "method": "GET",
    "params": {
        "headers": {
            "Content-Type": "application/json"
        }
    },
    "resourcePath": $util.toJson("/v1/users/${ctx.args.id}")
}
```

- Collez le code suivant dans la section Configurer le modèle de mappage de réponse :

```
## return the body
#if($ctx.result.statusCode == 200)
    ##if response is 200
    $ctx.result.body
#else
    ##if response is not 200, append the response to error block.
    $utils.appendError($ctx.result.body, "$ctx.result.statusCode")
#end
```

- Choisissez l'onglet Requête et exécutez la requête suivante :

```
query GetUser{
    getUser(id:1){
```

```
        id
        username
    }
}
```

Cela doit renvoyer la réponse suivante :

```
{
    "data": {
        "getUser": {
            "id": "1",
            "username": "nadia"
        }
    }
}
```

- Choisissez l'onglet Schéma.
- Dans le volet Types de données à droite sous le champ Mutation, recherchez le champ addUser et choisissez Joindre.
- Dans Nom de la source de données, choisissez HTTP.
- Collez le code suivant dans la section Configurer le modèle de mappage de demande :

```
{
    "version": "2018-05-29",
    "method": "POST",
    "resourcePath": "/v1/users",
    "params":{
      "headers":{
        "Content-Type": "application/json",
      },
      "body": $util.toJson($ctx.args.userInput)
    }
}
```

- Collez le code suivant dans la section Configurer le modèle de mappage de réponse :

```
## Raise a GraphQL field error in case of a datasource invocation error
#if($ctx.error)
```

```
    $util.error($ctx.error.message, $ctx.error.type)
#end
## if the response status code is not 200, then return an error. Else return the body
 **
#if($ctx.result.statusCode == 200)
    ## If response is 200, return the body.
    $ctx.result.body
#else
    ## If response is not 200, append the response to error block.
    $utils.appendError($ctx.result.body, "$ctx.result.statusCode")
#end
```

- Choisissez l'onglet Requête et exécutez la requête suivante :

```
mutation addUser{
    addUser(userInput:{
        id:"2",
        username:"shaggy"
    }){
        id
        username
    }
}
```

Cela doit renvoyer la réponse suivante :

```
{
    "data": {
        "getUser": {
        "id": "2",
        "username": "shaggy"
        }
    }
}
```

Invoquer AWS des services

Vous pouvez utiliser des résolveurs HTTP pour configurer une interface AWS d'API GraphQL pour les services. Les requêtes HTTP AWS doivent être signées à l'aide du processus Signature Version

4 afin de AWS pouvoir identifier leur expéditeur. AWS AppSync calcule la signature en votre nom lorsque vous associez un rôle IAM à la source de données HTTP.

Vous fournissez deux composants supplémentaires pour appeler AWS des services avec des résolveurs HTTP :

- Un rôle IAM autorisé à appeler les API du AWS service
- La configuration de signature dans la source de données

Par exemple, si vous souhaitez appeler l'ListGraphqlApis opération avec des résolveurs HTTP, vous devez d'abord créer un rôle IAM doté AWS AppSync de la politique suivante :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "appsync:ListGraphqlApis"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

Créez ensuite la source de données HTTP pour AWS AppSync. Dans cet exemple, vous appelez AWS AppSync dans la région USA Ouest (Oregon). Configurez la configuration HTTP suivante dans un fichier nommé `http.json`, qui inclut la région de signature et le nom du service :

```
{
    "endpoint": "https://appsync.us-west-2.amazonaws.com/",
    "authorizationConfig": {
        "authorizationType": "AWS_IAM",
        "awsIamConfig": {
            "signingRegion": "us-west-2",
            "signingServiceName": "appsync"
        }
    }
}
```

Utilisez ensuite le AWS CLI pour créer la source de données avec un rôle associé, comme suit :

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

Lorsque vous attachez un résolveur au champ dans le schéma, utilisez le modèle de mappage de demande suivant pour appeler AWS AppSync :

```
{
    "version": "2018-05-29",
    "method": "GET",
    "resourcePath": "/v1/apis"
}
```

Lorsque vous exécutez une requête GraphQL pour cette source de données, AWS AppSync signe la demande à l'aide du rôle que vous avez fourni et inclut la signature dans la demande. La requête renvoie une liste des API AWS AppSync GraphQL présentes dans votre compte dans cette AWS région.

Tutoriel : Aurora Serverless

AWS AppSyncfournit une source de données pour exécuter des commandes SQL sur des clusters Amazon Aurora Serverless qui ont été activés avec une API de données. Vous pouvez utiliser des AppSync résolveurs pour exécuter des instructions SQL sur l'API de données à l'aide de requêtes GraphQL, de mutations et d'abonnements.

Créer un cluster

Avant d'ajouter une source de données RDS, AppSync vous devez d'abord activer une API de données sur un cluster Aurora Serverless et configurer un secret à l'aide de. AWS Secrets Manager Vous pouvez d'abord créer un cluster Aurora Serverless avec AWS CLI :

```
aws rds create-db-cluster --db-cluster-identifier http-endpoint-test  --master-username
 USERNAME \
--master-user-password COMPLEX_PASSWORD --engine aurora --engine-mode serverless \
--region us-east-1
```

Cela renverra un ARN pour le cluster.

Créez un secret via la AWS Secrets Manager console ou également via la CLI avec un fichier d'entrée tel que le suivant en utilisant le NOM D'UTILISATEUR et le COMPLEX_PASSWORD de l'étape précédente :

```
{
    "username": "USERNAME",
    "password": "COMPLEX_PASSWORD"
}
```

Passez ceci en tant que paramètre à AWS CLI :

```
aws secretsmanager create-secret --name HttpRDSSecret --secret-string file://creds.json
 --region us-east-1
```

Cela renverra un ARN pour le secret.

Notez l'ARN de votre cluster Aurora Serverless et le code secret pour une utilisation ultérieure dans la AppSync console lors de la création d'une source de données.

Activer l'API de données

Vous pouvez activer l'API de données sur votre cluster en suivant les instructions fournies dans la documentation RDS. L'API de données doit être activée avant d'être ajoutée en tant que source de AppSync données.

Créer une base de données et une table

Une fois que vous avez activé votre API de données, vous pouvez vous assurer qu'elle fonctionne à l'aide de la `aws rds-data execute-statement` commande duAWS CLI. Cela garantira que votre cluster Aurora Serverless est correctement configuré avant de l'ajouter à votre AppSync API. Créez d'abord une base de données nommée TESTDB avec le paramètre `--sql` comme suit :

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789000:cluster:http-endpoint-test" \
--schema "mysql"  --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789000:secret:testHttp2-AmNvc1"  \
--region us-east-1 --sql "create DATABASE TESTDB"
```

Si elle s'exécute sans erreur, ajoutez une table avec la commande create table (créer une table) :

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789000:cluster:http-endpoint-test" \
 --schema "mysql"  --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789000:secret:testHttp2-AmNvc1" \
 --region us-east-1 \
 --sql "create table Pets(id varchar(200), type varchar(200), price float)" --database
 "TESTDB"
```

Si tout s'est déroulé sans problème, vous pouvez passer à l'ajout du cluster en tant que source de données dans votre AppSync API.

Schéma GraphQL

Maintenant que votre API de données Aurora sans serveur est opérationnel avec une table, nous allons créer un schéma GraphQL et attacher des résolveurs pour réaliser des mutations et des abonnements. Créez une nouvelle API dans la AWS AppSync console, accédez à la page Schéma, puis entrez les informations suivantes :

```
type Mutation {
    createPet(input: CreatePetInput!): Pet
    updatePet(input: UpdatePetInput!): Pet
    deletePet(input: DeletePetInput!): Pet
}

input CreatePetInput {
    type: PetType
    price: Float!
}

input UpdatePetInput {
id: ID!
    type: PetType
    price: Float!
}

input DeletePetInput {
    id: ID!
}

type Pet {
    id: ID!
    type: PetType
```

```
    price: Float
}

enum PetType {
    dog
    cat
    fish
    bird
    gecko
}

type Query {
    getPet(id: ID!): Pet
    listPets: [Pet]
    listPetsByPriceRange(min: Float, max: Float): [Pet]
}

schema {
    query: Query
    mutation: Mutation
}
```

Enregistrez votre schéma et accédez à la page Data Sources (Sources de données), puis créez une nouvelle source de données. Sélectionnez une Base de données relationnelle pour le type de source de données et saisissez un nom convivial. Utilisez le nom de la base de données que vous avez créé au cours de l'étape précédente, ainsi que l'ARN de cluster que vous avez créé. Pour le rôle, vous pouvez soit AppSync créer un nouveau rôle, soit en créer un avec une politique similaire à celle ci-dessous :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "rds-data:DeleteItems",
                "rds-data:ExecuteSql",
                "rds-data:ExecuteStatement",
                "rds-data:GetItems",
                "rds-data:InsertItems",
                "rds-data:UpdateItems"
            ],
```

```
            "Resource": [
                "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster",
                "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster:*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": [
                "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret",
                "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret:*"
            ]
        }
    ]
}
```

Notez qu'il y a deux Déclarations dans cette stratégie auxquelles vous accordez l'accès au rôle. La première ressource est votre cluster Aurora Serverless et la seconde est votre AWS Secrets Manager ARN. Vous devrez fournir les DEUX ARN dans la configuration de la source de AppSync données avant de cliquer sur Créer.

Configuration des résolveurs

Maintenant que nous avons un schéma GraphQL et une source de données RDS valides, nous pouvons attacher des résolveurs aux champs GraphQL sur notre schéma. Notre API proposera les fonctions suivantes :

1. créer un animal de compagnie via le champ Mutation.createPet
2. mettre à jour un animal de compagnie via le champ Mutation.updatePet
3. supprimer un animal de compagnie via le champ Mutation.deletePet
4. obtenir un animal de compagnie unique via le champ Query.getPet
5. répertorier tous les animaux de compagnie via le champ Query.listPets
6. listez les animaux de compagnie dans une fourchette de prix via la requête. listPetsByPriceRangechamp

Mutation.createPet

Dans l'éditeur de schéma de la console AWS AppSync , sur le côté droit, choisissez Joindre un résolveur pour `createPet(input: CreatePetInput!): Pet`. Choisissez votre source de données RDS. Dans la section request mapping template (modèle de mappage de requête), ajoutez le modèle suivant :

```
#set($id=$utils.autoId())
{
"version": "2018-05-29",
    "statements": [
        "insert into Pets VALUES (:ID, :TYPE, :PRICE)",
        "select * from Pets WHERE id = :ID"
    ],
    "variableMap": {
        ":ID": "$ctx.args.input.id",
        ":TYPE": $util.toJson($ctx.args.input.type),
        ":PRICE": $util.toJson($ctx.args.input.price)
    }
}
```

Les déclarations SQL s'exécuteront de façon séquentielle, selon leur ordre dans l'ensemble de déclarations. Les résultats reviendront dans le même ordre. Puisqu'il s'agit d'une mutation, on exécute une déclaration select après insert pour extraire ces valeurs validées afin de remplir le modèle de mappage de réponse GraphQL.

Dans la section response mapping template (modèle de mappage de réponse), ajoutez le modèle suivant :

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

Comme les déclarations possèdent deux requêtes SQL, nous devons spécifier le deuxième résultat dans la matrice qui revient de la base de données avec : `$utils.rds.toJsonString($ctx.result))[1][0]`).

Mutation.updatePet

Dans l'éditeur de schéma de la console AWS AppSync , sur le côté droit, choisissez Joindre un résolveur pour `updatePet(input: UpdatePetInput!): Pet`. Choisissez votre source de données RDS. Dans la section request mapping template (modèle de mappage de requête), ajoutez le modèle suivant :

```
{
"version": "2018-05-29",
    "statements": [
        $util.toJson("update Pets set type=:TYPE, price=:PRICE WHERE id=:ID"),
        $util.toJson("select * from Pets WHERE id = :ID")
    ],
    "variableMap": {
        ":ID": "$ctx.args.input.id",
        ":TYPE": $util.toJson($ctx.args.input.type),
        ":PRICE": $util.toJson($ctx.args.input.price)
    }
}
```

Dans la section response mapping template (modèle de mappage de réponse), ajoutez le modèle suivant :

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

Mutation.deletePet

Dans l'éditeur de schéma de la console AWS AppSync , sur le côté droit, choisissez Joindre un résolveur pour `deletePet(input: DeletePetInput!): Pet`. Choisissez votre source de données RDS. Dans la section request mapping template (modèle de mappage de requête), ajoutez le modèle suivant :

```
{
"version": "2018-05-29",
    "statements": [
        $util.toJson("select * from Pets WHERE id=:ID"),
        $util.toJson("delete from Pets WHERE id=:ID")
    ],
    "variableMap": {
        ":ID": "$ctx.args.input.id"
    }
}
```

Dans la section response mapping template (modèle de mappage de réponse), ajoutez le modèle suivant :

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.getPet

Maintenant que les mutations sont créées pour votre schéma, nous allons connecter les trois requêtes pour montrer comment obtenir des éléments individuels et des listes, et appliquer le filtrage SQL. Dans l'éditeur de schéma de la console AWS AppSync , sur le côté droit, choisissez Joindre un résolveur pour `getPet(id: ID!): Pet`. Choisissez votre source de données RDS. Dans la section request mapping template (modèle de mappage de requête), ajoutez le modèle suivant :

```
{
"version": "2018-05-29",
        "statements": [
            $util.toJson("select * from Pets WHERE id=:ID")
    ],
    "variableMap": {
        ":ID": "$ctx.args.id"
    }
}
```

Dans la section response mapping template (modèle de mappage de réponse), ajoutez le modèle suivant :

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.listPets

Dans l'éditeur de schéma de la console AWS AppSync , sur le côté droit, choisissez Joindre un résolveur pour `getPet(id: ID!): Pet`. Choisissez votre source de données RDS. Dans la section request mapping template (modèle de mappage de requête), ajoutez le modèle suivant :

```
{
    "version": "2018-05-29",
    "statements": [
        "select * from Pets"
    ]
}
```

Dans la section response mapping template (modèle de mappage de réponse), ajoutez le modèle suivant :

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

Requête. listPetsByPriceRange

Dans l'éditeur de schéma de la console AWS AppSync , sur le côté droit, choisissez Joindre un résolveur pour `getPet(id: ID!): Pet`. Choisissez votre source de données RDS. Dans la section request mapping template (modèle de mappage de requête), ajoutez le modèle suivant :

```
{
    "version": "2018-05-29",
    "statements": [
            "select * from Pets where price > :MIN and price < :MAX"
    ],

    "variableMap": {
        ":MAX": $util.toJson($ctx.args.max),
        ":MIN": $util.toJson($ctx.args.min)
    }
}
```

Dans la section response mapping template (modèle de mappage de réponse), ajoutez le modèle suivant :

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

Exécuter des mutations

Maintenant que vous avez configuré tous vos résolveurs avec les déclarations SQL et connecté votre API GraphQL à votre API de données Aurora sans serveur, vous pouvez commencer l'exécution de mutations et de requêtes. Dans la console AWS AppSync , choisissez l'onglet Queries (Requêtes) et saisissez la commande suivante pour créer un animal de compagnie :

```
mutation add {
    createPet(input : { type:fish, price:10.0 }){
        id
        type
        price
    }
}
```

La réponse doit contenir les id, type, et prix comme suit :

```
{
```

```
  "data": {
    "createPet": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "type": "fish",
      "price": "10.0"
    }
  }
}
```

Vous pouvez modifier cet élément en exécutant la mutation updatePet :

```
mutation update {
    updatePet(input : {
        id: ID_PLACEHOLDER,
        type:bird,
        price:50.0
    }){
        id
        type
        price
    }
}
```

Notez que nous avons utilisé l'id qui a été précédemment renvoyé de l'opération createPet. Il s'agira d'une valeur unique pour votre enregistrement car le résolveur a exploité `$util.autoId()`. Vous pouvez supprimer un enregistrement de cette façon :

```
mutation delete {
    deletePet(input : {id:ID_PLACEHOLDER}){
        id
        type
        price
    }
}
```

Créez quelques enregistrements avec la première mutation avec des valeurs différentes pour le prix,, puis exécutez quelques requêtes.

Exécuter des requêtes

Toujours dans l'onglet Queries (Requêtes) de la console, utilisez la déclaration suivante pour répertorier tous les enregistrements que vous avez créés :

```
query allpets {
    listPets {
        id
        type
        price
    }
}
```

C'est bien, mais tirons parti du prédicat SQL WHERE contenu `where price > :MIN and price < :MAX` dans notre modèle de mappage pour Query. listPetsByPriceRangeavec la requête GraphQL suivante :

```
query petsByPriceRange {
    listPetsByPriceRange(min:1, max:11) {
        id
        type
        price
    }
}
```

Vous devez uniquement voir des enregistrements avec un prix supérieur à $1 ou inférieur à $10. Enfin, vous pouvez effectuer des requêtes pour récupérer des enregistrements spécifiques, comme suit :

```
query onePet {
    getPet(id:ID_PLACEHOLDER){
        id
        type
        price
    }
}
```

Nettoyage des entrées

Nous recommandons aux développeurs de l'utiliser `variableMap` pour se protéger contre les attaques par injection de code SQL. Si aucune carte variable n'est utilisée, les développeurs sont chargés de nettoyer les arguments de leurs opérations GraphQL. L'une des façons de s'y prendre est de fournir des étapes de validation spécifique d'entrée dans le modèle de mappage de demande avant l'exécution d'une déclaration SQL sur votre API de données. Voyons comment modifier le

modèle de mappage de demande de l'exemple `listPetsByPriceRange`. Au lieu de vous baser uniquement sur l'entrée utilisateur, vous pouvez effectuer les actions suivantes :

```
#set($validMaxPrice = $util.matches("\d{1,3}[,\\.]?(\\d{1,2})?",$ctx.args.maxPrice))

#set($validMinPrice = $util.matches("\d{1,3}[,\\.]?(\\d{1,2})?",$ctx.args.minPrice))


#if (!$validMaxPrice || !$validMinPrice)
    $util.error("Provided price input is not valid.")
#end
{
    "version": "2018-05-29",
    "statements": [
            "select * from Pets where price > :MIN and price < :MAX"
    ],

    "variableMap": {
        ":MAX": $util.toJson($ctx.args.maxPrice),
        ":MIN": $util.toJson($ctx.args.minPrice)
    }
}
```

Une autre façon de vous protéger contre les entrées intruses lors de l'exécution de résolveurs sur vos API de données consiste à utiliser des instructions préparées avec une procédure stockée et des entrées paramétrées. Par exemple, dans le résolveur pour `listPets` définissez la procédure suivante qui exécute select comme une instruction préparée :

```
CREATE PROCEDURE listPets (IN type_param VARCHAR(200))
  BEGIN
     PREPARE stmt FROM 'SELECT * FROM Pets where type=?';
     SET @type = type_param;
     EXECUTE stmt USING @type;
     DEALLOCATE PREPARE stmt;
  END
```

Cela peut être créé dans votre instance Aurora sans serveur à l'aide de la commande execute sql suivante :

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:xxxxxxxxxxxx:cluster:http-endpoint-test" \
```

```
--schema "mysql"  --secret-arn "arn:aws:secretsmanager:us-
east-1:xxxxxxxxxxxx:secret:httpendpoint-xxxxxx"  \
--region us-east-1  --database "DB_NAME" \
--sql "CREATE PROCEDURE listPets (IN type_param VARCHAR(200)) BEGIN PREPARE stmt FROM
 'SELECT * FROM Pets where type=?'; SET @type = type_param; EXECUTE stmt USING @type;
 DEALLOCATE PREPARE stmt; END"
```

Le code résolveur obtenu pour listPets est simplifié, car désormais nous appelons simplement la procédure stockée. Au minimum, toute entrée de chaîne doit avoir des guillemets simples précédés d'un caractère d'échappement.

```
#set ($validType = $util.isString($ctx.args.type) && !
$util.isNullOrBlank($ctx.args.type))
#if (!$validType)
    $util.error("Input for 'type' is not valid.", "ValidationError")
#end

{
    "version": "2018-05-29",
    "statements": [
        "CALL listPets(:type)"
    ]
    "variableMap": {
        ":type": $util.toJson($ctx.args.type.replace("'", "''"))
    }
}
```

Échappement des chaînes

Les guillemets simples représentent le début et la fin des littéraux de chaîne dans une instruction SQL : par exemple, `'some string value'`. Pour permettre l'utilisation de valeurs de chaîne avec un ou plusieurs guillemets simples (`'`) dans une chaîne, chaque valeur doit être remplacée par deux guillemets simples (`''`). Par exemple, si la chaîne d'entrée est `Nadia's dog`, vous l'échappez pour l'instruction SQL comme

```
update Pets set type='Nadia''s dog' WHERE id='1'
```

Tutoriel : Pipeline Resolvers

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

AWS AppSync fournit un moyen simple de connecter un champ GraphQL à une source de données unique via des résolveurs unitaires. Toutefois, l'exécution d'une seule opération peut ne pas être suffisante. Les résolveurs de pipeline offrent la possibilité d'exécuter des opérations en série sur des sources de données. Créez des fonctions dans votre API et attachez-les à un résolveur de pipeline. L'exécution de chaque fonction est acheminée jusqu'à l'autre, jusqu'à ce qu'il ne reste plus aucune fonction à exécuter. Avec les résolveurs de pipeline, vous pouvez désormais créer des flux de travail plus complexes directement dans AWS AppSync. Dans ce didacticiel, vous construisez une application simple d'affichage de photos, dans laquelle les utilisateurs peuvent publier et visualiser les images publiées par leurs amis.

Configuration en un clic

Si vous souhaitez configurer automatiquement le point de terminaison GraphQL AWS AppSync avec tous les résolveurs configurés et les AWS ressources nécessaires, vous pouvez utiliser le modèle suivant : AWS CloudFormation

Cette pile crée les ressources suivantes dans votre compte :

- Rôle IAM pour AWS AppSync pour accéder aux ressources de votre compte
- 2 tables DynamoDB
- 1 groupe d'utilisateurs Amazon Cognito
- 2 groupes d'utilisateurs Amazon Cognito
- 3 utilisateurs de groupes d'utilisateurs Amazon Cognito
- 1 API AWS AppSync

À la fin du processus de création de la AWS CloudFormation pile, vous recevez un e-mail pour chacun des trois utilisateurs Amazon Cognito créés. Chaque e-mail contient un mot de passe temporaire à utiliser pour vous connecter en tant qu'utilisateur Amazon Cognito à la console AWS AppSync . Enregistrez les mots de passe pour le rappel du didacticiel.

Configuration manuelle

Si vous préférez exécuter manuellement un step-by-step processus via la AWS AppSync console, suivez le processus de configuration ci-dessous.

Configuration de vos AWS AppSync ressources non liées

L'API communique avec deux tables DynamoDB : une table d'images qui stocke des images et une table d'amis qui stocke les relations entre les utilisateurs. L'API est configurée pour utiliser le groupe d'utilisateurs Amazon Cognito en tant que type d'authentification. La AWS CloudFormation pile suivante configure ces ressources dans le compte.

À la fin du processus de création de la AWS CloudFormation pile, vous recevez un e-mail pour chacun des trois utilisateurs Amazon Cognito créés. Chaque e-mail contient un mot de passe temporaire à utiliser pour vous connecter en tant qu'utilisateur Amazon Cognito à la console AWS AppSync. Enregistrez les mots de passe pour le rappel du didacticiel.

Création de votre API GraphQL

Pour créer l'API GraphQL dans AWS AppSync :

1. Ouvrez la console AWS AppSync et choisissez Build From Scratch (Créer à partir de zéro), puis Start (Démarrer).
2. Définissez le nom de l'API en spécifiant `AppSyncTutorial-PicturesViewer`.
3. Sélectionnez Create (Créer).

La console AWS AppSync crée une nouvelle API GraphQL pour vous à l'aide du mode d'authentification de clé API. Vous pouvez utiliser la console pour configurer le reste de l'API GraphQL et exécuter des requêtes sur celle-ci jusqu'à la fin de ce didacticiel.

Configuration de l'API GraphQL

Vous devez configurer l'API AWS AppSync avec le groupe d'utilisateurs Amazon Cognito que vous venez de créer.

1. Sélectionnez l'onglet Settings.
2. Dans la section Authorization Type (Type d'autorisation), choisissez Amazon Cognito User Pool (Groupe d'utilisateurs Amazon Cognito).
3. Sous Configuration du groupe d'utilisateurs, choisissez US-WEST-2 pour la région. AWS
4. Choisissez le groupe UserPool d'utilisateurs AppSyncTutorial-.
5. Choisissez DENY (REFUSER) en tant que Default Action (Action par défaut).
6. Laissez le champ Regex du AppId client vide.
7. Choisissez Save (Enregistrer).

L'API est maintenant configurée pour utiliser le groupe d'utilisateurs Amazon Cognito en tant que type d'autorisation.

Configuration des sources de données pour les tables DynamoDB

Une fois les tables DynamoDB créées, accédez à votre API AWS AppSync GraphQL dans la console et choisissez l'onglet Sources de données. Vous allez maintenant créer une source de données AWS AppSync pour chacune des tables DynamoDB que vous venez de créer.

1. Choisissez l'onglet Source de données.
2. Choisissez New (Nouveau) pour créer une nouvelle source de données.
3. Pour le nom de la source de données, saisissez `PicturesDynamoDBTable`.
4. Pour le type de source de données, choisissez Table Amazon DynamoDB.
5. Pour la région, choisissez US-WEST-2.
6. Dans la liste des tables, choisissez la table AppSyncTutorial-Pictures DynamoDB.
7. Dans la section Créer ou utiliser un rôle existant, choisissez Rôle existant.
8. Choisissez le rôle qui vient d'être créé à partir du CloudFormation modèle. Si vous n'avez pas modifié le ResourceNamePrefix, le nom du rôle doit être AppSyncTutorial-DynaModbRole.
9. Sélectionnez Create (Créer).

Répétez le même processus pour la table des amis. Le nom de la table DynamoDB doit AppSyncTutorialêtre -Friends si vous n'avez pas modifié ResourceNamePrefixle paramètre au moment de créer la pile. CloudFormation

Création du schéma GraphQL

Maintenant que les sources de données sont connectées à vos tables DynamoDB, créons un schéma GraphQL. Dans l'éditeur de schéma de la console AWS AppSync , assurez-vous que votre schéma correspond au schéma ci-dessous :

```
schema {
    query: Query
    mutation: Mutation
}

type Mutation {
    createPicture(input: CreatePictureInput!): Picture!
    @aws_auth(cognito_groups: ["Admins"])
    createFriendship(id: ID!, target: ID!): Boolean
    @aws_auth(cognito_groups: ["Admins"])
}

type Query {
    getPicturesByOwner(id: ID!): [Picture]
    @aws_auth(cognito_groups: ["Admins", "Viewers"])
}

type Picture {
    id: ID!
    owner: ID!
    src: String
}

input CreatePictureInput {
    owner: ID!
    src: String!
}
```

Choisissez Save Schema (Enregistrer le schéma) pour enregistrer votre schéma.

Certains des champs de schéma ont été annotés avec la directive @aws_auth. La configuration d'action par défaut de l'API étant définie sur DENY (REFUSER), l'API rejette tous les utilisateurs

qui ne sont pas membres des groupes mentionnés dans la directive @aws_auth. Pour plus d'informations sur la façon de sécuriser votre API, vous pouvez lire la page Security (Sécurité). Dans ce cas, seuls les utilisateurs administrateurs ont accès aux champs Mutation.CreatePicture et Mutation.CreateFriendship, tandis que les utilisateurs membres des groupes Admins ou Viewers peuvent accéder à la requête. getPicturesByChamp du propriétaire. Tous les autres utilisateurs n'ont pas accès.

Configuration des résolveurs

Maintenant que vous avez un schéma GraphQL valide et deux sources de données, vous pouvez attacher des résolveurs aux champs GraphQL sur le schéma. L'&API propose les fonctions suivantes :

- Créer une image via le champ Mutation.createPicture
- Créer une relation via le champ Mutation.createFriendship
- Extraire une image via le champ Query.getPicture

Mutation.createPicture

Dans l'éditeur de schéma de la console AWS AppSync , sur le côté droit, choisissez Joindre un résolveur pour `createPicture(input: CreatePictureInput!): Picture!`. Choisissez la source de données PicturesDynamoDynamoDB DBTable. Dans la section request mapping template (modèle de mappage de requête), ajoutez le modèle suivant :

```
#set($id = $util.autoId())

{
    "version" : "2018-05-29",

    "operation" : "PutItem",

    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($id),
        "owner": $util.dynamodb.toDynamoDBJson($ctx.args.input.owner)
    },

    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args.input)
}
```

Dans la section response mapping template (modèle de mappage de réponse), ajoutez le modèle suivant :

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

La fonctionnalité de création d'image est créée. Vous enregistrez une image dans la table Images, en utilisant un UUID généré de façon aléatoire comme ID de l'image et en utilisant le nom d'utilisateur Cognito comme propriétaire de l'image.

Mutation.createFriendship

Dans l'éditeur de schéma de la console AWS AppSync , sur le côté droit, choisissez Joindre un résolveur pour `createFriendship(id: ID!, target: ID!): Boolean`. Choisissez la source de données FriendsDynamoDynamoDB DBTable. Dans la section request mapping template (modèle de mappage de requête), ajoutez le modèle suivant :

```
#set($userToFriendFriendship = { "userId" : "$ctx.args.id", "friendId":
 "$ctx.args.target" })
#set($friendToUserFriendship = { "userId" : "$ctx.args.target", "friendId":
 "$ctx.args.id" })
#set($friendsItems = [$util.dynamodb.toMapValues($userToFriendFriendship),
 $util.dynamodb.toMapValues($friendToUserFriendship)])

{
    "version" : "2018-05-29",
    "operation" : "BatchPutItem",
    "tables" : {
        ## Replace 'AppSyncTutorial-' default below with the ResourceNamePrefix you
 provided in the CloudFormation template
        "AppSyncTutorial-Friends": $util.toJson($friendsItems)
    }
}
```

Important : Dans le modèle de BatchPutItemdemande, le nom exact de la table DynamoDB doit figurer. Le nom de table par défaut est AppSyncTutorial-Friends. Si vous utilisez le mauvais nom de table, un message d'erreur s'affiche lorsque AppSync vous essayez d'assumer le rôle fourni.

Pour simplifier ce didacticiel, procédez comme si la demande d'amitié avait été approuvée et enregistrez l'entrée de relation directement dans le AppSyncTutorialFriendstableau.

En effet, vous stockez deux éléments pour chaque amitié car la relation est bidirectionnelle. Pour plus d'informations sur les meilleures pratiques d'Amazon DynamoDB en matière de many-to-many représentation des relations, consultez la section Meilleures pratiques DynamoDB.

Dans la section response mapping template (modèle de mappage de réponse), ajoutez le modèle suivant :

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
true
```

Remarque : Assurez-vous que votre modèle de requête contient le bon nom de table. Le nom par défaut est AppSyncTutorial-Friends, mais le nom de votre table peut être différent si vous modifiez le CloudFormation ResourceNamePrefixparamètre.

Requête. getPicturesByPropriétaire

Maintenant que vous avez des relations d'amitié et des images, vous devez offrir aux utilisateurs la possibilité de voir les photos de leurs amis. Pour satisfaire à cette exigence, vous devez d'abord vérifier que le demandeur est ami avec le propriétaire, puis demander les images.

Cette fonctionnalité requiert deux opérations de source de données, vous allez donc créer deux fonctions. La première fonction, isFriend, vérifie si le demandeur et le propriétaire sont amis. La deuxième fonction, getPicturesByOwner, récupère les photos demandées à partir d'un identifiant de propriétaire. Examinons le flux d'exécution ci-dessous pour le résolveur proposé sur la requête. getPicturesByChamp propriétaire :

1. Modèle de mappage Avant : préparer le contexte et les arguments d'entrée du champ.
2. Fonction isFriend : vérifie si le demandeur est le propriétaire de l'image. Dans le cas contraire, il vérifie si les utilisateurs demandeur et propriétaire sont amis en effectuant une opération GetItem DynamoDB sur la table des amis.
3. getPicturesByFonction propriétaire : récupère les images de la table Pictures à l'aide d'une opération de requête DynamoDB sur l'index secondaire global owner-index.
4. Modèle de mappage Après : mappe les résultats d'images pour que les attributs DynamoDB mappent correctement vers les champs de type GraphQL attendus.

Créez d'abord les fonctions.

Fonction isFriend

1. Choisissez l'onglet Functions (Fonctions).
2. Choisissez Create Function (Créer une fonction) pour créer une fonction.
3. Pour le nom de la source de données, saisissez `FriendsDynamoDBTable`.
4. Pour le nom de la fonction, saisissez isFriend.
5. Dans la zone de texte du modèle de mappage de requête, collez le modèle suivant :

```
#set($ownerId = $ctx.prev.result.owner)
#set($callerId = $ctx.prev.result.callerId)

## if the owner is the caller, no need to make the check
#if($ownerId == $callerId)
    #return($ctx.prev.result)
#end

{
    "version" : "2018-05-29",

    "operation" : "GetItem",

    "key" : {
        "userId" : $util.dynamodb.toDynamoDBJson($callerId),
        "friendId" : $util.dynamodb.toDynamoDBJson($ownerId)
    }
}
```

6. Dans la zone de texte du modèle de mappage de réponse, collez le modèle suivant :

```
#if($ctx.error)
    $util.error("Unable to retrieve friend mapping message: ${ctx.error.message}",
 $ctx.error.type)
#end

## if the users aren't friends
#if(!$ctx.result)
    $util.unauthorized()
#end
```

```
$util.toJson($ctx.prev.result)
```

7. Choisissez Create Function (Créer une fonction).

Résultat : vous avez créé la fonction isFriend.

getPicturesByFonction du propriétaire

1. Choisissez l'onglet Functions (Fonctions).
2. Choisissez Create Function (Créer une fonction) pour créer une fonction.
3. Pour le nom de la source de données, saisissez `PicturesDynamoDBTable`.
4. Pour le nom de la fonction, saisissez `getPicturesByOwner`.
5. Dans la zone de texte du modèle de mappage de requête, collez le modèle suivant :

```
{
    "version" : "2018-05-29",

    "operation" : "Query",

    "query" : {
        "expression": "#owner = :owner",
        "expressionNames": {
            "#owner" : "owner"
        },
        "expressionValues" : {
            ":owner" : $util.dynamodb.toDynamoDBJson($ctx.prev.result.owner)
        }
    },

    "index": "owner-index"
}
```

6. Dans la zone de texte du modèle de mappage de réponse, collez le modèle suivant :

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end

$util.toJson($ctx.result)
```

7. Choisissez Create Function (Créer une fonction).

Résultat : vous avez créé la fonction getPicturesByPropriétaire. Maintenant que les fonctions ont été créées, attachez un résolveur de pipeline à la requête. getPicturesByChamp du propriétaire.

Dans l'éditeur de schéma de la console AWS AppSync , sur le côté droit, choisissez Joindre un résolveur pour `Query.getPicturesByOwner(id: ID!): [Picture]`. Sur la page suivante, choisissez le lien Convert to pipeline resolver (Convertir en résolveur de pipeline) qui s'affiche sous la source de données dans la liste déroulante. Utilisez ce qui suit pour le modèle de mappage Avant :

```
#set($result = { "owner": $ctx.args.id, "callerId": $ctx.identity.username })
$util.toJson($result)
```

Dans la section after mapping template (modèle de mappage après), utilisez ce qui suit :

```
#foreach($picture in $ctx.result.items)
    ## prepend "src://" to picture.src property
    #set($picture['src'] = "src://${picture['src']}")
#end
$util.toJson($ctx.result.items)
```

Choisissez Create Resolver (Créer un résolveur). Vous avez réussi à joindre votre première résolveur de pipeline. Sur la même page, ajoutez les deux fonctions que vous avez précédemment créées. Dans la section des fonctions, choisissez Add A Function (Ajouter une fonction), puis choisissez ou saisissez le nom de la première fonction, isFriend. Ajoutez la deuxième fonction en suivant le même processus pour la fonction getPicturesByOwner. Assurez-vous que la fonction IsFriend apparaît en premier dans la liste, suivie de la fonction getPicturesByOwner. Vous pouvez utiliser les flèches vers le haut et vers le bas pour réorganiser l'ordre d'exécution des fonctions dans le pipeline.

Maintenant que le résolveur de pipeline est créé et que vous avez attaché les fonctions, nous allons tester l'API GraphQL nouvellement créée.

Test de votre API GraphQL

Tout d'abord, vous devez remplir les images et les relations d'amitié en exécutant quelques mutations à l'aide de l'utilisateur administrateur que vous avez créé. Sur le côté gauche de la console AWS AppSync , choisissez l'onglet Queries (Requêtes).

createPicture Mutation

1. Dans la console AWS AppSync choisissez l'onglet Queries (Requêtes).
2. Choisissez Login With User Pools (Connexion avec les groupes d'utilisateur).

3. Dans le modal, entrez l'ID de client d'exemple Cognito créé par la CloudFormation pile (par exemple, 37solo6mmhh7k4v63cqdfgdg5d).
4. Entrez le nom d'utilisateur que vous avez passé en paramètre à la CloudFormation pile. La valeur par défaut est nadia.
5. Utilisez le mot de passe temporaire qui a été envoyé à l'e-mail que vous avez fourni en tant que paramètre à la CloudFormation pile (par exemple, UserPoolUserEmail).
6. Choisissez Login (Connexion). Vous devriez maintenant voir le bouton renommé Logout nadia, ou le nom d'utilisateur que vous avez choisi lors de la création de la CloudFormation pile (c'est-à-dire, UserPoolUsername).

Nous allons envoyer quelques mutations createPicture pour remplir la table d'images. Exécutez la requête GraphQL suivante dans la console :

```
mutation {
  createPicture(input:{
    owner: "nadia"
    src: "nadia.jpg"
  }) {
    id
    owner
    src
  }
}
```

La réponse doit être similaire à ce qui suit :

```
{
  "data": {
    "createPicture": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "owner": "nadia",
      "src": "nadia.jpg"
    }
  }
}
```

Ajoutons quelques images de plus :

```
mutation {
```

```
  createPicture(input:{
    owner: "shaggy"
    src: "shaggy.jpg"
  }) {
    id
    owner
    src
  }
}
```

```
mutation {
  createPicture(input:{
    owner: "rex"
    src: "rex.jpg"
  }) {
    id
    owner
    src
  }
}
```

Vous avez ajouté trois images en utilisant nadia en tant qu'utilisateur administrateur.

Mutation createFriendship

Nous allons ajouter une entrée d'amitié. Exécutez les mutations suivantes dans la console.

Remarque : vous devez être connecté en tant qu'utilisateur administrateur (l'utilisateur administrateur par défaut est nadia).

```
mutation {
  createFriendship(id: "nadia", target: "shaggy")
}
```

La réponse doit être similaire à ce qui suit :

```
{
  "data": {
    "createFriendship": true
  }
}
```

nadia et shaggy sont amis. rex n'est ami avec personne.

getPicturesByRequête du propriétaire

Pour cette étape, connectez-vous en tant qu'utilisateur nadia à l'aide des groupes d'utilisateurs Cognito, en utilisant les informations d'identification définies au début de ce didacticiel. Comme nadia, extrayez les images détenues par shaggy.

```
query {
    getPicturesByOwner(id: "shaggy") {
        id
        owner
        src
    }
}
```

Comme nadia et shaggy sont amis, la requête devrait renvoyer l'image adéquate.

```
{
  "data": {
    "getPicturesByOwner": [
      {
        "id": "05a16fba-cc29-41ee-a8d5-4e791f4f1079",
        "owner": "shaggy",
        "src": "src://shaggy.jpg"
      }
    ]
  }
}
```

De même, si nadia essaie d'extraire ses propres images,elle réussira. Le résolveur de pipeline a été optimisé pour éviter d'exécuter l'opération GetItem IsFriend dans ce cas. Essayez la requête suivante :

```
query {
    getPicturesByOwner(id: "nadia") {
        id
        owner
        src
    }
}
```

Si vous activez la journalisation sur votre API (dans le volet Settings (Paramètres), configurez le niveau de débogage comme ALL (TOUS), et exécutez à nouveau la même requête. Elle renvoie des journaux pour l'exécution du champ. En examinant les journaux, vous pouvez déterminer si la fonction isFriend a été renvoyée de façon précoce lors de l'étape Modèle de mappage de requête :

```
{
  "errors": [],
  "mappingTemplateType": "Request Mapping",
  "path": "[getPicturesByOwner]",
  "resolverArn": "arn:aws:appsync:us-west-2:XXXX:apis/XXXX/types/Query/fields/
getPicturesByOwner",
  "functionArn": "arn:aws:appsync:us-west-2:XXXX:apis/XXXX/functions/
o2f42p2jrfdl3dw7s6xub2csdfs",
  "functionName": "isFriend",
  "earlyReturnedValue": {
    "owner": "nadia",
    "callerId": "nadia"
  },
  "context": {
    "arguments": {
      "id": "nadia"
    },
    "prev": {
      "result": {
        "owner": "nadia",
        "callerId": "nadia"
      }
    },
    "stash": {},
    "outErrors": []
  },
  "fieldInError": false
}
```

La earlyReturnedValueclé représente les données renvoyées par la directive #return.

Enfin, même si Rex est membre du Viewers Cognito UserPool Group, et parce que Rex n'est ami avec personne, il ne pourra accéder à aucune des photos détenues par Shaggy ou Nadia. Si vous vous connectez en tant que rex dans la console et exécutez la requête suivante :

```
query {
    getPicturesByOwner(id: "nadia") {
```

```
        id
        owner
        src
    }
}
```

Vous obtenez l'erreur de non autorisation suivante :

```
{
  "data": {
    "getPicturesByOwner": null
  },
  "errors": [
    {
      "path": [
        "getPicturesByOwner"
      ],
      "data": null,
      "errorType": "Unauthorized",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 9,
          "sourceName": null
        }
      ],
      "message": "Not Authorized to access getPicturesByOwner on type Query"
    }
  ]
}
```

Vous avez réussi à mettre en place une autorisation complexe à l'aide de résolveurs de pipeline.

Tutoriel : Delta Sync

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Les applications clientes dans AWS AppSync stockent des données en mettant localement en cache des réponses GraphQL sur un disque dans une application web/mobile. Les sources de données et les opérations `Sync` versionnées permettent aux clients d'effectuer le processus de synchronisation à l'aide d'un seul résolveur. Cela permet aux clients de remplir leur cache local avec les résultats d'une requête de base pouvant comprendre un grand nombre d'enregistrements, puis de recevoir uniquement les données modifiées depuis leur dernière requête (les mises à jour delta). En permettant aux clients de séparer l'hydratation de base du cache avec une demande initiale et les mises à jour incrémentielles d'une autre demande, vous pouvez déplacer le calcul de votre application cliente vers le backend. Ceci est nettement plus efficace pour les applications clientes qui basculent fréquemment entre les états en ligne et hors ligne.

Pour implémenter Delta Sync, la requête `Sync` utilise l'opération `Sync` sur une source de données versionnée. Lorsqu'une mutation AWS AppSync modifie un élément dans une source de données versionnée, un enregistrement de cette modification est également stocké dans la table Delta . Vous pouvez choisir d'utiliser différentes tables Delta (par exemple, une par type, une par domaine) pour les autres sources de données versionnées ou une seule table Delta pour votre API. AWS AppSync recommande de ne pas utiliser une seule table Delta pour plusieurs API afin d'éviter la collision des clés primaires.

En outre, les clients Delta Sync peuvent également recevoir un abonnement en tant qu'argument, puis les coordonne l'abonnement et écrit entre les transitions en lige et hors ligne. Delta Sync effectue cette opération en relançant automatiquement les abonnements, notamment le backoff exponentiel et réessaie avec une instabilité via différents scénarios d'erreur et le stockage des événements dans une file d'attente. La requête de base ou delta appropriée est ensuite exécutée avant de fusionner des événements de la file d'attente, puis de traiter les abonnements normalement.

La documentation des options de configuration du client, y compris l'Amplify DataStore, est disponible sur le site Web Amplify Framework. Cette documentation explique comment configurer les sources de données DynamoDB versionnées et les opérations `Sync` pour qu'elles fonctionnent avec le client Delta Sync pour un accès optimal aux données.

Configuration en un clic

Pour configurer automatiquement le point de terminaison GraphQL AWS AppSync avec tous les résolveurs configurés et les AWS ressources nécessaires, utilisez ce modèle : AWS CloudFormation

Cette pile crée les ressources suivantes dans votre compte :

- 2 tables DynamoDB (Base et Delta)
- 1 API AWS AppSync avec clé d'API
- 1 rôle IAM avec politique pour les tables DynamoDB

Deux tables sont utilisées pour partitionner vos requêtes de synchronisation dans une seconde table agissant comme un journal d'événements manqués lorsque les clients étaient hors ligne. Pour maintenir l'efficacité des requêtes sur la table delta, vous pouvez utiliser des TTL Amazon DynamoDB afin de préparer automatiquement les événements si nécessaire. L'heure TTL est configurable pour vos besoins sur la source de données (vous pouvez définir 1 heure, 1 jour, etc.).

Schema

Pour illustrer Delta Sync, l'exemple d'application crée un schéma Posts basé sur une table Base et Delta dans DynamoDB. AWS AppSync écrit automatiquement les mutations dans les deux tables. La requête de synchronisation tire des enregistrements des tables de Base ou Delta selon les besoins, et un seul abonnement est défini pour montrer comment les clients peuvent en tirer parti dans leur logique de reconnexion.

```
input CreatePostInput {
    author: String!
    title: String!
    content: String!
    url: String
    ups: Int
    downs: Int
    _version: Int
}

interface Connection {
  nextToken: String
  startedAt: AWSTimestamp!
}

type Mutation {
    createPost(input: CreatePostInput!): Post
    updatePost(input: UpdatePostInput!): Post
    deletePost(input: DeletePostInput!): Post
}
```

```
type Post {
    id: ID!
    author: String!
    title: String!
    content: String!
    url: AWSURL
    ups: Int
    downs: Int
    _version: Int
    _deleted: Boolean
    _lastChangedAt: AWSTimestamp!
}

type PostConnection implements Connection {
    items: [Post!]!
    nextToken: String
    startedAt: AWSTimestamp!
}

type Query {
    getPost(id: ID!): Post
    syncPosts(limit: Int, nextToken: String, lastSync: AWSTimestamp): PostConnection!
}

type Subscription {
    onCreatePost: Post
        @aws_subscribe(mutations: ["createPost"])
    onUpdatePost: Post
        @aws_subscribe(mutations: ["updatePost"])
    onDeletePost: Post
        @aws_subscribe(mutations: ["deletePost"])
}

input DeletePostInput {
    id: ID!
    _version: Int!
}

input UpdatePostInput {
    id: ID!
    author: String
    title: String
    content: String
```

```
    url: String
    ups: Int
    downs: Int
    _version: Int!
}

schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
}
```

Le schéma GraphQL est standard, mais il peut être nécessaire d'appeler avant de continuer, pour deux raisons. Tout d'abord, toutes les mutations écrivent automatiquement dans la table Base en premier lieu, puis dans la table Delta. La table de Base est la source de vérité centrale pour l'état tandis que la table Delta est votre journal. Si vous ne passez pas dans la `lastSync: AWSTimestamp`, la requête `syncPosts` s'exécute sur la table de Base et hydrate le cache tout en s'exécutant périodiquement en tant que processus de rattrapage global pour les cas limites où les clients sont hors ligne plus longtemps que le temps TTL que vous aviez configuré dans la table Delta. Si vous passez dans le `lastSync: AWSTimestamp`, la requête `syncPosts` s'exécute sur votre table Delta et est utilisée par les clients pour récupérer des événements modifiés la dernière fois qu'ils étaient hors ligne. Les clients Amplify transmettent automatiquement la valeur `lastSync: AWSTimestamp` et persistent sur le disque de manière appropriée.

Le champ _deleted sur Post est utilisé pour SUPPRIMER des opérations. Lorsque des clients sont hors ligne et que des enregistrements sont supprimés de la table Base, cet attribut informe les clients effectuant la synchronisation d'expulser les éléments de leur cache local. Si des clients sont hors ligne pour de plus longues périodes et que l'élément a été supprimé avant que le client ne récupère cette valeur avec une requête Delta Sync, l'événement de rattrapage global dans la requête de base (configurable dans le client), s'exécute et supprime l'élément à partir du cache. Ce champ est marqué comme facultatif car il ne renvoie de valeur que lors de l'exécution d'une requête de synchronisation ayant supprimé les objets présents.

Mutations

Pour toutes les mutations, AWS AppSync effectue une opération de création/mise à jour/suppression standard dans la table Base et enregistre également automatiquement la modification dans la table Delta. Vous pouvez réduire ou prolonger la période de conservation des enregistrements en modifiant la valeur `DeltaSyncTableTTL` sur la source de données. Pour les organisations avec une

vitesse de données élevée, il est recommandé de la garder courte. Si vos clients sont hors ligne plus longtemps, il peut être prudent de la garder plus longue.

Requêtes de synchronisation

La requête de base est une opération DynamoDB Sync sans `lastSync` valeur spécifiée. Cela fonctionne pour de nombreuses organisations car la requête de base ne s'exécute qu'au démarrage et par la suite, à intervalles réguliers.

La requête delta est une opération de synchronisation DynamoDB avec `lastSync` une valeur spécifiée. La requête delta s'exécute chaque fois que le client se reconnecte alors qu'il était hors ligne (à condition que le temps périodique de requête de base n'ait pas déclenchée l'exécution). Les clients suivent automatiquement la dernière fois qu'ils ont exécuté une requête pour synchroniser les données.

Lorsqu'une requête delta est exécutée, le résolveur de la requête utilise `ds_pk` et `ds_sk` pour interroger uniquement les enregistrements qui ont changé depuis la dernière synchronisation par le client. Le client stocke la réponse GraphQL appropriée.

Pour de plus amples informations sur l'exécution des requêtes de synchronisation, veuillez consulter la documentation Opération de synchronisation.

Exemple

Commençons d'abord par appeler une mutation `createPost` pour créer un élément :

```
mutation create {
  createPost(input: {author: "Nadia", title: "My First Post", content: "Hello World"})
 {
    id
    author
    title
    content
    _version
    _lastChangedAt
    _deleted
  }
}
```

La valeur de retour de cette mutation sera la suivante :

```
{
  "data": {
    "createPost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
      "author": "Nadia",
      "title": "My First Post",
      "content": "Hello World",
      "_version": 1,
      "_lastChangedAt": 1574469356331,
      "_deleted": null
    }
  }
}
```

Si vous examinez le contenu de la table Base, vous verrez un enregistrement qui ressemble à :

```
{
  "_lastChangedAt": {
    "N": "1574469356331"
  },
  "_version": {
    "N": "1"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "My First Post"
  }
}
```

Si vous examinez le contenu de la table Delta, vous verrez un enregistrement qui ressemble à :

```
{
  "_lastChangedAt": {
    "N": "1574469356331"
```

```
  },
  "_ttl": {
    "N": "1574472956"
  },
  "_version": {
    "N": "1"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "ds_pk": {
    "S": "AppSync-delta-sync-post:2019-11-23"
  },
  "ds_sk": {
    "S": "00:35:56.331:81d36bbb-1579-4efe-92b8-2e3f679f628b:1"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "My First Post"
  }
}
```

Maintenant, nous pouvons simuler une requête de base qu'un client va exécuter pour hydrater son magasin de données local en utilisant une requête `syncPosts` comme :

```
query baseQuery {
  syncPosts(limit: 100, lastSync: null, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
      _lastChangedAt
    }
    startedAt
    nextToken
  }
```

```
}
```

La valeur de retour de cette requête de base sera la suivante :

```
{
  "data": {
    "syncPosts": {
      "items": [
        {
          "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
          "author": "Nadia",
          "title": "My First Post",
          "content": "Hello World",
          "_version": 1,
          "_lastChangedAt": 1574469356331
        }
      ],
      "startedAt": 1574469602238,
      "nextToken": null
    }
  }
}
```

Nous enregistrerons la valeur `startedAt` plus tard pour simuler une requête Delta mais nous devons d'abord apporter une modification à notre table. Utilisons la mutation `updatePost` pour modifier notre publication existante :

```
mutation updatePost {
  updatePost(input: {id: "81d36bbb-1579-4efe-92b8-2e3f679f628b", _version: 1, title:
 "Actually this is my Second Post"}) {
    id
    author
    title
    content
    _version
    _lastChangedAt
    _deleted
  }
}
```

La valeur de retour de cette mutation sera la suivante :

```
{
  "data": {
    "updatePost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
      "author": "Nadia",
      "title": "Actually this is my Second Post",
      "content": "Hello World",
      "_version": 2,
      "_lastChangedAt": 1574469851417,
      "_deleted": null
    }
  }
}
```

Si vous examinez maintenant le contenu de la table Base vous devriez voir l'élément mis à jour :

```
{
  "_lastChangedAt": {
    "N": "1574469851417"
  },
  "_version": {
    "N": "2"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "Actually this is my Second Post"
  }
}
```

Si vous examinez maintenant le contenu de la table Delta, vous devriez voir deux enregistrements :

1. Enregistrement lors de la création de l'élément
2. Enregistrement de la date à laquelle l'élément a été mis à jour.

Le nouvel élément ressemblera à :

```
{
  "_lastChangedAt": {
    "N": "1574469851417"
  },
  "_ttl": {
    "N": "1574473451"
  },
  "_version": {
    "N": "2"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "ds_pk": {
    "S": "AppSync-delta-sync-post:2019-11-23"
  },
  "ds_sk": {
    "S": "00:44:11.417:81d36bbb-1579-4efe-92b8-2e3f679f628b:2"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "Actually this is my Second Post"
  }
}
```

Maintenant, nous pouvons simuler une requête Delta pour récupérer les modifications qui se sont produites lorsqu'un client était hors ligne. Nous utiliserons la valeur `startedAt` renvoyée par notre requête de Base pour effectuer la requête :

```
query delta {
  syncPosts(limit: 100, lastSync: 1574469602238, nextToken: null) {
    items {
      id
      author
      title
      content
```

```
      _version
    }
    startedAt
    nextToken
  }
}
```

La valeur de retour de cette requête Delta sera la suivante :

```
{
  "data": {
    "syncPosts": {
      "items": [
        {
          "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
          "author": "Nadia",
          "title": "Actually this is my Second Post",
          "content": "Hello World",
          "_version": 2
        }
      ],
      "startedAt": 1574470400808,
      "nextToken": null
    }
  }
}
```

Configuration et réglages

AWS AppSync vous permet d'effectuer les opérations suivantes :

- Données de cache souvent demandées mais peu susceptibles de changer d'une demande à l'autre. Cela peut réduire la charge de travail de vos résolveurs. Pour plus d'informations, consultez the section called “Mise en cache et compression”.
- Versionnez des objets GraphQL pour gérer et éviter les conflits entre plusieurs clients. Pour plus d'informations, consultez the section called “Détection et synchronisation des conflits”.
- Utilisez des noms de domaine personnalisés pour configurer un domaine unique et mémorable qui fonctionne à la fois pour votre GraphQL et pour les API en temps réel. Pour plus d'informations, consultez la section Configuration de noms de domaine personnalisés.
- Autorisez l'accès à vos API GraphQL via un VPC. Pour plus d'informations, consultez la section Utilisation AWS AppSync d'API privées.
- Activez l'introspection et définissez la profondeur des requêtes et les limites de résolution par requête. Pour plus d'informations, consultez la section Limites de configuration.

AWS AppSyncInclut en outre les AWS outils standard suivants pour la journalisation, la surveillance et le suivi :

- Se connecter AWS CloudTrail
- Surveillance avec Amazon CloudWatch
- Traçage avec AWS X-Ray

Mise en cache et compression

AWS AppSyncles fonctionnalités de mise en cache des données côté serveur rendent les données disponibles dans un cache en mémoire à haut débit, améliorant ainsi les performances et réduisant la latence. Cela réduit le besoin d'accéder directement aux sources de données. La mise en cache est disponible pour les résolveurs d'unités et de pipelines.

AWS AppSyncvous permet également de compresser les réponses des API afin que le contenu de la charge utile se charge et se télécharge plus rapidement. Cela réduit potentiellement la charge qui pèse sur vos applications tout en réduisant potentiellement vos frais de transfert de données. Le comportement de compression est configurable et peut être défini à votre guise.

Reportez-vous à cette section pour obtenir de l'aide pour définir le comportement souhaité de la mise en cache et de la compression côté serveur dans votre API. AWS AppSync

Types d'instances

AWS AppSynchéberge des instances Amazon ElastiCache pour Redis dans le même AWS compte et dans la même AWS région que votre AWS AppSync API.

Les types ElastiCache d'instances Redis suivants sont disponibles :

petit

1 vCPU, 1,5 GiB de RAM, performances réseau faibles à modérées

medium

2 vCPU, 3 GiB de RAM, performances réseau faibles à modérées

large

2 vCPU, 12,3 GiB de RAM, performances réseau allant jusqu'à 10 Gigabit

xlarge

4 vCPU, 25,05 GiB de RAM, performances réseau allant jusqu'à 10 Gigabit

2xlarge

8 vCPU, 50,47 GiB de RAM, performances réseau allant jusqu'à 10 Gigabit

4xlarge

16 vCPU, 101,38 GiB de RAM, performances réseau allant jusqu'à 10 Gigabit

8xlarge

32 vCPU, 203,26 GiB de RAM, performances réseau 10 Gigabit (non disponible dans toutes les régions)

12xlarge

48 vCPU, 317,77 GiB de RAM, performance réseau 10 Gigabit

> **Note**
>
> Historiquement, vous spécifiiez un type d'instance spécifique (tel `quet2.medium`). En juillet 2020, ces anciens types d'instances sont toujours disponibles, mais leur utilisation

est déconseillée et déconseillée. Nous vous recommandons d'utiliser les types d'instances génériques décrits ici.

Comportement de mise en cache

Les comportements liés à la mise en cache sont les suivants :

Aucune

Pas de mise en cache côté serveur.

Mise en cache complète des requêtes

Si les données ne se trouvent pas dans le cache, elles sont extraites de la source de données et renseignées dans le cache jusqu'à l'expiration de la durée de vie (TTL). Toutes les demandes suivantes adressées à votre API sont renvoyées depuis le cache. Cela signifie que les sources de données ne sont pas contactées directement à moins que le TTL n'expire. Dans ce paramètre, nous utilisons le contenu des `context.identity` cartes `context.arguments` et comme clés de mise en cache.

Mise en cache par résolveur

Avec ce paramètre, chaque résolveur doit être explicitement activé pour qu'il mette en cache les réponses. Vous pouvez spécifier un TTL et des clés de mise en cache sur le résolveur. Les clés de mise en cache que vous pouvez spécifier sont les cartes de niveau supérieur `context.arguments` `context.sourcecontext.identity`, et/ou les champs de chaîne de ces cartes. La valeur TTL est obligatoire, mais les clés de mise en cache sont facultatives. Si vous ne spécifiez aucune clé de mise en cache, les valeurs par défaut sont le contenu des cartes `context.argumentscontext.source`, et`context.identity`.

Par exemple, vous pouvez utiliser les combinaisons suivantes :

- context.arguments et context.source
- context.arguments et context.identity.sub
- context.arguments.id ou context.arguments. InputType.id
- context.source.id et context.identity.sub
- context.identity.claims.nom d'utilisateur

Lorsque vous spécifiez uniquement un TTL et aucune clé de mise en cache, le comportement du résolveur est identique à celui de la mise en cache complète des demandes.

Durée de vie du cache

Ce paramètre définit la durée pendant laquelle les entrées mises en cache sont stockées en mémoire. Le TTL maximal est de 3 600 secondes (1 heure), après quoi les entrées sont automatiquement supprimées.

Chiffrement du cache

Le chiffrement du cache est disponible sous les deux formes suivantes. Ils sont similaires aux paramètres autorisés ElastiCache par Redis. Vous ne pouvez activer les paramètres de chiffrement que lors de la première activation de la mise en cache pour votre AWS AppSync API.

- Chiffrement en transit — Les requêtes entre AWS AppSync le cache et les sources de données (à l'exception des sources de données HTTP non sécurisées) sont cryptées au niveau du réseau. Certains traitements étant nécessaires pour chiffrer et déchiffrer les données au niveau des terminaux, le chiffrement en transit peut avoir un impact sur les performances.
- Chiffrement au repos : les données enregistrées sur le disque depuis la mémoire lors des opérations de swap sont chiffrées sur l'instance de cache. Ce paramètre a également un impact sur les performances.

Pour invalider les entrées du cache, vous pouvez effectuer un appel d'API Flush Cache à l'aide de la AWS AppSync console ou du AWS Command Line Interface ()AWS CLI.

Pour plus d'informations, consultez le type de ApiCachedonnées dans la référence de l'AWS AppSyncAPI.

Expulsion du cache

Lorsque vous configurez AWS AppSync la mise en cache côté serveur, vous pouvez configurer un TTL maximal. Cette valeur définit la durée pendant laquelle les entrées mises en cache sont stockées en mémoire. Dans les situations où vous devez supprimer des entrées spécifiques de votre cache, vous pouvez utiliser l'utilitaire AWS AppSync d'`evictFromApiCache`extensions dans la demande ou la réponse de votre résolveur. (Par exemple, lorsque les données de vos sources de données ont changé et que votre entrée de cache est désormais obsolète.) Pour expulser un élément du cache, vous devez connaître sa clé. C'est pourquoi, si vous devez expulser des éléments de manière dynamique, nous vous recommandons d'utiliser la mise en cache par résolveur et de définir explicitement une clé à utiliser pour ajouter des entrées à votre cache.

Expulsion d'une entrée de cache

Pour expulser un élément du cache, utilisez l'utilitaire d'`evictFromApiCache`extensions. Spécifiez le nom du type et le nom du champ, puis fournissez un objet contenant des éléments clé-valeur pour créer la clé de l'entrée que vous souhaitez expulser. Dans l'objet, chaque clé représente une entrée valide provenant de l'`context`objet utilisé dans la liste du résolveur mis en cache. `cachingKey` Chaque valeur est la valeur réelle utilisée pour construire la valeur de la clé. Vous devez placer les éléments de l'objet dans le même ordre que les clés de mise en cache dans la liste du résolveur mis en cache. `cachingKey`

Par exemple, consultez le schéma suivant :

```
type Note {
  id: ID!
  title: String
  content: String!
}

type Query {
  getNote(id: ID!): Note
}

type Mutation {
  updateNote(id: ID!, content: String!): Note
}
```

Dans cet exemple, vous pouvez activer la mise en cache par résolveur, puis l'activer pour la requête. `getNote` Ensuite, vous pouvez configurer la clé de mise en cache pour qu'elle soit composée de`[context.arguments.id]`.

Lorsque vous essayez d'obtenir une`Note`, pour créer la clé de cache, AWS AppSync effectue une recherche dans son cache côté serveur à l'aide de l'`id`argument de la `getNote` requête.

Lorsque vous mettez à jour un`Note`, vous devez supprimer l'entrée correspondant à la note spécifique afin de vous assurer que la demande suivante la récupère depuis la source de données principale. Pour cela, vous devez créer un gestionnaire de demandes.

L'exemple suivant montre une façon de gérer l'expulsion à l'aide de cette méthode :

```
import { util, Context } from '@aws-appsync/utils';
import { update } from '@aws-appsync/utils/dynamodb';
```

```
export function request(ctx) {
 extensions.evictFromApiCache('Query', 'getNote', { 'ctx.args.id': ctx.args.id });
 return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });
}

export const response = (ctx) => ctx.result;
```

Vous pouvez également gérer l'expulsion dans le gestionnaire de réponses.

Lorsque la `updateNote` mutation est traitée, AWS AppSync tente d'expulser l'entrée. Si une entrée est correctement effacée, la réponse contient une `apiCacheEntriesDeleted` valeur dans l'`extensions`objet qui indique le nombre d'entrées supprimées :

```
"extensions": {  "apiCacheEntriesDeleted": 1}
```

Expulsion d'une entrée de cache en fonction de son identité

Vous pouvez créer des clés de mise en cache basées sur plusieurs valeurs de l'`context`objet.

Par exemple, prenons le schéma suivant qui utilise les groupes d'utilisateurs Amazon Cognito comme mode d'authentification par défaut et est soutenu par une source de données Amazon DynamoDB :

```
type Note {
  id: ID! # a slug; e.g.: "my-first-note-on-graphql"
  title: String
  content: String!
}

type Query {
  getNote(id: ID!): Note
}

type Mutation {
  updateNote(id: ID!, content: String!): Note
}
```

Les types `Note` d'objets sont enregistrés dans une table DynamoDB. La table contient une clé composite qui utilise le nom d'utilisateur Amazon Cognito comme clé primaire et le `id` (un slug) `Note` comme clé de partition. Il s'agit d'un système mutualisé qui permet à plusieurs utilisateurs d'héberger et de mettre à jour leurs `Note` objets privés, qui ne sont jamais partagés.

Comme il s'agit d'un système à lecture intensive, la `getNote` requête est mise en cache à l'aide de la mise en cache par résolveur, la clé de mise en cache étant composée de. `[context.identity.username, context.arguments.id]` Lorsqu'un `Note` est mis à jour, vous pouvez supprimer l'entrée correspondante. `Note` Vous devez ajouter les composants dans l'objet dans l'ordre dans lequel ils sont spécifiés dans la `cachingKeys` liste de votre résolveur.

L'exemple suivant illustre cela :

```
import { util, Context } from '@aws-appsync/utils';
import { update } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 extensions.evictFromApiCache('Query', 'getNote', {
  'ctx.identity.username': ctx.identity.username,
  'ctx.args.id': ctx.args.id,
 });
 return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });
}

export const response = (ctx) => ctx.result;
```

Un système principal peut également mettre à jour l'entrée `Note` et l'expulser. Par exemple, prenons cette mutation :

```
type Mutation {
  updateNoteFromBackend(id: ID!, content: String!, username: ID!): Note @aws_iam
}
```

Vous pouvez supprimer l'entrée, mais ajouter les composants de la clé de mise en cache à l'`cachingKeys`objet.

Dans l'exemple suivant, l'expulsion se produit dans la réponse du résolveur :

```
import { util, Context } from '@aws-appsync/utils';
import { update } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 extensions.evictFromApiCache('Query', 'getNote', {
  'ctx.identity.username': ctx.args.username,
  'ctx.args.id': ctx.args.id,
 });
```

```
 return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });
}

export const response = (ctx) => ctx.result;
```

Dans les cas où vos données de backend ont été mises à jour en dehors de AWS AppSync, vous pouvez expulser un élément du cache en appelant une mutation utilisant une source de `NONE` données.

Compression des réponses de l'API

AWS AppSync permet aux clients de demander des charges utiles compressées. Sur demande, les réponses de l'API sont compressées et renvoyées en réponse aux demandes indiquant que le contenu compressé est préféré. Les réponses d'API compressées se chargent plus rapidement, le contenu est téléchargé plus rapidement et vos frais de transfert de données peuvent également être réduits.

> **Note**
>
> La compression est disponible sur toutes les nouvelles API créées après le 1er juin 2020. AWS AppSync compresse les objets au mieux. Dans de rares cas, la compression AWS AppSync peut être ignorée en fonction de divers facteurs, notamment de la capacité actuelle.

AWS AppSync peut compresser des tailles de charge utile de requêtes GraphQL comprises entre 1 000 et 10 000 000 octets. Pour activer la compression, le client doit envoyer l'`Accept-Encoding`en-tête avec la valeur`gzip`. La compression peut être vérifiée en vérifiant la valeur de l'`Content-Encoding`en-tête dans la réponse (`gzip`).

L'explorateur de requêtes de la AWS AppSync console définit automatiquement la valeur d'en-tête de la demande par défaut. Si vous exécutez une requête dont le taux de réponse est suffisamment élevé, la compression peut être confirmée à l'aide des outils de développement de votre navigateur.

Configuration de noms de domaine personnalisés

AvecAWS AppSync, vous pouvez utiliser des noms de domaine personnalisés pour configurer un domaine unique et mémorable qui fonctionne à la fois pour votre GraphQL et pour les API en temps réel.

En d'autres termes, vous pouvez utiliser des URL de point de terminaison simples et mémorisables avec les noms de domaine de votre choix en créant des noms de domaine personnalisés que vous associez auAWS AppSyncAPI dans votre compte.

Lorsque vous configurez unAWS AppSyncAPI, deux points de terminaison sont provisionnés :

AWS AppSyncPoint de terminaison GraphQL :

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSyncpoint final en temps réel :

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/
graphql
```

Avec les noms de domaine personnalisés, vous pouvez interagir avec les deux points de terminaison en utilisant un seul domaine. Par exemple, si vous configurez`api.example.com`en tant que domaine personnalisé, vous pouvez interagir à la fois avec votre GraphQL et avec vos points de terminaison en temps réel à l'aide des URL suivantes :

AWS AppSyncpoint de terminaison GraphQL de domaine personnalisé :

```
https://api.example.com/graphql
```

AWS AppSyncpoint de terminaison en temps réel de domaine personnalisé :

```
wss://api.example.com/graphql/realtime
```

> **Note**
>
> AWS AppSyncLes API ne prennent en charge que les protocoles TLS 1.2 et TLS 1.3 pour les noms de domaine personnalisés.

Enregistrement et configuration d'un nom de domaine

Pour configurer des noms de domaine personnalisés pour votreAWS AppSyncAPI, vous devez avoir un nom de domaine Internet enregistré. Vous pouvez enregistrer un domaine Internet en utilisantAmazon Route 53 domain registrationou un bureau d'enregistrement de domaines tiers de votre choix. Pour plus d'informations sur la Route 53, voirQu'est-ce qu'Amazon Route 53 ?dans leGuide du développeur Amazon Route 53.

Le nom de domaine personnalisé d'une API peut être le nom d'un sous-domaine ou le domaine racine (également appelé « zone apex ») d'un domaine Internet enregistré. Après avoir créé un nom de domaine personnalisé dansAWS AppSync, vous devez créer ou mettre à jour l'enregistrement des ressources de votre fournisseur DNS pour le mapper à votre point de terminaison d'API. Sans ce mappage, les demandes d'API liées au nom de domaine personnalisé ne peuvent pas atteindreAWS AppSync.

Création d'un nom de domaine personnalisé dansAWS AppSync

Création d'un nom de domaine personnalisé pour unAWS AppSyncL'API met en place unAmazon CloudFrontdistribution. Vous devez configurer un enregistrement DNS afin de mapper le nom de domaine personnalisé àCloudFrontnom de domaine de distribution. Ce mappage est nécessaire pour acheminer les demandes d'API liées au nom de domaine personnaliséAWS AppSyncà travers le mappageCloudFrontdistribution. Vous devez également fournir un certificat pour le nom de domaine personnalisé.

Pour configurer le nom de domaine personnalisé ou pour mettre à jour son certificat, vous devez être autorisé à le mettre à jourCloudFrontdistributions et décrivez lesAWS Certificate Managercertificat (ACM) que vous prévoyez d'utiliser. Pour accorder ces autorisations, joignez les pièces suivantesAWS Identity and Access ManagementDéclaration de politique (IAM) à l'intention d'un utilisateur, d'un groupe ou d'un rôle IAM dans votre compte :

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowUpdateDistributionForAppSyncCustomDomainName",
      "Effect": "Allow",
      "Action": ["cloudfront:updateDistribution"],
      "Resource": ["*"]
    },
    {
      "Sid": "AllowDescribeCertificateForAppSyncCustomDomainName",
      "Effect": "Allow",
      "Action": "acm:DescribeCertificate",
      "Resource": "arn:aws:acm:<region>:<account-id>:certificate/<certificate-id>"
    }
  ]
}
```

AWS AppSyncprend en charge les noms de domaine personnalisés en tirant parti de l'indication du nom du serveur (SNI) surCloudFrontdistribution. Pour plus d'informations sur l'utilisation de noms de domaine personnalisés sur unCloudFrontdistribution, y compris le format de certificat requis et la longueur maximale de la clé de certificat, voirUtilisation du protocole HTTPS avecCloudFrontdans leAmazonCloudFrontGuide du développeur.

Pour configurer un nom de domaine personnalisé comme nom d'hôte de l'API, le propriétaire de l'API doit fournir un certificat SSL/TLS pour le nom de domaine personnalisé. Pour fournir un certificat, effectuez l'une des opérations suivantes :

- Demandez un nouveau certificat dans ACM ou importez un certificat émis par une autorité de certification tierce dans ACM dans le`us-east-1` AWSRégion (USA Est (Virginie du Nord)). Pour plus d'informations sur ACM, voirQu'est-ce queAWS Certificate Manager?dans leAWS Certificate ManagerGuide de l'utilisateur.
- Fournissez un certificat de serveur IAM. Pour plus d'informations, voirGestion des certificats de serveur dans IAMdans leGuide de l'utilisateur IAM.

Noms de domaine personnalisés Wildcard dansAWS AppSync

AWS AppSyncprend en charge les noms de domaine personnalisés avec caractères génériques. Pour configurer un nom de domaine personnalisé avec caractère générique, spécifiez un caractère générique (`*`) en tant que premier sous-domaine d'un domaine personnalisé. Cela représente tous les sous-domaines possibles du domaine racine. Par exemple, le nom de domaine personnalisé générique`*.example.com`donne lieu à des sous-domaines tels que`a.example.com`,`b.example.com`, et`c.example.com`. Tous ces sous-domaines sont acheminés vers le même domaine.

Pour utiliser un nom de domaine personnalisé avec caractère générique dansAWS AppSync, vous devez fournir un certificat émis par ACM contenant un nom générique capable de protéger plusieurs sites du même domaine. Pour plus d'informations, voirCaractéristiques du certificat ACMdans leAWS Certificate ManagerGuide de l'utilisateur.

Détection et synchronisation des conflits

Sources de données versionnées

AWS AppSync prend actuellement en charge le contrôle de version sur les sources de données DynamoDB. Les opérations de détection de conflits, de résolution de conflits et de synchronisation

nécessitent une source de données `Versioned`. Lorsque vous activez le contrôle de version sur une source de données, AWS AppSync va automatiquement :

- Améliorer les éléments avec les métadonnées de gestion des versions d'objets.
- Enregistrer les modifications apportées aux éléments ayant subi des mutations AWS AppSync dans une table Delta.
- Conservez les éléments supprimés dans la table Base avec une « désactivation » pendant une durée configurable.

Configuration de source de données versionnée

Lorsque vous activez le contrôle de version sur une source de données DynamoDB, vous spécifiez les champs suivants :

`BaseTableTTL`

Nombre de minutes pour conserver les éléments supprimés dans la table Base avec une « désactivation » - champ de métadonnées indiquant que l'élément a été supprimé. Vous pouvez définir cette valeur sur 0 si vous souhaitez que les éléments soient retirés immédiatement lorsqu'ils sont supprimés. Ce champ est obligatoire.

`DeltaSyncTableName`

Nom de la table dans laquelle les modifications apportées aux éléments avec des mutations AWS AppSync sont stockées. Ce champ est obligatoire.

`DeltaSyncTableTTL`

Nombre de minutes pour conserver les éléments dans la table Delta. Ce champ est obligatoire.

Table de synchronisation Delta

AWS AppSync prend actuellement en charge la journalisation Delta Sync pour les mutations utilisant`PutItem``UpdateItem`, et les opérations`DeleteItem` DynamoDB.

Lorsqu'une mutation AWS AppSync modifie un élément dans une source de données versionnée, un enregistrement de cette modification est stocké dans une table Delta optimisée pour les mises à jour incrémentielles. Vous pouvez choisir d'utiliser différentes tables Delta (par exemple, une par type, une par zone de domaine) pour d'autres sources de données versionnées ou une seule table Delta

pour votre API. AWS AppSync recommande de ne pas utiliser une seule table Delta pour plusieurs API afin d'éviter la collision de clés primaires.

Le schéma requis pour cette table est le suivant :

`ds_pk`

Valeur de chaîne utilisée comme clé de partition. Il est construit en concaténant le nom de la source de données de base et le format ISO 8601 de la date à laquelle la modification est survenue (par exemple`Comments:2019-01-01`).

Lorsque l'`customPartitionKey`indicateur du modèle de mappage VTL est défini comme nom de colonne de la clé de partition (voir Resolver Mapping Template Reference for DynamoDB dans le Guide duAWS AppSync développeur), le format`ds_pk` change et la chaîne est construite en y ajoutant la valeur de la clé de partition dans le nouvel enregistrement de la table de base. Par exemple, si l'enregistrement de la table de base possède une valeur de clé de partition`1a` et une valeur de clé de tri de`2b`, la nouvelle valeur de la chaîne sera :`Comments:2019-01-01:1a`.

`ds_sk`

Valeur de chaîne utilisée comme clé de tri. Il est construit en concaténant le format ISO 8601 indiquant l'heure à laquelle la modification s'est produite, la clé primaire de l'élément et la version de l'élément. La combinaison de ces champs garantit l'unicité de chaque entrée de`09:30:00` la table Delta (par exemple, pour une heure`2`, un identifiant`1a` et une version de`09:30:00:1a:2`).

Lorsque l'`customPartitionKey`indicateur du modèle de mappage VTL est défini sur le nom de colonne de la clé de partition (voir Resolver Mapping Template Reference for DynamoDB dans le Guide duAWS AppSync développeur), le format`ds_sk` change et la chaîne est construite en remplaçant la valeur de la clé combinée par la valeur de la clé de tri dans la table de base. En utilisant l'exemple précédent ci-dessus, si l'enregistrement de la table de base possède une valeur de clé de partition`1a` et une valeur de clé de tri de`2b`, la nouvelle valeur de la chaîne sera :`09:30:00:2b:3`.

`_ttl`

Valeur numérique qui stocke l'horodatage, en secondes d'époque, lorsqu'un élément doit être supprimé de la table Delta. Cette valeur est déterminée en ajoutant la valeur `DeltaSyncTableTTL` configurée sur la source de données au moment où la modification s'est produite. Ce champ doit être configuré comme Attribut TTL DynamoDB.

Le rôle IAM configuré pour être utilisé avec la table Base doit également contenir l'autorisation d'opérer sur la table Delta. Dans cet exemple, la stratégie d'autorisations d'une table Base appelée `Comments` et d'une table Delta appelée `ChangeLog` s'affiche :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:000000000000:table/Comments",
                "arn:aws:dynamodb:us-east-1:000000000000:table/Comments/*",
                "arn:aws:dynamodb:us-east-1:000000000000:table/ChangeLog",
                "arn:aws:dynamodb:us-east-1:000000000000:table/ChangeLog/*"
            ]
        }
    ]
}
```

Métadonnées de source de données versionnées

AWS AppSync gère les champs de métadonnées`Versioned` des sources de données en votre nom. La modification de ces champs vous-même peut entraîner des erreurs dans votre application ou une perte de données. Ces champs comprennent :

`_version`

Compteur à augmentation monotone qui est mis à jour chaque fois qu'un changement se produit sur un élément.

`_lastChangedAt`

Valeur numérique qui stocke l'horodatage, en millisecondes Epoch, lors de la dernière modification d'un élément.

`_deleted`

Valeur booléenne de désactivation qui indique qu'un élément a été supprimé. Cela peut être utilisé par les applications pour expulser les éléments supprimés des magasins de données locaux.

`_ttl`

Valeur numérique qui stocke l'horodatage, en secondes Epoch, lorsqu'un élément doit être supprimé de la source de données sous-jacente.

`ds_pk`

Valeur de chaîne utilisée comme clé de partition pour les tables Delta.

`ds_sk`

Valeur de chaîne utilisée comme clé de tri pour les tables Delta.

`gsi_ds_pk`

Attribut de valeur de chaîne généré pour prendre en charge un index secondaire global en tant que clé de partition. Il ne sera inclus que si les`populateIndexFields` indicateurs`customPartitionKey` et sont activés dans le modèle de mappage VTL (voir Resolver Mapping Template Reference for DynamoDB dans le Guide duAWS AppSync développeur). Si cette option est activée, la valeur sera construite en concaténant le nom de la source de données de base et le format ISO 8601 de la date à laquelle la modification s'est produite (par exemple, si la table de base est nommée Commentaires, cet enregistrement sera défini comme`Comments:2019-01-01`).

`gsi_ds_sk`

Attribut de valeur de chaîne généré pour prendre en charge un index secondaire global en tant que clé de tri. Il ne sera inclus que si les`populateIndexFields` indicateurs`customPartitionKey` et sont activés dans le modèle de mappage VTL (voir Resolver Mapping Template Reference for DynamoDB dans le Guide duAWS AppSync développeur). Si cette option est activée, la valeur sera construite en concaténant le format ISO 8601 de l'heure à laquelle la modification s'est produite, la clé de partition de l'élément dans la table de base, la clé de tri de l'élément dans la table de base et la version de l'élément (par exemple, pour une heure`09:30:00`, une valeur de clé de partition de`1a`, une valeur de`2b` clé de tri et une version de`3`, ce serait`09:30:00:1a#2b:3`).

Ces champs de métadonnées auront un impact sur la taille globale des éléments dans la source de données sous-jacente. AWS AppSync recommande de réserver un espace de stockage de 500 octets ou plus pour les métadonnées des sources de données versionnées lors de la conception de votre application. Pour utiliser ces métadonnées dans les applications clientes, incluez les champs `_version`, `_lastChangedAt` et `_deleted` sur vos types GraphQL et dans le jeu de sélection pour les mutations.

Détection et résolution des conflits

Lorsque des écritures simultanées se produisent avec AWS AppSync, vous pouvez configurer des stratégies de détection et de résolution des conflits pour gérer les mises à jour de manière appropriée. La détection des conflits détermine si la mutation est en conflit avec l'élément écrit réel dans la source de données. La détection des conflits est activée en définissant SyncConfig la valeur du`conflictDetection` champ sur`VERSION`.

La résolution des conflits est l'action qui est effectuée en cas de détection d'un conflit. Ceci est déterminé en définissant le champ Gestionnaire de conflits dans le SyncConfig. Il existe trois stratégies de résolution des conflits :

- OPTIMISTIC_CONCURRENCY
- AUTOMERGE
- LAMBDA

Chacune de ces stratégies de résolution de conflits est détaillée ci-dessous.

Les versions sont automatiquement incrémentées AppSync au cours des opérations d'écriture et ne doivent pas être modifiées par les clients ou en dehors d'un résolveur configuré avec une source de données compatible avec les versions. Cela modifiera le comportement de cohérence du système et pourrait entraîner une perte de données.

Simultanéité optimiste

La simultanéité optimiste est une stratégie de résolution de conflits fournie par AWS AppSync pour les sources de données versionnées. Lorsque le résolveur de conflits est défini sur Simultanéité optimiste, si une mutation entrante est détectée comme ayant une version différente de la version réelle de l'objet, le gestionnaire de conflits rejette simplement la demande entrante. Dans la réponse GraphQL, l'élément existant sur le serveur qui a la dernière version sera fourni. Le client doit alors gérer ce conflit localement et réessayer la mutation avec la version mise à jour de l'élément.

Automerge

Automerge fournit aux développeurs un moyen facile de configurer une stratégie de résolution de conflits sans écrire de logique côté client pour fusionner manuellement les conflits qui n'ont pas pu être gérés par d'autres stratégies. Automerge respecte un ensemble de règles strictes lors de la fusion de données pour résoudre les conflits. Les principes d'Automerge tournent autour du type de données sous-jacent du champ GraphQL. Ce sont les suivants :

- Conflit sur un champ scalaire : champ scalaire GraphQL ou tout autre champ qui n'est pas une collection (par exemple List, Set, Map). Rejetez la valeur entrante pour le champ scalaire et sélectionnez la valeur existante dans le serveur.
- Conflit sur une liste : le type GraphQL et le type de base de données sont des listes. Concaténez la liste entrante avec la liste existante dans le serveur. Les valeurs de liste dans la mutation entrante seront ajoutées à la fin de la liste dans le serveur. Les valeurs dupliquées seront conservées.
- Conflit sur un ensemble : le type GraphQL est une liste et le type de base de données est un ensemble. Appliquez une union de jeu à l'aide de l'ensemble entrant et de l'ensemble existant dans le serveur. Cela respecte les propriétés d'un ensemble, ce qui signifie qu'il n'y a aucune entrée en double.
- Lorsqu'une mutation entrante ajoute un nouveau champ à l'élément ou est effectuée sur un champ ayant la valeur de`null`, fusionnez-le avec l'élément existant.
- Conflit sur une carte : lorsque le type de données sous-jacent dans la base de données est une carte (c.-à-d. un document clé-valeur), appliquez les règles ci-dessus lors de l'analyse et du traitement de chaque propriété de la carte.

Automerge est conçu pour détecter, fusionner et réessayer automatiquement les demandes avec une version mise à jour, ce qui évite au client de devoir fusionner manuellement les données en conflit.

Pour montrer un exemple de la façon dont Automerge gère un conflit sur un type scalaire. Nous utiliserons le dossier suivant comme point de départ.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 4
}
```

Maintenant, une mutation entrante peut essayer de mettre à jour l'élément, mais avec une version plus ancienne puisque le client n'a pas encore été synchronisé avec le serveur. Elle se présente ainsi :

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 55,
  "_version" : 2
}
```

Notez la version obsolète de 2 dans la requête entrante. Pendant ce flux, Automerge fusionnera les données en rejetant la mise à jour du champ 'jersey' sur '55' et en conservant la valeur sur '5', ce qui entraîne l'enregistrement de l'image suivante de l'élément dans le serveur.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 5 # version is incremented every time automerge performs a merge that is
 stored on the server.
}
```

Compte tenu de l'état de l'élément indiqué ci-dessus dans la version 5, supposons maintenant une mutation entrante qui tente de muter l'élément avec l'image suivante :

```
{
  "id" : 1,
  "name" : "Shaggy",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner"] # underlying data type is a Set
  "points": [24, 30, 27] # underlying data type is a List
  "_version" : 3
}
```

Il y a trois points d'intérêt dans la mutation entrante. Le nom, un scalaire, a été modifié, mais deux nouveaux champs « intérêts », un Set et « points », une Liste, ont été ajoutés. Dans ce scénario, un conflit sera détecté en raison de l'incompatibilité de version. Automerge respecte ses propriétés et rejette le changement de nom parce qu'il est un scalaire et il procède à l'ajout sur les champs non conflictuels. Il en résulte que l'élément qui est enregistré dans le serveur s'affiche comme suit.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner"] # underlying data type is a Set
  "points": [24, 30, 27] # underlying data type is a List
  "_version" : 6
}
```

Avec l'image mise à jour de l'élément avec la version 6, supposons maintenant qu'une mutation entrante (avec une autre incompatibilité de version) essaie de transformer l'élément en ce qui suit :

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "brunch"] # underlying data type is a Set
  "points": [30, 35] # underlying data type is a List
  "_version" : 5
}
```

Ici, nous observons que le champ entrant pour « intérêts » a une valeur en double qui existe dans le serveur et deux nouvelles valeurs. Dans ce cas, puisque le type de données sous-jacent est un ensemble, Automerge combinera les valeurs existantes dans le serveur avec celles de la requête entrante et supprimera les doublons. De même, il y a un conflit sur le champ « points » où il y a une valeur en double et une nouvelle valeur. Mais puisque le type de données sous-jacent ici est une liste, Automerge ajoutera simplement toutes les valeurs de la requête entrante à la fin des valeurs déjà existantes dans le serveur. L'image fusionnée résultante stockée sur le serveur apparaîtrait comme suit :

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a
 Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "_version" : 7
}
```

Supposons maintenant que l'élément stocké dans le serveur apparaît comme suit, à la version 8.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a
 Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "stats": {
      "ppg": "35.4",
      "apg": "6.3"
  }
  "_version" : 8
}
```

Mais une demande entrante essaie de mettre à jour l'élément avec l'image suivante, encore une fois avec une incompatibilité de version :

```
{
  "id" : 1,
  "name" : "Nadia",
  "stats": {
      "ppg": "25.7",
      "rpg": "6.9"
  }
  "_version" : 3
}
```

Maintenant, dans ce scénario, nous pouvons voir que les champs qui existent déjà dans le serveur sont manquants (intérêts, points, jersey). En outre, la valeur de « ppg » dans la carte « stats » est en cours d'édition, une nouvelle valeur « rpg » est ajoutée et « apg » est omise. Automerge conserve les champs qui ont été omis (remarque : si les champs sont destinés à être supprimés, la demande doit être réessayée avec la version correspondante), ce qui signifie qu'ils ne seront pas perdus. Il appliquera également les mêmes règles aux champs dans les cartes et donc le changement de « ppg » sera rejeté alors que « apg » est conservé et « rpg », un nouveau champ », est ajouté. L'élément résultant stocké dans le serveur apparaîtra désormais sous la forme suivante :

```
{
  "id" : 1,
  "name" : "Nadia",
```

```
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a
 Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "stats": {
      "ppg": "35.4",
      "apg": "6.3",
      "rpg": "6.9"
  }
  "_version" : 9
}
```

Lambda

Options de résolution des conflits :

- `RESOLVE`: remplacez l'article existant par un nouvel article fourni dans la charge utile de réponse. Vous ne pouvez réessayer la même opération que sur un seul élément à la fois. Actuellement pris en charge pour DynamoDB `PutItem` et `UpdateItem`.
- `REJECT`: Rejette la mutation et renvoie une erreur avec l'élément existant dans la réponse GraphQL. Actuellement pris en charge pour DynamoDB `PutItem`, `UpdateItem` & `DeleteItem`.
- `REMOVE` : Supprime l'élément existant. Actuellement pris en charge pour DynamoDB `DeleteItem`.

Requête d'appel Lambda

Le résolveurAWS AppSync DynamoDB appelle la fonction Lambda spécifiée dans le`LambdaConflictHandlerArn`. Il utilise le même `service-role-arn` que celui configuré sur la source de données. La charge utile de l'appel a la structure suivante :

```
{
    "newItem": { ... },
    "existingItem": {... },
    "arguments": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

Les champs sont définis comme suit :

`newItem`

L'élément d'aperçu, si la mutation a réussi.

`existingItem`

L'élément réside actuellement dans la table DynamoDB.

`arguments`

Arguments de la mutation GraphQL.

`resolver`

Informations sur le résolveur AWS AppSync.

`identity`

Informations sur l'appelant. Ce champ est défini sur null, s'il s'agit d'un accès avec la clé API.

Exemple de charge utile :

```
{
    "newItem": {
        "id": "1",
        "author": "Jeff",
        "title": "Foo Bar",
        "rating": 5,
        "comments": ["hello world"],
    },
    "existingItem": {
        "id": "1",
        "author": "Foo",
        "rating": 5,
        "comments": ["old comment"]
    },
    "arguments": {
        "id": "1",
        "author": "Jeff",
        "title": "Foo Bar",
        "comments": ["hello world"]
    },
    "resolver": {
        "tableName": "post-table",
```

```
        "awsRegion": "us-west-2",
        "parentType": "Mutation",
        "field": "updatePost"
    },
    "identity": {
         "accountId": "123456789012",
         "sourceIp": "x.x.x.x",
         "username": "AIDAAAAAAAAAAAAAAAAA",
         "userArn": "arn:aws:iam::123456789012:user/appsync"
    }
}
```

Réponse à l'appel de Lambda

Pour `PutItem` et la résolution des conflits `UpdateItem`

`RESOLVE` la mutation. La réponse doit être au format suivant.

```
{
    "action": "RESOLVE",
    "item": { ... }
}
```

Le champ `item` représente un objet qui sera utilisé pour remplacer l'élément existant dans la source de données sous-jacente. La clé primaire et les métadonnées de synchronisation seront ignorées si elles sont incluses dans `item`.

`REJECT` la mutation. La réponse doit être au format suivant.

```
{
    "action": "REJECT"
}
```

Résolution des conflits `DeleteItem`

`REMOVE` l'élément. La réponse doit être au format suivant.

```
{
    "action": "REMOVE"
}
```

`REJECT` la mutation. La réponse doit être au format suivant.

```
{
    "action": "REJECT"
}
```

L'exemple de fonction Lambda ci-dessous vérifie qui effectue l'appel et le nom du résolveur. S'il est créé par`jeffTheAdmin`,`REMOVE` l'objet pour le DeletePost résolveur ou`RESOLVE` le conflit avec un nouvel élément pour les résolveurs Update/Put. Sinon, la mutation est `REJECT`.

```
exports.handler = async (event, context, callback) => {
    console.log("Event: "+ JSON.stringify(event));

    // Business logic goes here.
    var response;
    if ( event.identity.user == "jeffTheAdmin" ) {
        let resolver = event.resolver.field;

        switch(resolver) {
            case "deletePost":
                response = {
                    "action" : "REMOVE"
                }
                break;

            case "updatePost":
            case "createPost":
                response = {
                    "action" : "RESOLVE",
                    "item": event.newItem
                }
                break;
            default:
                response = { "action" : "REJECT" };
        }
    } else {
        response = { "action" : "REJECT" };
    }

    console.log("Response: "+ JSON.stringify(response));
    return response;
}
```

Erreurs

`ConflictUnhandled`

La détection de conflits détecte une incompatibilité de version et le gestionnaire de conflits rejette la mutation.

Exemple : résolution de conflits avec un gestionnaire de conflits Simultanéité optimiste. Ou, le gestionnaire de conflits Lambda retourné avec `REJECT`.

`ConflictError`

Une erreur interne se produit lors de la tentative de résolution d'un conflit.

Exemple : le gestionnaire de conflits Lambda a renvoyé une réponse mal formée.
Ou, ne peut pas appeler le gestionnaire de conflits Lambda car la ressource fournie `LambdaConflictHandlerArn` est introuvable.

`MaxConflicts`

Le nombre maximal de tentatives de résolution de conflit a été atteint.

Exemple : Trop de demandes simultanées sur le même objet. Avant que le conflit soit résolu, l'objet est mis à jour vers une nouvelle version par un autre client.

`BadRequest`

Le client tente de mettre à jour les champs de métadonnées (`_version`, `_ttl`, `_lastChangedAt`, `_deleted`).

Exemple : le client tente de mettre à jour _version d'un objet avec une mutation de mise à jour.

`DeltaSyncWriteError`

Échec de l'écriture de l'enregistrement de synchronisation delta.

Exemple : la mutation a réussi, mais une erreur interne s'est produite lors de la tentative d'écriture dans la table de synchronisation delta.

`InternalFailure`

Une erreur interne s'est produite.

CloudWatch Journaux

Si uneAWS AppSync API a activé les CloudWatch journaux avec les paramètres de journalisation définis sur Journaux au niveau du champ`enabled` et sur le niveau de journalisation pour les journaux au niveau du champ`ALL`, elleAWS AppSync enverra des informations de détection et de résolution des conflits au groupe de journaux. Pour de plus amples informations sur le format des messages du journal, veuillez consulter la documentation relative à la détection des conflits et à la journalisation de la synchronisation.

Opérations de synchronisation

Les sources de données versionnées prennent en charge les`Sync` opérations qui vous permettent de récupérer tous les résultats d'une table DynamoDB, puis de ne recevoir que les données modifiées depuis votre dernière requête (les mises à jour delta). Lorsque AWS AppSync reçoit une demande pour une opération `Sync`, il utilise les champs spécifiés dans la demande pour déterminer si la table Base ou la table Delta doit être accessible.

- Si le champ `lastSync` n'est pas spécifié, une `Scan` sur la table Base est exécutée.
- Si le champ `lastSync` est spécifié, mais que la valeur est antérieure à `current moment - DeltaSyncTTL`, un `Scan` sur la table Base est effectué.
- Si le champ `lastSync` est spécifié et que la valeur est sur ou après `current moment - DeltaSyncTTL`, un `Query` sur la table Delta est effectué.

AWS AppSync renvoie le`startedAt` champ au modèle de mappage des réponses pour toutes les`Sync` opérations. Le champ `startedAt` est le moment, en millisecondes Epoch, où l'opération `Sync` a commencé et où vous pouvez la stocker localement et l'utiliser dans une autre requête. Si un jeton de pagination a été inclus dans la requête, cette valeur sera la même que celle renvoyée par la requête pour la première page de résultats.

Pour de plus amples informations sur le format des modèles de mappage `Sync`, veuillez consultez la référence du modèle de mappage.

Surveillance et journalisation

Pour surveiller votre API AWS AppSync GraphQL et résoudre les problèmes liés aux requêtes, vous pouvez activer la connexion à Amazon Logs. CloudWatch

Installation et configuration

Pour activer la journalisation automatique sur une API GraphQL, utilisez la AWS AppSync console.

1. Connectez-vous à la AppSyncconsole AWS Management Console et ouvrez-la.
2. Sur la page API, choisissez le nom d'une API GraphQL.
3. Sur la page d'accueil de votre API, dans le volet de navigation, sélectionnez Paramètres.
4. Sous Journalisation, procédez de la façon suivante :
   a. Activez Activer les journaux.
   b. Pour une journalisation détaillée au niveau de la demande, cochez la case sous Inclure le contenu détaillé. (facultatif)
   c. Sous Niveau de journalisation du résolveur de champs, choisissez votre niveau de journalisation préféré au niveau du champ (aucun, erreur ou tout). (facultatif)
   d. Sous Créer ou utiliser un rôle existant, choisissez Nouveau rôle pour créer un nouveau rôle AWS Identity and Access Management (IAM) qui permet d' AWS AppSync écrire CloudWatch des journaux. Vous pouvez également choisir Rôle existant pour sélectionner le nom de ressource Amazon (ARN) d'un rôle IAM existant dans votre AWS compte.
5. Choisissez Enregistrer.

Configuration manuelle des rôles IAM

Si vous choisissez d'utiliser un rôle IAM existant, celui-ci doit accorder AWS AppSync les autorisations requises pour écrire des journaux. CloudWatch Pour le configurer manuellement, vous devez fournir un ARN de rôle de service afin de AWS AppSync pouvoir assumer le rôle lors de l'écriture des journaux.

Dans la console IAM, créez une nouvelle politique dont le nom `AWSAppSyncPushToCloudWatchLogsPolicy` a la définition suivante :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
```

```
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

Créez ensuite un nouveau rôle portant ce nom AWSAppSyncPushToCloudWatchLogsRoleet associez la politique nouvellement créée au rôle. Modifiez la relation de confiance pour ce rôle comme suit :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Effect": "Allow",
        "Principal": {
            "Service": "appsync.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
        }
    ]
}
```

Copiez l'ARN du rôle et utilisez-le lors de la configuration de la journalisation pour une API AWS AppSync GraphQL.

CloudWatch métriques

Vous pouvez utiliser CloudWatch des métriques pour surveiller et émettre des alertes concernant des événements spécifiques susceptibles d'entraîner des codes d'état HTTP ou une latence. Les métriques suivantes sont émises :

Liste des métriques

`4XXError`

Erreurs résultant de demandes non valides en raison d'une configuration client incorrecte. Généralement, ces erreurs se produisent n'importe où en dehors du traitement GraphQL. Par exemple, ces erreurs peuvent se produire lorsque la demande inclut une charge utile JSON

incorrecte ou une requête incorrecte, lorsque le service est limité ou lorsque les paramètres d'autorisation sont mal configurés.

Unité : nombre. Utilisez la statistique Somme pour obtenir le total des occurrences de ces erreurs.

`5XXError`

Erreurs rencontrées lors de l'exécution d'une requête GraphQL. Cela peut se produire, par exemple, lors de l'appel d'une requête pour un schéma vide ou incorrect. Cela peut également se produire lorsque l'ID ou la AWS région du groupe d'utilisateurs Amazon Cognito n'est pas valide. Cela peut également se produire en cas AWS AppSync de problème lors du traitement d'une demande.

Unité : nombre. Utilisez la statistique Somme pour obtenir le total des occurrences de ces erreurs.

`Latency`

Le délai entre le moment où il AWS AppSync reçoit une demande d'un client et le moment où il renvoie une réponse au client. Cela n'inclut pas la latence du réseau rencontrée pour une réponse afin d'atteindre les appareils finaux.

Unité : milliseconde. Utilisez la statistique Moyenne pour évaluer les latences attendues.

`Requests`

Le nombre de demandes (requêtes + mutations) traitées par toutes les API de votre compte, par région.

Unité : nombre. Le nombre de demandes traitées dans une région donnée.

`TokensConsumed`

Les jetons sont alloués en `Requests` fonction de la quantité de ressources (temps de traitement et mémoire utilisée) qu'un utilisateur `Request` consomme. En général, chacun `Request` consomme un jeton. Toutefois, ceux `Request` qui consomment de grandes quantités de ressources se voient attribuer des jetons supplémentaires selon les besoins.

Unité : nombre. Le nombre de jetons alloués aux demandes traitées dans une région donnée.

`NetworkBandwidthOutAllowanceExceeded`

> **Note**
>
> Dans la AWS AppSync console, sur la page des paramètres du cache, l'option Cache Health Metrics vous permet d'activer cette métrique de santé liée au cache.

Les paquets réseau ont été abandonnés car le débit dépassait la limite de bande passante agrégée. Cela est utile pour diagnostiquer les goulots d'étranglement dans une configuration de cache. Les données sont enregistrées pour une API particulière en les spécifiant `API_Id` dans la `appsyncCacheNetworkBandwidthOutAllowanceExceeded` métrique.

Unité : nombre. Nombre de paquets abandonnés après avoir dépassé la limite de bande passante pour une API spécifiée par ID.

`EngineCPUUtilization`

> **Note**
>
> Dans la AWS AppSync console, sur la page des paramètres du cache, l'option Cache Health Metrics vous permet d'activer cette métrique de santé liée au cache.

L'utilisation du processeur (pourcentage) allouée au processus Redis. Cela est utile pour diagnostiquer les goulots d'étranglement dans une configuration de cache. Les données sont enregistrées pour une API particulière en les spécifiant `API_Id` dans la `appsyncCacheEngineCPUUtilization` métrique.

Unité : Pourcentage. Pourcentage de processeur actuellement utilisé par le processus Redis pour une API spécifiée par ID.

Abonnements en temps réel

Toutes les métriques sont émises dans une dimension : GraphQLAPIID. Cela signifie que toutes les métriques sont couplées avec des ID d'API GraphQL. Les métriques suivantes sont liées aux abonnements GraphQL sur Pure : WebSockets

Liste des métriques

`ConnectRequests`

Le nombre de demandes de WebSocket connexion adressées à AWS AppSync, y compris les tentatives réussies et infructueuses.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total de demandes de connexion.

`ConnectSuccess`

Le nombre de WebSocket connexions réussies à AWS AppSync. Il est possible d'avoir des connexions sans abonnements.

Unité : nombre. Utilisez la statistique Sum (Somme) pour obtenir le total des occurrences de connexions réussies.

`ConnectClientError`

Le nombre de WebSocket connexions rejetées en AWS AppSync raison d'erreurs côté client. Cela peut indiquer que le service est limité ou que les paramètres d'autorisation sont mal configurés.

Unité : nombre. Utilisez la statistique Sum (Somme) pour obtenir le total des occurrences des erreurs de connexion côté client.

`ConnectServerError`

Nombre d'erreurs survenues AWS AppSync lors du traitement des connexions. Cela se produit généralement lorsqu'un problème inattendu prend place côté serveur.

Unité : nombre. Utilisez la statistique Sum (Somme) pour obtenir le total des occurrences d'erreurs de connexion côté serveur.

`DisconnectSuccess`

Le nombre de WebSocket déconnexions réussies de AWS AppSync.

Unité : nombre. Utilisez la statistique Sum (Somme) pour obtenir le total d'occurrences de déconnexions réussies.

`DisconnectClientError`

Nombre d'erreurs client survenues AWS AppSync lors de la déconnexion des WebSocket connexions.

Unité : nombre. Utilisez la statistique Sum (Somme) pour obtenir le total des occurrences d'erreurs de connexion.

`DisconnectServerError`

Nombre d'erreurs de serveur survenues AWS AppSync lors de la déconnexion WebSocket des connexions.

Unité : nombre. Utilisez la statistique Sum (Somme) pour obtenir le total des occurrences d'erreurs de connexion.

`SubscribeSuccess`

Le nombre d'abonnements enregistrés avec succès AWS AppSync via WebSocket. Il est possible d'avoir des connexions sans abonnement, mais il n'est pas possible d'avoir des abonnements sans connexions.

Unité : nombre. Utilisez la statistique Sum (Somme) pour obtenir le nombre total des occurrences d'abonnements réussis.

`SubscribeClientError`

Le nombre d'abonnements rejetés en AWS AppSync raison d'erreurs côté client. Cela peut se produire lorsqu'une charge utile JSON est incorrecte, que le service est limité ou que les paramètres d'autorisation sont mal configurés.

Unité : nombre. Utilisez la statistique Sum (Somme) pour obtenir le total des occurrences d'erreurs d'abonnement côté client.

`SubscribeServerError`

Nombre d'erreurs survenues AWS AppSync lors du traitement des abonnements. Cela se produit généralement lorsqu'un problème inattendu prend place côté serveur.

Unité : nombre. Utilisez la statistique Sum (Somme) pour obtenir le nombre total d'occurrences des erreurs d'abonnement côté serveur.

`UnsubscribeSuccess`

Le nombre de demandes de désabonnement traitées avec succès.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total de demandes de désabonnement réussies.

`UnsubscribeClientError`

Le nombre de demandes de désabonnement rejetées en AWS AppSync raison d'erreurs côté client.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total d'occurrences des erreurs de demande de désabonnement côté client.

`UnsubscribeServerError`

Nombre d'erreurs survenues AWS AppSync lors du traitement des demandes de désabonnement. Cela se produit généralement lorsqu'un problème inattendu prend place côté serveur.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total d'occurrences des erreurs de demande de désabonnement côté serveur.

`PublishDataMessageSuccess`

Nombre de messages d'événement d'abonnement qui ont été publiés avec succès.

Unité : nombre. Utilisez la statistique Sum (Somme) pour obtenir le total des messages d'événement d'abonnement qui ont été publiés avec succès.

`PublishDataMessageClientError`

Nombre de messages d'événement d'abonnement qui n'ont pas pu être publiés en raison d'erreurs côté client.

`Unit`: Compte. Utilisez la statistique Sum (Somme) pour obtenir le total des occurrences d'erreurs d'événement d'abonnement de publication côté client.

`PublishDataMessageServerError`

Nombre d'erreurs survenues AWS AppSync lors de la publication des messages d'événements d'abonnement. Cela se produit généralement lorsqu'un problème inattendu prend place côté serveur.

Unité : nombre. Utilisez la statistique Sum (Somme) pour obtenir le total des occurrences d'erreurs d'événement d'abonnement de publication côté serveur.

`PublishDataMessageSize`

Taille des messages d'événement d'abonnement publiés.

Unité : octets.

`ActiveConnections`

Le nombre de WebSocket connexions simultanées entre les clients et AWS AppSync en 1 minute.

Unité : nombre. Utilisez la statistique Sum (Somme) pour obtenir le nombre total de connexions ouvertes.

`ActiveSubscriptions`

Nombre d'abonnements simultanés provenant de clients en 1 minute.

Unité : nombre. Utilisez la statistique Sum (Somme) pour obtenir le total des abonnements actifs.

`ConnectionDuration`

Durée pendant laquelle la connexion reste ouverte.

Unité : millisecondes. Utilisez la statistique Average (Moyenne) pour évaluer la durée de connexion.

`OutboundMessages`

Le nombre de messages mesurés publiés avec succès. Un message mesuré équivaut à 5 kB de données livrées.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total de messages mesurés publiés avec succès.

`InboundMessageSuccess`

Nombre de messages entrants traités avec succès. Chaque type d'abonnement invoqué par une mutation génère un message entrant.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total de messages entrants traités avec succès.

`InboundMessageError`

Le nombre de messages entrants dont le traitement a échoué en raison de demandes d'API non valides, telles que le dépassement de la limite de charge utile de l'abonnement de 240 kB.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total de messages entrants présentant des échecs de traitement liés à l'API.

InboundMessageFailure

Nombre de messages entrants dont le traitement a échoué en raison d'erreurs provenant de AWS AppSync.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total de messages entrants présentant des échecs de traitement AWS AppSync associés.

InboundMessageDelayed

Le nombre de messages entrants retardés. Les messages entrants peuvent être retardés lorsque le quota de débit de messages entrants ou le quota de débit de messages sortants est dépassé.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total de messages entrants qui ont été retardés.

InboundMessageDropped

Le nombre de messages entrants abandonnés. Les messages entrants peuvent être supprimés lorsque le quota de débit de messages entrants ou le quota de débit de messages sortants est dépassé.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total de messages entrants qui ont été supprimés.

InvalidationSuccess

Le nombre d'abonnements invalidés (désabonnés) avec succès par une mutation avec. `$extensions.invalidateSubscriptions()`

Unité : nombre. Utilisez la statistique Sum pour récupérer le nombre total d'abonnements désabonnés avec succès.

InvalidationRequestSuccess

Nombre de demandes d'invalidation traitées avec succès.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total de demandes d'invalidation traitées avec succès.

InvalidationRequestError

Le nombre de demandes d'invalidation dont le traitement a échoué en raison de demandes d'API non valides.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total de demandes d'invalidation présentant des échecs de traitement liés à l'API.

`InvalidationRequestFailure`

Le nombre de demandes d'invalidation dont le traitement a échoué en raison d'erreurs provenant de AWS AppSync.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total de demandes d'invalidation associées à des échecs de AWS AppSync traitement.

`InvalidationRequestDropped`

Le nombre de demandes d'invalidation abandonnées lorsque le quota de demandes d'invalidation a été dépassé.

Unité : nombre. Utilisez la statistique Sum pour obtenir le nombre total de demandes d'invalidation abandonnées.

Comparaison des messages entrants et sortants

Lorsqu'une mutation est exécutée, les champs d'abonnement contenant la directive @aws_subscribe pour cette mutation sont invoqués. Chaque appel d'abonnement génère un message entrant. Par exemple, si deux champs d'abonnement spécifient la même mutation dans @aws_subscribe, deux messages entrants sont générés lorsque cette mutation est appelée.

Un message sortant équivaut à 5 kB de données livrées aux WebSocket clients. Par exemple, l'envoi de 15 Ko de données à 10 clients génère 30 messages sortants (15 kB* 10 clients/5 kB par message = 30 messages).

Vous pouvez demander des augmentations de quota pour les messages entrants ou sortants. Pour plus d'informations, consultez les AWS AppSync points de terminaison et les quotas dans le guide de référence AWS général et les instructions pour demander une augmentation de quota dans le guide de l'utilisateur du Service Quotas.

Métriques améliorées

Les métriques améliorées émettent des données granulaires sur l'utilisation et les performances des API, telles que le nombre de AWS AppSync demandes et d'erreurs, la latence et les accès/échecs du cache. Toutes les données métriques améliorées sont envoyées à votre CloudWatch compte, et vous pouvez configurer les types de données qui seront envoyées.

> **Note**
>
> Des frais supplémentaires sont appliqués lors de l'utilisation de métriques améliorées. Pour plus d'informations, consultez les niveaux de tarification détaillés de la surveillance dans la section CloudWatchTarification d'Amazon.

Ces statistiques se trouvent sur les différentes pages de paramètres de la AWS AppSync console. Sur la page des paramètres de l'API, la section Mesures améliorées vous permet d'activer ou de désactiver les éléments suivants :

1. Comportement des métriques du résolveur : ces options contrôlent la manière dont les métriques supplémentaires pour les résolveurs sont collectées. Vous pouvez choisir d'activer les métriques complètes du résolveur de demandes (métriques activées pour tous les résolveurs dans les demandes) ou les métriques par résolveur (métriques activées uniquement pour les résolveurs dont la configuration est définie sur activée). Les options suivantes sont disponibles :

Métrique	Dimension métrique	Nom de la métrique	Unité	Description
Erreurs GraphQL par résolveur	API_ID, résolveur	Erreur GraphQL	Count	Nombre d'erreurs GraphQL survenues par résolveur.
Demandes par résolveur	API_ID, résolveur	Demande	Count	Le nombre d'invocations survenues lors d'une demande. Ceci est enregistré par résolveur.
Latence par résolveur	API_ID, résolveur	Latence	Milliseconde	Le temps nécessaire pour terminer une invocatio

				n du résolveur . La latence est mesurée en millisecondes et est enregistrée par résolveur.
Nombre de visites en cache par résolveur	API_ID, résolveur	CacheHit	Count	Le nombre de connexions au cache lors d'une demande. Cela ne sera émis que si un cache est utilisé. Les accès au cache sont enregistrés par résolveur.
Erreurs de cache par résolveur	API_ID, résolveur	CacheMiss	Count	Le nombre de caches manquants lors d'une demande. Cela ne sera émis que si un cache est utilisé. Les erreurs de cache sont enregistrées par résolveur.

2. Comportement des métriques des sources de données : ces options contrôlent la manière dont les métriques supplémentaires pour les sources de données sont collectées. Vous pouvez choisir d'activer les métriques complètes des sources de données des demandes (métriques activées pour toutes les sources de données dans les demandes) ou les métriques par source de données (métriques activées uniquement pour les sources de données dont la configuration est définie comme activée). Les options suivantes sont disponibles :

Métrique	Dimension métrique	Nom de la métrique	Unité	Description
Demandes par source de données	API_ID, source de données	Demande	Count	Le nombre d'invocations survenues lors d'une demande. Les demandes sont enregistrées par source de données. Si les demandes complètes sont activées, chaque source de données aura sa propre entrée CloudWatch.
Latence par source de données	API_ID, source de données	Latence	Milliseconde	Le temps nécessaire pour terminer un appel de source de données. La latence est enregistrée par source de données.
Erreurs par source de données	API_ID, source de données	Erreur GraphQL	Count	Nombre d'erreurs survenues lors d'un appel de source de données.

3. Métriques d'opération : active les métriques au niveau des opérations GraphQL.

Métrique	Dimension métrique	Nom de la métrique	Unité	Description
Demandes par opération	API_ID, Opération	Demande	Count	Nombre de fois qu'une opération GraphQL spécifiée a été appelée.
Erreurs GraphQL par opération	API_ID, Opération	Erreur GraphQL	Count	Nombre d'erreurs GraphQL survenues lors d'une opération GraphQL spécifiée.

CloudWatch journaux

Vous pouvez configurer deux types de journalisation sur n'importe quelle API GraphQL nouvelle ou existante : au niveau de la demande et au niveau du champ.

Journaux au niveau des demandes

Lorsque la journalisation au niveau de la demande (inclure du contenu détaillé) est configurée, les informations suivantes sont enregistrées :

- Le nombre de jetons consommés
- La demande et la réponse des en-têtes HTTP
- La requête GraphQL exécutée dans la requête
- Le résumé général des opérations
- Les abonnements à GraphQL nouveaux et anciens qui sont enregistrés

Journaux au niveau du terrain

Lorsque la journalisation au niveau des champs est configurée, les informations suivantes sont enregistrées :

- Mappage des demandes généré avec la source et les arguments pour chaque champ
- Le mappage de réponse transformé pour chaque champ, qui inclut les données résultant de la résolution de ce champ
- Suivi des informations pour chaque champ

Si vous activez la journalisation, AWS AppSync gère les CloudWatch journaux. Le processus comprend la création de groupes de journaux et de flux de journaux, et la génération de rapports dans les flux de journaux avec ces journaux.

Lorsque vous activez la journalisation sur une API GraphQL et que vous envoyez des requêtes, vous AWS AppSync créez un groupe de journaux et enregistrez des flux sous le groupe de journaux. Le groupe de journaux est nommé selon le format `/aws/appsync/apis/{graphql_api_id}`. Dans chaque groupe de journaux, les journaux sont également divisés en flux de journaux. Ces derniers sont classés selon l'heure du dernier événement lors de la consignation des données.

Chaque événement du journal est marqué avec le x-amzn- RequestId de cette demande. Cela vous permet de filtrer les événements de connexion CloudWatch afin d'obtenir toutes les informations enregistrées concernant cette demande. Vous pouvez l'obtenir RequestId à partir des en-têtes de réponse de chaque requête AWS AppSync GraphQL.

La journalisation au niveau du champ est configurée avec les niveaux de journaux suivants :

- Aucun : aucun journal au niveau du champ n'est capturé.
- Erreur : enregistre les informations suivantes uniquement pour les champs erronés :
    - Section d'erreur dans la réponse du serveur
    - Erreurs au niveau du champ
    - Fonctions de demande/réponse générées qui ont été résolues pour les champs d'erreur
- Tout : enregistre les informations suivantes pour tous les champs de la requête :
    - Informations de suivi au niveau du champ
    - Fonctions de demande/réponse générées qui ont été résolues pour chaque champ

Avantages de la surveillance

Vous pouvez utiliser la journalisation et les métriques pour identifier, dépanner et optimiser vos requêtes GraphQL. Par exemple, cela vous aidera à déboguer les problèmes de latence à l'aide des informations de suivi consignées pour chaque champ de la requête. Pour illustrer cela, supposons que vous utilisiez un ou plusieurs résolveurs imbriqués dans une requête GraphQL. Un exemple d'opération sur le terrain dans CloudWatch Logs peut ressembler à ce qui suit :

```
{
    "path": [
        "singlePost",
        "authors",
        0,
        "name"
    ],
    "parentType": "Post",
    "returnType": "String!",
    "fieldName": "name",
    "startOffset": 416563350,
    "duration": 11247
}
```

Cela peut correspondre à un schéma GraphQL, comme illustré ci-dessous :

```
type Post {
  id: ID!
  name: String!
  authors: [Author]
}

type Author {
  id: ID!
  name: String!
}

type Query {
  singlePost(id:ID!): Post
}
```

Dans les résultats du journal précédents, le chemin indique un seul élément de vos données renvoyées par l'exécution d'une requête nommée`singlePost()`. Dans cet exemple, il représente

le champ de nom du premier index (0). Le StartOffset donne un décalage par rapport au début de l'opération de requête GraphQL. La durée est le temps total de résolution du champ. Ces valeurs peuvent être utiles afin de déterminer la raison pour laquelle les données d'une source de données spécifique sont exécutées plus lentement que prévu, ou si un champ spécifique ralentit l'ensemble de la requête. Par exemple, vous pouvez choisir d'augmenter le débit provisionné pour une table Amazon DynamoDB ou de supprimer un champ spécifique d'une requête qui nuit aux performances globales de l'opération.

Depuis le 8 mai 2019, AWS AppSync génère des événements de journal sous forme de JSON entièrement structuré. Cela peut vous aider à utiliser les services d'analyse des CloudWatch journaux tels que Logs Insights et Amazon OpenSearch Service pour comprendre les performances de vos requêtes GraphQL et les caractéristiques d'utilisation de vos champs de schéma. Par exemple, vous pouvez identifier facilement les résolveurs rencontrant des latences importantes et qui pourraient être la cause profonde d'un problème de performances. Vous pouvez également identifier les champs utilisés le plus et le moins fréquemment dans votre schéma et évaluer l'impact des dépréciations des champs GraphQL.

Détection des conflits et journalisation des synchronisations

Si une AWS AppSync API a configuré la journalisation dans CloudWatch les journaux avec le niveau de journalisation du résolveur de champs défini sur Tous, les informations relatives à AWS AppSync la détection et à la résolution des conflits sont transmises au groupe de journaux. Cela fournit un aperçu détaillé de la manière dont l' AWS AppSync API a répondu à un conflit. Pour vous aider à interpréter la réponse, les informations suivantes sont fournies dans les journaux :

Liste des métriques

`conflictType`

Détermine si un conflit s'est produit en raison d'une incompatibilité de version ou de la condition fournie par le client.

`conflictHandlerConfigured`

État le gestionnaire de conflits configuré sur le résolveur au moment de la demande.

`message`

Fournit des informations sur la façon dont le conflit a été détecté et résolu.

`syncAttempt`

Nombre d'essais du serveur pour synchroniser les données avant de rejeter la demande.

`data`

Si le gestionnaire de conflits configuré l'est`Automerge`, ce champ est renseigné pour indiquer la décision `Automerge` prise pour chaque champ. Les actions proposées peuvent être les suivantes :

- REJETÉ - `Automerge` Lorsque la valeur du champ entrant est rejetée en faveur de la valeur du serveur.
- AJOUTÉ - Lors de l'`Automerge`ajout du champ entrant en raison de l'absence de valeur préexistante sur le serveur.
- AJOUTÉ - Quand `Automerge` ajoute les valeurs entrantes aux valeurs de la liste qui existe sur le serveur.
- **`Automerge`**MERGED - Lorsque fusionne les valeurs entrantes avec les valeurs de l'ensemble existant sur le serveur.

Utiliser le nombre de jetons pour optimiser vos demandes

Un jeton est alloué aux requêtes consommant moins de 1 500 Ko de secondes de mémoire et de temps de vCPU. Les demandes dont la consommation de ressources est supérieure à 1 500 kB-secondes reçoivent des jetons supplémentaires. Par exemple, si une demande consomme 3 350 Ko de secondes, AWS AppSync alloue trois jetons (arrondis à la valeur entière suivante) à la demande. Par défaut, AWS AppSync alloue un maximum de 5 000 ou 10 000 jetons de demande par seconde aux API de votre compte, en fonction de la AWS région dans laquelle elle est déployée. Si vos API utilisent chacune en moyenne deux jetons par seconde, vous serez limité à 2 500 ou 5 000 requêtes par seconde, respectivement. Si vous avez besoin de plus de jetons par seconde que le montant alloué, vous pouvez soumettre une demande pour augmenter le quota par défaut pour le taux de jetons de demande. Pour plus d'informations, consultez les rubriques AWS AppSync Points de terminaison et quotas dans le Références générales AWS guide et Demande d'augmentation des quotas dans le Guide de l'utilisateur du Service Quotas.

Un nombre élevé de jetons par demande peut indiquer qu'il est possible d'optimiser vos demandes et d'améliorer les performances de votre API. Les facteurs susceptibles d'augmenter le nombre de jetons par demande sont les suivants :

- La taille et la complexité de votre schéma GraphQL.
- La complexité des modèles de mappage des demandes et des réponses.
- Le nombre d'appels au résolveur par demande.

- La quantité de données renvoyées par les résolveurs.
- La latence des sources de données en aval.
- Modèles de schéma et de requête qui nécessitent des appels successifs à la source de données (par opposition aux appels en parallèle ou par lots).
- Configuration de journalisation, en particulier au niveau du champ et contenu détaillé des journaux.

> **Note**
>
> Outre les AWS AppSync métriques et les journaux, les clients peuvent accéder au nombre de jetons consommés dans une demande via l'en-tête de réponse`x-amzn-appsync-TokensConsumed`.

Référence du type de journal

RequestSummary

- requestId : identifiant unique de la demande.
- graphQLAPIId : ID de l'API GraphQL effectuant la demande.
- StatusCode : réponse au code d'état HTTP.
- latence : nd-to-end latence E de la demande, en nanosecondes, sous forme de nombre entier.

```
{
     "logType": "RequestSummary",
     "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
     "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4",
     "statusCode": 200,
     "latency": 242000000
}
```

ExecutionSummary

- requestId : identifiant unique de la demande.
- graphQLAPIId : ID de l'API GraphQL effectuant la demande.
- StartTime : horodatage de début du traitement GraphQL de la demande, au format RFC 3339.

- EndTime : horodatage de fin du traitement GraphQL de la demande, au format RFC 3339.
- durée : le temps de traitement total écoulé par GraphQL, en nanosecondes, sous forme d'entier.
- version : version du schéma du ExecutionSummary.
- parsing :
    - StartOffset : le décalage de départ pour l'analyse, en nanosecondes, par rapport à l'invocation, sous la forme d'un entier.
    - duration : temps consacré à l'analyse, en nanosecondes, indiqué en tant que nombre entier.
- validation :
    - StartOffset : le décalage de départ pour la validation, en nanosecondes, par rapport à l'invocation, sous forme d'entier.
    - duration : temps consacré à la validation, en nanosecondes, indiqué en tant que nombre entier.

```
{
    "duration": 217406145,
    "logType": "ExecutionSummary",
    "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
    "startTime": "2019-01-01T06:06:18.956Z",
    "endTime": "2019-01-01T06:06:19.174Z",
    "parsing": {
        "startOffset": 49033,
        "duration": 34784
    },
    "version": 1,
    "validation": {
        "startOffset": 129048,
        "duration": 69126
    },
    "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4"
}
```

Tracing

- requestId : identifiant unique de la demande.
- graphQLAPIId : ID de l'API GraphQL effectuant la demande.
- StartOffset : décalage de départ pour la résolution du champ, en nanosecondes, par rapport à l'invocation, sous forme d'entier.

- duration : temps consacré à la résolution du champ, en nanosecondes, indiqué en tant que nombre entier.
- fieldName : nom du champ en cours de résolution.
- parentType : type de parent du champ en cours de résolution.
- returnType : type de retour du champ en cours de résolution.
- path : liste des segments de chemin, commençant à la racine de la réponse et se terminant par le champ en cours de résolution.
- resolverArn : ARN du résolveur utilisé pour la résolution des champs. Peut ne pas être présent sur les champs imbriqués.

```
{
    "duration": 216820346,
    "logType": "Tracing",
    "path": [
        "putItem"
    ],
    "fieldName": "putItem",
    "startOffset": 178156,
    "resolverArn": "arn:aws:appsync:us-east-1:111111111111:apis/
pmo28inf75eepg63qxq4ekoeg4/types/Mutation/fields/putItem",
    "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
    "parentType": "Mutation",
    "returnType": "Item",
    "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4"
}
```

Analyser vos journaux avec CloudWatch Logs Insights

Voici des exemples de requêtes que vous pouvez exécuter pour obtenir des informations exploitables sur les performances et l'état de vos opérations GraphQL. Ces exemples sont disponibles sous forme d'exemples de requêtes dans la console CloudWatch Logs Insights. Dans la CloudWatchconsole, choisissez Logs Insights, sélectionnez le groupe de AWS AppSync journaux pour votre API GraphQL, puis choisissez des AWS AppSync requêtes sous Exemples de requêtes.

La requête suivante renvoie les 10 principales requêtes GraphQL avec un maximum de jetons consommés :

```
filter @message like "Tokens Consumed"
```

```
| parse @message "* Tokens Consumed: *" as requestId, tokens
| sort tokens desc
| display requestId, tokens
| limit 10
```

La requête suivante renvoie les 10 principaux résolveurs rencontrant une latence maximale :

```
  fields resolverArn, duration
| filter logType = "Tracing"
| limit 10
| sort duration desc
```

La requête suivante renvoie les résolveurs les plus fréquemment appelés :

```
  fields ispresent(resolverArn) as isRes
| stats count() as invocationCount by resolverArn
| filter isRes and logType = "Tracing"
| limit 10
| sort invocationCount desc
```

La requête suivante renvoie les résolveurs rencontrant le plus grand nombre d'erreurs dans les modèles de mappage :

```
  fields ispresent(resolverArn) as isRes
| stats count() as errorCount by resolverArn, logType
| filter isRes and (logType = "RequestMapping" or logType = "ResponseMapping") and
 fieldInError
| limit 10
| sort errorCount desc
```

La requête suivante renvoie les statistiques de latence du résolveur :

```
  fields ispresent(resolverArn) as isRes
| stats min(duration), max(duration), avg(duration) as avg_dur by resolverArn
| filter isRes and logType = "Tracing"
| limit 10
| sort avg_dur desc
```

La requête suivante renvoie les statistiques de latence du champ :

```
  stats min(duration), max(duration), avg(duration) as avg_dur
```

```
  by concat(parentType, '/', fieldName) as fieldKey
| filter logType = "Tracing"
| limit 10
| sort avg_dur desc
```

Les résultats des requêtes CloudWatch Logs Insights peuvent être exportés vers des CloudWatch tableaux de bord.

Analysez vos journaux avec OpenSearch Service

Vous pouvez rechercher, analyser et visualiser vos AWS AppSync journaux avec Amazon OpenSearch Service afin d'identifier les goulots d'étranglement liés aux performances et les causes profondes des problèmes opérationnels. Vous pouvez identifier les résolveurs rencontrant la latence maximale et le maximum d'erreurs. En outre, vous pouvez utiliser les OpenSearch tableaux de bord pour créer des tableaux de bord dotés de puissantes visualisations. OpenSearch Dashboards est un outil de visualisation et d'exploration de données open source disponible dans OpenSearch Service. À l'aide OpenSearch des tableaux de bord, vous pouvez surveiller en permanence les performances et l'état de vos opérations GraphQL. Par exemple, vous pouvez créer des tableaux de bord pour visualiser la latence P90 de vos requêtes GraphQL et analyser les latences P90 de chaque résolveur.

Lorsque vous utilisez OpenSearch Service, utilisez « cwl* » comme modèle de filtre pour rechercher OpenSearch des index. OpenSearch Le service indexe les journaux diffusés depuis CloudWatch Logs avec le préfixe « cwl - ». Pour différencier les journaux d' AWS AppSync API CloudWatch des autres journaux envoyés au OpenSearch Service, nous vous recommandons d'ajouter une expression de filtre supplémentaire `graphQLAPIID.keyword=`*`YourGraphQLAPIID`* à votre recherche.

Migration du format de journal

Les événements de journal AWS AppSync générés le 8 mai 2019 ou après cette date sont formatés au format JSON entièrement structuré. Pour analyser les requêtes GraphQL antérieures au 8 mai 2019, vous pouvez migrer les anciens journaux vers du JSON entièrement structuré à l'aide d'un script disponible dans l'GitHub exemple. Si vous avez besoin d'utiliser le format de journal antérieur au 8 mai 2019, créez un ticket de support avec les paramètres suivants : définissez Type sur Account Management (Gestion de compte), puis définissez Category (Catégorie) sur General Account Question (Question d'ordre général sur le compte).

Vous pouvez également utiliser des filtres métriques CloudWatch pour transformer les données du journal en CloudWatch métriques numériques, afin de pouvoir les représenter graphiquement ou de définir une alarme.

Suivi avecAWS X-Ray

Vous pouvez utiliserAWS X-Raysuivre les demandes au fur et à mesure qu'elles sont exécutées dansAWSAppSync. Vous pouvez utiliser X-Ray avecAWSAppSync dans tous lesAWSRégions où X-Ray est disponible. X-Ray vous donne une vue d'ensemble détaillée d'une demande GraphQL. Cela vous permet d'analyser les latences dans vos API et leurs résolveurs et sources de données sous-jacents. Vous pouvez utiliser une carte des services X-Ray pour afficher la latence d'une demande, y compris lesAWSservices intégrés à X-Ray. Vous pouvez également configurer des règles d'échantillonnage pour indiquer à X-Ray les demandes à enregistrer et les taux d'échantillonnage selon des critères que vous spécifiez.

Pour plus d'informations sur l'échantillonnage dans X-Ray, veuillez consulter la sectionConfiguration des règles d'échantillonnage dans leAWS X-RayConsole.

Installation et configuration

Vous pouvez activer le suivi X-Ray pour une API GraphQL via le moduleAWSconsole AppSync.

1. Connectez-vous à la consoleAWSconsole AppSync.
2. Choisissez Paramètres dans le volet de navigation.
3. Sous X-Ray, activez Enable X-Ray (Activer X-Ray).
4. Sélectionnez Enregistrer. Le suivi X-Ray est maintenant activé pour votre API.

Si vous utilisez le moduleAWS CLIouAWS CloudFormation, vous pouvez également activer le suivi des X-Ray lorsque vous créez un nouveauAWSAPI AppSync ou mise à jour d'une API existanteAWSAPI AppSync, en définissant le`xrayEnabled`propriété à`true`.

Lorsque le suivi X-Ray est activé pour unAWSAPI AppSync, uneAWS Identity and Access Management Rôle lié à un serviceest automatiquement créé dans votre compte avec les autorisations appropriées. Cela permet àAWSAppSync pour envoyer des suivis à X-Ray de manière sécurisée.

Suivi de votre API avec X-Ray

Echantillonnage

En utilisant les règles d'échantillonnage, vous pouvez contrôler la quantité de données que vous enregistrez dansAWSAppSync peut modifier le comportement d'échantillonnage à la volée sans modifier ni redéployer votre code. Par exemple, cette règle échantillonne les demandes à l'API GraphQL avec l'ID d'API `3n572shhcpfokwhdnq1ogu59v6`.

- Rule name (Nom de règle) : `test-sample`
- Priority (Priorité) : `10`
- Reservoir size (Taille du réservoir) : `10`
- Fixed rate (Fréquence fixe) : `10`
- Service name (Nom du service) : `*`
- Service type (Type de service) : `AWS::AppSync::GraphQLAPI`
- HTTP method (Méthode HTTP) : `*`
- Resource ARN (ARN des ressources) : `arn:aws:appsync:us-west-2:123456789012:apis/3n572shhcpfokwhdnq1ogu59v6`
- Host (Hôte) `*`

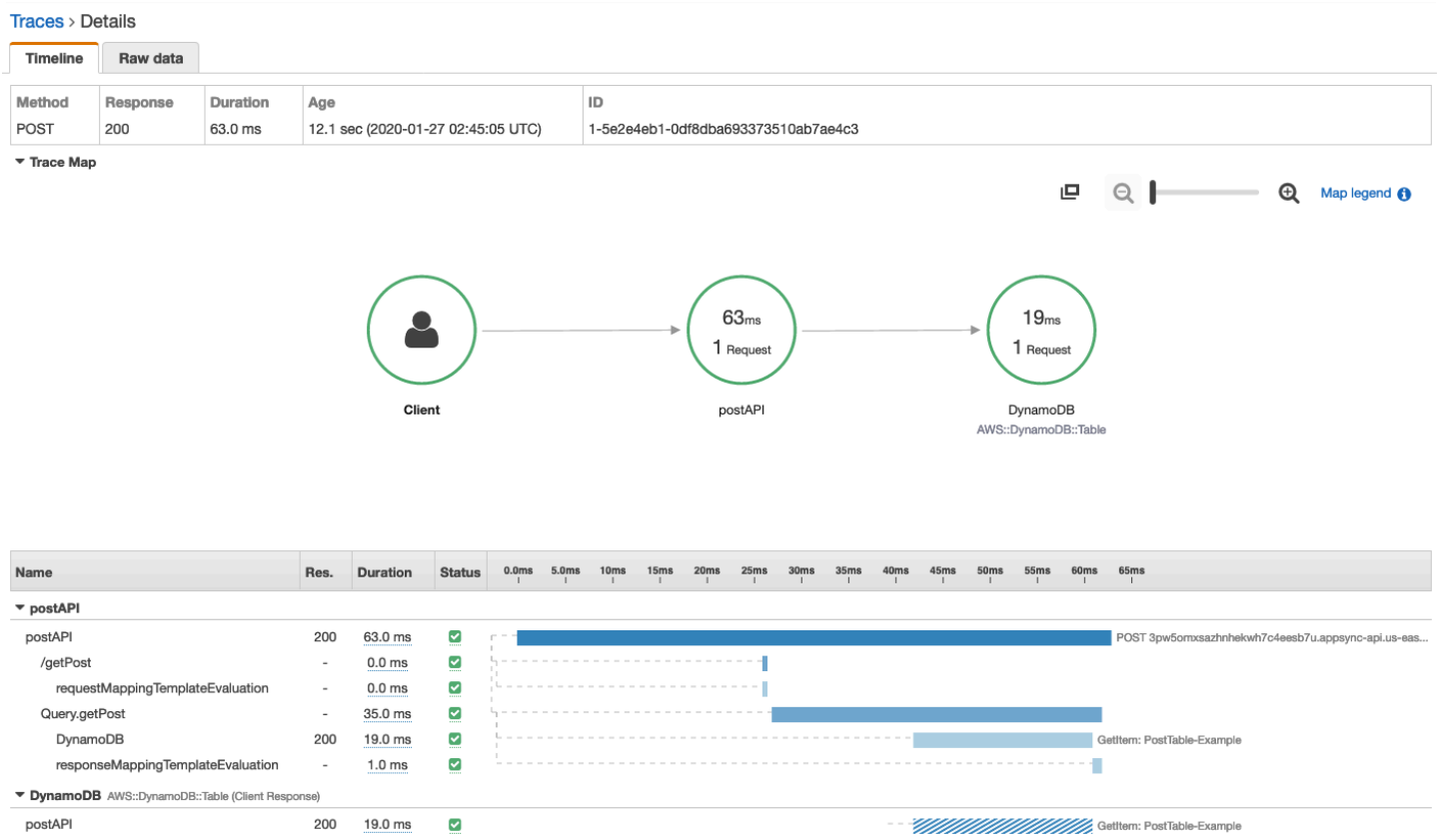
Présentation des suivis

Lorsque vous activez le suivi X-Ray pour votre API GraphQL, vous pouvez utiliser la page détaillée du suivi X-Ray pour examiner les informations de latence détaillées sur les demandes faites à votre API. L'exemple suivant montre la vue du suivi ainsi que la carte des services pour cette demande spécifique. La demande a été effectuée à une API appeléepostAPIavec un type Post, dont les données sont contenues dans une table Amazon DynamoDB appeléePostTable-Example.

L'image de suivi suivante correspond à la requête GraphQL suivante :

```
query getPost {
    getPost(id: "1") {
      id
      title
    }
}
```

Le résolveur de la consolegetPostquery utilise la source de données DynamoDB sous-jacente. La vue de suivi suivante montre l'appel à DynamoDB, ainsi que les latences de différentes parties de l'exécution de la requête :



- Dans l'image précédente, `/getPost` représente le chemin d'accès complet à l'élément en cours de résolution. Dans ce cas, `getPost` étant un champ sur le type `Query` racine, il apparaît directement après la racine du chemin d'accès.
- `requestMappingTemplateEvaluation`représente le temps passé parAWSAppSync évalue le modèle de mappage de demande pour cet élément dans la requête.
- `Query.getPost` représente un type et un champ (au format `Type.field`). Il peut contenir plusieurs sous-segments, en fonction de la structure de l'API et de la demande en cours de suivi.
  - `DynamoDB` représente la source de données attachée à ce résolveur. Il contient la latence de l'appel réseau à DynamoDB pour résoudre le champ.
  - `responseMappingTemplateEvaluation`représente le temps passé parAWSAppSync évalue le modèle de mappage de réponse pour cet élément dans la requête.

Lorsque vous affichez des suivis dans X-Ray, vous pouvez obtenir des informations contextuelles et des métadonnées supplémentaires sur les sous-segments de la consoleAWSSegment AppSync en choisissant les sous-segments et en explorant la vue détaillée.

Pour certaines requêtes complexes ou profondément imbriquées, notez que le segment livré à X-Ray parAWSAppSync peut être supérieur à la taille maximale autorisée pour les documents de segment, comme défini dansAWS X-RayDocuments de segment. Les X-Ray n'affichent pas les segments qui dépassent la limite.

Journalisation des appels d'API AWS AppSync avec AWS CloudTrail

AWS AppSync est intégré à AWS CloudTrail, un service qui fournit une registre des actions effectuées par un utilisateur, un rôle ou un service AWS dans AWS AppSync. CloudTrail capture tous les appels d'API pour AWS AppSync en tant qu'événements. Les appels capturés incluent les appels depuis la console AWS AppSync et les appels de code aux API AWS AppSync. Vous pouvez utiliser les informations collectées parCloudTrailpour déterminer la demande qui a été faite àAWS AppSync, l'adresse IP du demandeur, l'auteur de la demande, la date à laquelle elle a été faite et des informations supplémentaires.

Vous pouvez créer unsentierpour permettre la livraison continue deCloudTrailévénements relatifs à un compartiment Amazon Simple Storage Service (Amazon S3), y compris les événements pourAWS AppSync. Si vous ne configurez pas de parcours, vous pouvez toujours consulter les événements les plus récents dansCloudTrailconsole.

> **Important**
>
> Toutes les actions GraphQL ne sont pas enregistrées actuellement. AppSyncn'enregistre pas les actions de requête et de mutation dansCloudTrail.

Pour plus d'informations sur CloudTrail, consultez le AWS CloudTrailGuide de l'utilisateur .

Informations AWS AppSync dans CloudTrail

CloudTrail est activé sur votre compte AWS lorsque vous créez le compte. Dans leCloudTrailconsole enHistorique de l'événement, vous pouvez consulter, rechercher et télécharger les événements

récents dans votreAWScompte. Pour plus d'informations, voirAfficher les événements avecCloudTrailHistorique de l'événementdans leAWS CloudTrailGuide de l'utilisateur.

Pour enregistrer en continu les événements dans votre compte AWS, y compris les événements d'AWS AppSync, créez un journal d'activité. Par défaut, lorsque vous créez un journal de suivi dans la console, il s'applique à toutes les régions AWS. Le journal de suivi consigne les événements de toutes les Régions dans la partition AWS et livre les fichiers journaux dans le compartiment Amazon S3 de votre choix. En outre, vous pouvez configurer d'autres services AWS pour analyser plus en profondeur les données d'événement collectées dans les journaux CloudTrail et agir sur celles-ci. Pour de plus amples informations, veuillez consulter les rubriques suivantes dans le Guide de l'utilisateur AWS CloudTrail :

- Création d'un sentier pour votreAWSCompte
- AWSIntégrations de services avecCloudTrailJournaux
- Configuration des Notifications de Amazon SNS pour CloudTrail
- Réception de fichiers journaux CloudTrail de plusieurs régions
- Réception de fichiers journaux CloudTrail de plusieurs comptes

CloudTrailenregistre toutAWS AppSyncOpérations d'API. Par exemple, les appels au`CreateGraphqlApi`,`CreateDataSource`, et`ListResolvers`Les API génèrent des entrées dansCloudTrailfichiers journaux. Ces opérations, ainsi que d'autres, sont documentées dans leAWS AppSyncRéférence d'API.

Chaque événement ou entrée de journal contient des informations sur la personne ayant initié la demande. Les informations relatives à l'identité vous permettent de déterminer :

- Si la demande a été effectuée avec les informations d'identification utilisateur racine ou AWS Identity and Access Management (IAM).
- Si la demande a été effectuée avec les informations d'identification de sécurité temporaires d'un rôle ou d'un utilisateur fédéré.
- Si la requête a été effectuée par un autre service AWS.

Pour plus d'informations, voirCloudTrailÉlément UserIdentitydans leAWS CloudTrailGuide de l'utilisateur.

Présentation des AWS AppSync entrées des fichiers journaux

CloudTrailfournit les événements sous forme de fichiers journaux contenant une ou plusieurs entrées de journal. Un événement représente une demande unique provenant de n'importe quelle source et inclut des informations sur l'opération demandée, la date et l'heure de l'opération, les paramètres de la demande, etc. Comme ces fichiers journaux ne constituent pas une trace ordonnée des appels d'API publics, ils n'apparaissent pas dans un ordre spécifique.

L'exemple suivantCloudTrailune entrée de journal illustre`CreateApiKey`opération.


```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/Alice",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "Alice"
    },
    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "CreateApiKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
    "requestParameters": {
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": {
      "apiKey": {
      "id": "***",
      "expires": 1518037200000
      }
    },
    "requestID": "99999999-9999-9999-9999-999999999999",
    "eventID": "99999999-9999-9999-9999-999999999999",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
    }
  ]
```

```
}
```

L'exemple suivantCloudTrailune entrée de journal illustreListApiKeysopération.

```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/Alice",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "Alice"
    },
    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "ListApiKeys",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
    "requestParameters": {
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": {
      "apiKeys": [
              {
                    "id": "***",
                    "expires": 1517954400000
              },
              {
                    "id": "***",
                    "expires": 1518037200000
              },
            ]
    },
    "requestID": "99999999-9999-9999-9999-999999999999",
    "eventID": "99999999-9999-9999-9999-999999999999",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
    }
  ]
```

```
}
```

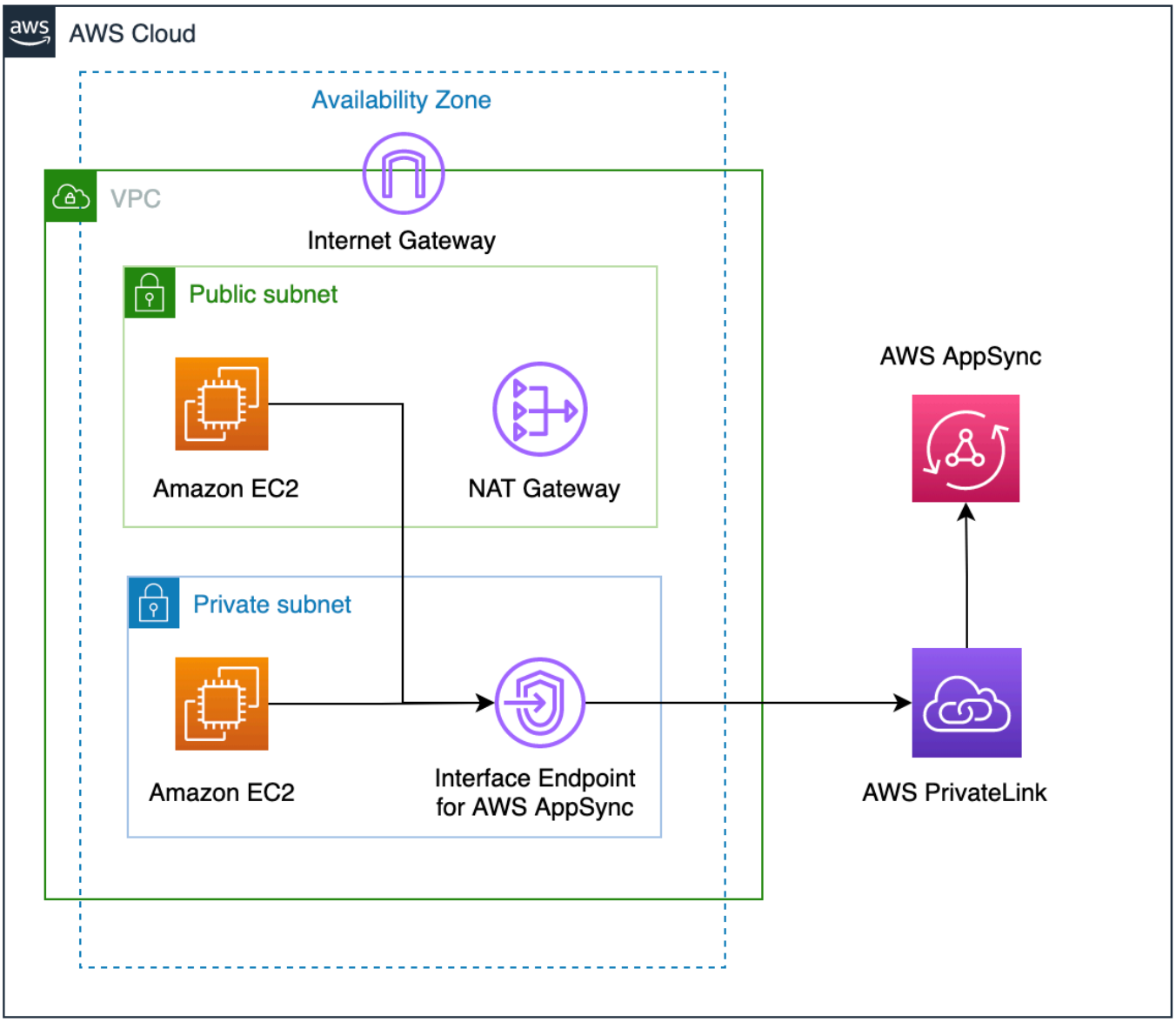
L'exemple suivantCloudTrailune entrée de journal illustre`DeleteApiKey`opération.

```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/Alice",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "Alice"
    },
    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "DeleteApiKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
    "requestParameters": {
      "id": "***",
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": null,
    "requestID": "99999999-9999-9999-9999-999999999999",
    "eventID": "99999999-9999-9999-9999-999999999999",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
    }
  ]
}
```

En utilisantAWS AppSyncAPI privées

Si vous utilisez Amazon Virtual Private Cloud (Amazon VPC), vous pouvez créerAWS AppSyncLes API privées, qui sont des API accessibles uniquement depuis un VPC. Avec une API privée, vous pouvez restreindre l'accès des API à vos applications internes et vous connecter à vos points de terminaison GraphQL et Realtime sans exposer les données publiquement.

Pour établir une connexion privée entre votre VPC etAWS AppSyncservice, vous devez créer unpoint de terminaison VPC d'interface. Les points de terminaison d'interface sont alimentés par AWS PrivateLink, ce qui vous permet d'accéder en privé aux API AWS AppSync sans passerelle Internet, périphérique NAT, connexion VPN ou connexion AWS Direct Connect. Les instances de votre VPC ne requièrent pas d'adresses IP publiques pour communiquer avec les API AWS AppSync. Trafic entre votre VPC etAWS AppSyncne quitte pas leAWSréseau.



Certains facteurs supplémentaires doivent être pris en compte avant d'activer les fonctionnalités de l'API privée :

- Configuration des points de terminaison de l'interface VPC pourAWS AppSynclorsque les fonctionnalités de DNS privé sont activées, les ressources du VPC ne pourront pas invoquer d'autresAWS AppSyncAPI publiques utilisant leAWS AppSyncURL de l'API générée. Cela est dû au fait que la demande à l'API publique est acheminée via le point de terminaison de l'interface, ce qui n'est pas autorisé pour les API publiques. Pour invoquer des API publiques dans ce scénario, il est recommandé de configurer des noms de domaine personnalisés sur les API publiques, qui peuvent ensuite être utilisés par les ressources du VPC pour appeler l'API publique.
- VotreAWS AppSyncLes API privées ne seront disponibles que depuis votre VPC. LeAWS AppSyncconsole L'éditeur de requêtes ne pourra accéder à votre API que si la configuration réseau de votre navigateur peut acheminer le trafic vers votre VPC (par exemple, connexion via VPN ou viaAWS Direct Connect).
- Avec un point de terminaison d'interface VPC pourAWS AppSync, vous pouvez accéder à n'importe quelle API privée dans le mêmeAWScompte et région. Pour restreindre davantage l'accès aux API privées, vous pouvez envisager les options suivantes :
  - S'assurer que seuls les administrateurs requis peuvent créer des interfaces de point de terminaison VPC pourAWS AppSync.
  - Utilisation de politiques personnalisées relatives aux points de terminaison du VPC pour limiter les API pouvant être invoquées à partir des ressources du VPC.
  - Pour les ressources du VPC, nous vous recommandons d'utiliser l'autorisation IAM pour appelerAWS AppSyncAPI en veillant à ce que les ressources se voient attribuer des rôles délimités aux API.
- Lorsque vous créez ou utilisez des politiques qui restreignent les principes IAM, vous devez définir`authorizationType`de la méthode pour`AWS_IAM`ou`NONE`.

CréationAWS AppSyncAPI privées

Les étapes ci-dessous vous montrent comment créer des API privées dansAWS AppSyncservice.

> **Warning**
>
> Vous pouvez activer les fonctionnalités de l'API privée uniquement lors de la création de l'API. Ce paramètre ne peut pas être modifié sur unAWS AppSyncAPI ou unAWS AppSyncAPI privée après sa création.

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.

- Dans le Tableau de bord, choisissez Créer une API.

2. ChoisissezConcevez une API à partir de zéro, puis choisissezSuivant.
3. Dans leAPI privéesection, choisissezUtiliser les fonctionnalités de l'API privée.
4. Configurez le reste des options, passez en revue les données de votre API, puis choisissezCréez.

Avant de pouvoir utiliser votreAWS AppSyncAPI privée, vous devez configurer un point de terminaison d'interface pourAWS AppSyncdans votre VPC. Notez que l'API privée et le VPC doivent se trouver dans le même emplacementAWScompte et région.

Création d'un point de terminaison d'interface pourAWS AppSync

Vous pouvez créer un point de terminaison d'interface pourAWS AppSyncà l'aide de la console Amazon VPC ou duAWS Command Line Interface(AWS CLI). Pour de plus amples informations, veuillez consulter Création d'un point de terminaison d'interface dans le Guide de l'utilisateur Amazon VPC.

Console

1. Connectez-vous auAWS Management Consoleet ouvrez lePoints de terminaisonpage de la console Amazon VPC.
2. Choisissez Créer un point de terminaison.
   a. Dans leCatégorie de servicechamp, vérifiez queAWSservicesest sélectionné.
   b. Dans leServicestable, choisissez`com.amazonaws.{region}.appsync-api`. Vérifiez queTypela valeur de la colonne est`Interface`.
   c. Dans leVPCdans ce champ, choisissez un VPC et ses sous-réseaux.
   d. Pour activer les fonctionnalités DNS privées pour le point de terminaison de l'interface, cochezActiver le nom DNScase à cocher.
   e. Dans leGroupe de sécuritédans ce champ, sélectionnez un ou plusieurs groupes de sécurité.
3. Choisissez Créer un point de terminaison.

CLI

Utilisez la commande `create-vpc-endpoint` et spécifiez l'ID du VPC, le type du point de terminaison de VPC (interface), le nom du service, les sous-réseaux qui utiliseront le point de terminaison et les groupes de sécurité à associer aux interfaces réseau du point de terminaison. Par exemple :

```
$ aws ec2 create-vpc-endpoint –vpc-id vpc-ec43eb89 \
  –vpc-endpoint-type Interface \
  –service-name com.amazonaws.{region}.appsync-api \
  –subnet-id subnet-abababab –security-group-id sg-1a2b3c4d
```

Pour utiliser l'option DNS privé, vous devez définir`enableDnsHostnames`et`enableDnsSupport`attributsvaleurs de votre VPC. Pour plus d'informations, consultez Affichage et mise à jour de la prise en charge de DNS pour votre VPC dans le Guide de l'utilisateur Amazon VPC. Si vous activez les fonctionnalités DNS privées pour le point de terminaison de l'interface, vous pouvez envoyer des demandes à votreAWS AppSyncAPI GraphQL et point de terminaison en temps réel utilisant ses points de terminaison DNS publics par défaut en utilisant le format ci-dessous :

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

Pour plus d'informations sur les points de terminaison de service, voirPoints de terminaison de service et quotasdans leAWSRéférence générale.

Pour plus d'informations sur les interactions de service avec les points de terminaison de l'interface, voirAccès à un service via un point de terminaison d'interfacedans leGuide de l'utilisateur d'Amazon VPC.

Pour plus d'informations sur la création et la configuration d'un point de terminaison à l'aideAWS CloudFormation, consultez leAWSPoint de terminaison ::EC2 : :VPCressource dans leAWS CloudFormationGuide de l'utilisateur.

Exemples avancés

Si vous activez les fonctionnalités DNS privées pour le point de terminaison de l'interface, vous pouvez envoyer des demandes à votreAWS AppSyncAPI GraphQL et point de terminaison en temps réel utilisant ses points de terminaison DNS publics par défaut en utilisant le format ci-dessous :

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

À l'aide des noms d'hôtes DNS publics du point de terminaison VPC de l'interface, l'URL de base pour appeler l'API sera au format suivant :

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.
{region}.vpce.amazonaws.com/graphql
```

Vous pouvez également utiliser le nom d'hôte DNS spécifique à AZ si vous avez déployé un point de terminaison dans l'AZ :

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}-{az_id}.appsync-api.
{region}.vpce.amazonaws.com/graphql.
```

L'utilisation du nom DNS public du point de terminaison du VPC nécessiteraAWS AppSyncNom d'hôte du point de terminaison de l'API à transmettre`Host`ou en tant que `x-appsync-domain`en-tête de la demande. Ces exemples utilisent un`TodoAPIqui a été créé dans le`Lancer un exemple de schémaguide :

```
curl https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.
{region}.vpce.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-H "Host:{api_url_identifier}.appsync-api.{region}.amazonaws.com" \
-d '{"query":"mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input:
 $createtodoinput) {\n id\n name\n where\n when\n description\n }\n}","variables":
{"createtodoinput":{"name":"My first GraphQL task","when":"Friday Night","where":"Day
 1","description":"Learn more about GraphQL"}}}'
```

Dans les exemples suivants, nous utiliseronsTodoapplication générée dansLancer un exemple de schémaguide. Pour tester l'exemple d'API Todo, nous allons utiliser le DNS privé pour appeler l'API. Vous pouvez utiliser n'importe quel outil de ligne de commande de votre choix ; cet exemple utilisebouclepour envoyer des requêtes et des mutations etwscatpour configurer des abonnements. Pour imiter notre exemple, remplacez les valeurs entre crochets`{ }`dans les commandes ci-dessous avec les valeurs correspondantes de votreAWScompte.

Tester l'opération de mutation —**`createTodo`**Demander

```
curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
```

```
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query":"mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input:
 $createtodoinput) {\n id\n name\n where\n when\n description\n }\n}","variables":
{"createtodoinput":{"name":"My first GraphQL task","when":"Friday Night","where":"Day
 1","description":"Learn more about GraphQL"}}}'
```

Tester l'opération de mutation —**`createTodo`**Réponse

```
{
    "data": {
        "createTodo": {
            "id": "<todo-id>",
            "name": "My first GraphQL task",
            "where": "Day 1",
            "when": "Friday Night",
            "description": "Learn more about GraphQL"
        }
    }
}
```

Tester le fonctionnement de la requête —**`listTodos`**Demander

```
curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query":"query ListTodos {\n listTodos {\n items {\n description\n id\n name
\n when\n where\n }\n }\n}\n","variables":{"createtodoinput":{"name":"My first
 GraphQL task","when":"Friday Night","where":"Day 1","description":"Learn more about
 GraphQL"}}}'
```

Tester le fonctionnement de la requête —**`listTodos`**Demander

```
{
  "data": {
    "listTodos": {
      "items": [
        {
          "description": "Learn more about GraphQL",
          "id": "<todo-id>",
          "name": "My first GraphQL task",
          "when": "Friday night",
```

```
          "where": "Day 1"
        }
      ]
    }
  }
}
```

Test du fonctionnement de l'abonnement — Abonnement à**`createTodo`**mutation

Pour configurer les abonnements GraphQL dansAWS AppSync, voirCréation d'un système en temps réelWebSocketclient. À partir d'une instance Amazon EC2 dans un VPC, vous pouvez tester votreAWS AppSyncPoint de terminaison d'abonnement à une API privée utilisantwscat. L'exemple ci-dessous utilise un`API KEY`pour autorisation.

```
$ header=`echo '{"host":"{api_url_identifier}.appsync-api.{region}.amazonaws.com","x-
api-key":"da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}"}' | base64 | tr -d '\n'`
$ wscat -p 13 -s graphql-ws -c  "wss://{api_url_identifier}.appsync-realtime-api.us-
west-2.amazonaws.com/graphql?header=$header&payload=e30="
Connected (press CTRL+C to quit)
> {"type": "connection_init"}
< {"type":"connection_ack","payload":{"connectionTimeoutMs":300000}}
< {"type":"ka"}
> {"id":"f7a49717","payload":{"data":"{\"query\":\"subscription
 onCreateTodo {onCreateTodo {description id name where when}}\",
\"variables\":{}}","extensions":{"authorization":{"x-api-key":"da2-
{xxxxxxxxxxxxxxxxxxxxxxxxxx}","host":"{api_url_identifier}.appsync-api.
{region}.amazonaws.com"}}},"type":"start"}
< {"id":"f7a49717","type":"start_ack"}
```

Vous pouvez également utiliser le nom de domaine du point de terminaison du VPC tout en vous assurant de spécifierHôteen-tête dans le`wscat`commande pour établir le websocket :

```
$ header=`echo '{"host":"{api_url_identifier}.appsync-api.{region}.amazonaws.com","x-
api-key":"da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}"}' | base64 | tr -d '\n'`
$ wscat -p 13 -s graphql-ws -c  "wss://{vpc_endpoint_id}-
{endpoint_dns_identifier}.appsync-api.{region}.vpce.amazonaws.com/graphql?header=
$header&payload=e30=" --header Host:{api_url_identifier}.appsync-realtime-api.us-
west-2.amazonaws.com
Connected (press CTRL+C to quit)
> {"type": "connection_init"}
< {"type":"connection_ack","payload":{"connectionTimeoutMs":300000}}
< {"type":"ka"}
```

```
> {"id":"f7a49717","payload":{"data":"{\"query\":\"subscription
 onCreateTodo {onCreateTodo {description id priority title}}\",
\"variables\":{}}","extensions":{"authorization":{"x-api-key":"da2-
{xxxxxxxxxxxxxxxxxxxxxxxxxx}","host":"{api_url_identifier}.appsync-api.
{region}.amazonaws.com"}}},"type":"start"}
< {"id":"f7a49717","type":"start_ack"}
```

Exécutez le code de mutation ci-dessous :

```
curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query":"mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input:
 $createtodoinput) {\n id\n name\n where\n when\n description\n }\n}","variables":
{"createtodoinput":{"name":"My first GraphQL task","when":"Friday Night","where":"Day
 1","description":"Learn more about GraphQL"}}}'
```

Ensuite, un abonnement est déclenché et le message de notification apparaît comme indiqué ci-dessous :

```
< {"id":"f7a49717","type":"data","payload":{"data":{"onCreateTodo":{"description":"Go
 to the shops","id":"169ce516-b7e8-4a6a-88c1-ab840184359f","priority":5,"title":"Go to
 the shops"}}}}
```

Utiliser les politiques IAM pour limiter la création d'API publiques

AWS AppSyncprend en charge IAM`Condition`déclarationsà utiliser avec des API privées. Le`visibility`un champ peut être inclus dans les déclarations de politique IAM pour`appsync:CreateGraphqlApi`opération permettant de contrôler quels rôles et utilisateurs IAM peuvent créer des API privées et publiques. Cela permet à un administrateur IAM de définir une politique IAM qui permettra uniquement à un utilisateur de créer une API GraphQL privée. Un utilisateur qui tente de créer une API publique recevra un message non autorisé.

Par exemple, un administrateur IAM peut créer la déclaration de politique IAM suivante pour permettre la création d'API privées :

```
{
    "Sid": "AllowPrivateAppSyncApis",
    "Effect": "Allow",
    "Action": "appsync:CreateGraphqlApi",
    "Resource": "*",
```

```
    "Condition": {
        "ForAnyValue:StringEquals": {
            "appsync:Visibility": "PRIVATE"
        }
    }
}
```

Un administrateur IAM peut également ajouter les éléments suivantspolitique de contrôle des servicespour bloquer tous les utilisateurs d'unAWSorganisation depuis la créationAWS AppSyncAPI autres que les API privées :

```
{
    "Sid": "BlockNonPrivateAppSyncApis",
    "Effect": "Deny",
    "Action": "appsync:CreateGraphqlApi",
    "Resource": "*",
    "Condition": {
        "ForAnyValue:StringNotEquals": {
            "appsync:Visibility": "PRIVATE"
        }
    }
}
```

Configuration de la complexité de l'exécution, de la profondeur des requêtes et de l'introspection de GraphQL avec AWS AppSync

AWS AppSyncvous permet d'activer ou de désactiver les fonctionnalités d'introspection et de définir des limites au nombre de niveaux imbriqués et de résolveurs dans une seule requête.

Utilisation de la fonction d'introspection

> **Tip**
>
> Pour plus d'informations sur l'introspection dans GraphQL, consultez cet article sur le site Web de la fondation GraphQL.

Par défaut, GraphQL vous permet d'utiliser l'introspection pour interroger le schéma lui-même afin de découvrir ses types, ses champs, ses requêtes, ses mutations, ses abonnements, etc. Il s'agit

d'une fonctionnalité importante pour apprendre comment les données sont mises en forme et traitées par votre service GraphQL. Cependant, il y a certaines choses à prendre en compte lorsqu'il s'agit d'introspection. Vous avez peut-être un cas d'utilisation qui bénéficierait de la désactivation de l'introspection, par exemple un cas dans lequel les noms de champs peuvent être sensibles ou masqués ou dans le cas où le schéma complet de l'API est destiné à être laissé sans documentation pour les consommateurs. Dans ces cas, la publication de données de schéma par introspection peut entraîner la fuite de données volontairement privées.

Pour éviter que cela ne se produise, vous pouvez désactiver l'introspection. Cela empêchera les parties non autorisées d'utiliser les champs d'introspection de votre schéma. Cependant, il est important de noter que l'introspection est utile aux équipes de développement pour savoir comment les données de leur service sont traitées. En interne, il peut être utile de garder l'introspection activée tout en la désactivant dans le code de production comme couche de sécurité supplémentaire. Une autre façon de gérer cela consiste à ajouter une méthode d'autorisation, qui fournit AWS AppSync également. Pour plus d'informations, consultez la section Autorisation.

AWS AppSyncvous permet d'activer ou de désactiver l'introspection au niveau de l'API. Pour activer ou désactiver l'introspection, procédez comme suit :

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
2. Sur la page API, choisissez le nom d'une API GraphQL.
3. Sur la page d'accueil de votre API, dans le volet de navigation, sélectionnez Paramètres.
4. Dans les configurations d'API, choisissez Modifier.
5. Sous Requêtes d'introspection, procédez comme suit :
    - Activez ou désactivez l'option Activer les requêtes d'introspection.
6. Choisissez Enregistrer.

Lorsque l'introspection est activée (comportement par défaut), l'utilisation du système d'introspection fonctionnera normalement. Par exemple, l'image ci-dessous montre un `__schema` champ traitant tous les types disponibles dans le schéma :

Lorsque vous désactivez cette fonctionnalité, une erreur de validation apparaîtra plutôt dans la réponse :

Configuration des limites de profondeur des requêtes

Il peut arriver que vous souhaitiez contrôler de manière plus précise le fonctionnement de l'API au cours d'une opération. L'un de ces contrôles consiste à ajouter une limite au nombre de niveaux imbriqués qu'une requête peut traiter. Par défaut, les requêtes peuvent traiter un nombre illimité de niveaux imbriqués. Le fait de limiter les requêtes à un certain nombre de niveaux imbriqués peut avoir des répercussions sur les performances et la flexibilité de votre projet. Prenons la requête suivante :

```
query MyQuery {
```

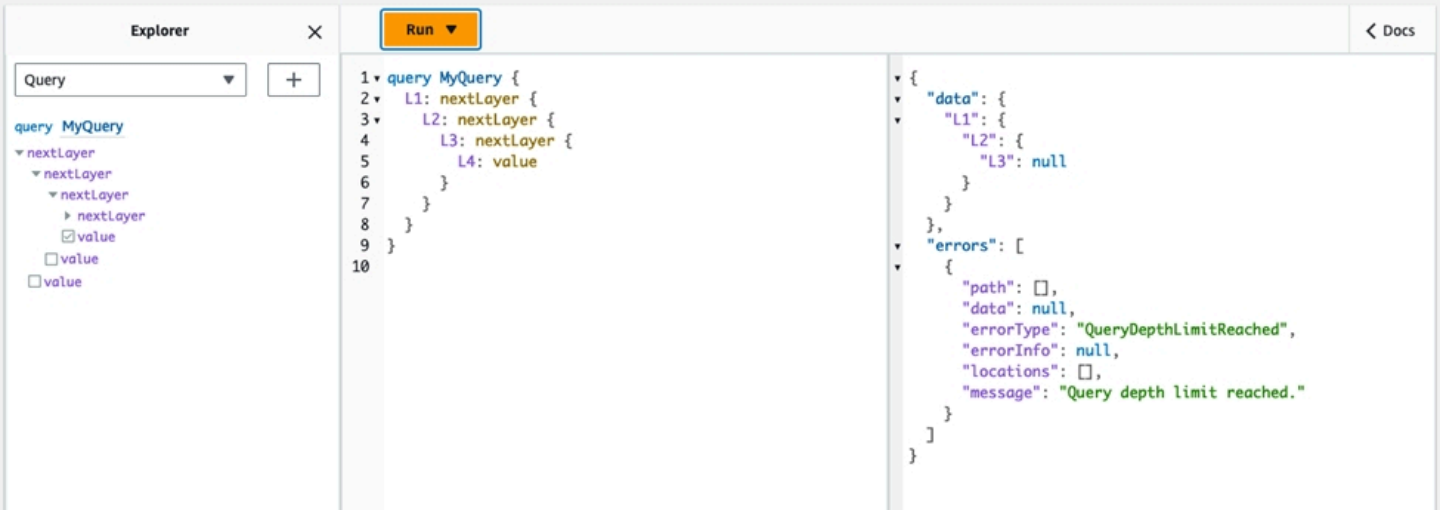
```
  L1: nextLayer {
    L2: nextLayer {
      L3: nextLayer {
        L4: value
      }
    }
  }
}
```

Votre projet peut nécessiter de limiter les requêtes à `L1` ou dans un `L2` but précis. Par défaut, l'intégralité de la requête de `L1` à `L4` serait traitée sans aucun moyen de contrôle. En définissant une limite, vous pouvez empêcher les requêtes d'accéder à tout ce qui dépasse le niveau spécifié.

Pour ajouter une limite de profondeur de requête, procédez comme suit :

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
2. Sur la page API, choisissez le nom d'une API GraphQL.
3. Sur la page d'accueil de votre API, dans le volet de navigation, sélectionnez Paramètres.
4. Dans les configurations d'API, choisissez Modifier.
5. Sous Profondeur de la requête, procédez comme suit :
   a. Activez ou désactivez l'option Activer la profondeur des requêtes.
   b. Dans Profondeur maximale, définissez la limite de profondeur. Cela peut être compris entre 1 et`75`.
6. Choisissez Enregistrer.

Lorsqu'une limite est définie, le dépassement de sa limite supérieure entraîne une `QueryDepthLimitReached` erreur. Par exemple, l'image ci-dessous montre une requête dont la limite de 2 profondeur dépasse la limite jusqu'aux troisième (`L3`) et quatrième (`L4`) niveaux :



Notez que les champs peuvent toujours être marqués comme nullables ou non nullables dans le schéma. Si un champ non nullable reçoit une `QueryDepthLimitReached` erreur, cette erreur sera renvoyée au premier champ parent nullable.

Configuration des limites du nombre de résolveurs

Vous pouvez également contrôler le nombre de résolveurs que chaque requête peut traiter. Tout comme la profondeur de la requête, vous pouvez définir une limite à ce montant. Prenez la requête suivante qui contient trois résolveurs :

```
query MyQuery {
  resolver1: resolver
  resolver2: resolver
  resolver3: resolver
}
```

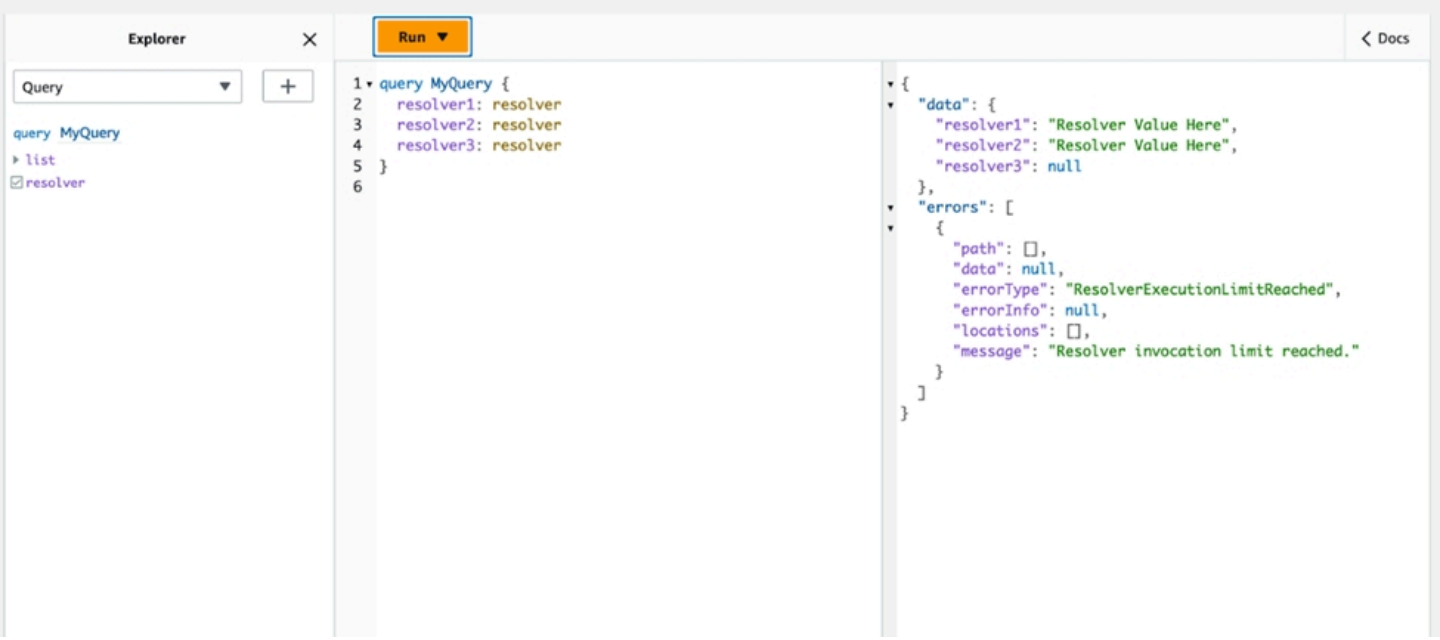
Par défaut, chaque requête peut traiter jusqu'à 10 000 résolveurs. Dans l'exemple ci-dessus`resolver1`,`resolver2`, et `resolver3` seront traités. Cependant, votre projet peut nécessiter de limiter chaque requête à la gestion d'un ou deux résolveurs au total. En définissant une limite, vous pouvez indiquer à la requête de ne traiter aucun résolveur au-delà d'un certain nombre, comme le premier (`resolver1`) ou le second (`resolver2`) résolveur.

Pour ajouter une limite au nombre de résolveurs, procédez comme suit :

1. Connectez-vous à AWS Management Console et ouvrez la console AppSync.
2. Sur la page API, choisissez le nom d'une API GraphQL.

3. Sur la page d'accueil de votre API, dans le volet de navigation, sélectionnez Paramètres.
4. Dans les configurations d'API, choisissez Modifier.
5. Sous Limite du nombre de résolveurs, procédez comme suit :
   a. Activez Activer le nombre de résolveurs.
   b. Dans Nombre maximal de résolveurs, définissez la limite du nombre. Cela peut être compris entre 1 et`10000`.
6. Choisissez Enregistrer.

Tout comme la limite de profondeur de requête, le dépassement de la limite de résolution configurée entraîne la fin de la requête avec une `ResolverExecutionLimitReached` erreur sur les résolveurs supplémentaires. Dans l'image ci-dessous, une requête dont le nombre de résolveurs est limité à 2 essaie de traiter trois résolveurs. En raison de cette limite, le troisième résolveur génère une erreur et ne s'exécute pas.



Utilisation de variables d'environnement dans AWS AppSync

Vous pouvez utiliser des variables d'environnement pour ajuster le comportement de vos AWS AppSync résolveurs et de vos fonctions sans mettre à jour votre code. Les variables d'environnement sont des paires de chaînes stockées avec la configuration de votre API qui sont mises à la disposition de vos résolveurs et de vos fonctions pour qu'elles puissent les exploiter lors de l'exécution. Ils sont particulièrement utiles dans les situations où vous devez référencer des données de configuration

qui ne sont disponibles que lors de la configuration initiale, mais qui doivent être utilisées par vos résolveurs et vos fonctions pendant l'exécution. Les variables d'environnement exposent les données de configuration dans votre code, réduisant ainsi le besoin de coder ces valeurs en dur.

> **Note**
>
> Pour renforcer la sécurité de la base de données, nous vous recommandons d'utiliser Secrets Manager ou AWS Systems Manager Parameter Store au lieu de variables d'environnement pour stocker les informations d'identification ou les informations sensibles. Pour tirer parti de cette fonctionnalité, consultez la section Invocation de AWS services avec des sources de données AWS AppSync HTTP.

Les variables d'environnement doivent suivre plusieurs comportements et règles pour fonctionner correctement :

- Les JavaScript résolveurs/fonctions et les modèles VTL prennent en charge les variables d'environnement.
- Les variables d'environnement ne sont pas évaluées avant l'invocation de la fonction.
- Les variables d'environnement ne prennent en charge que les valeurs de chaîne.
- Toute valeur définie dans une variable d'environnement est considérée comme une chaîne littérale et n'est pas développée.
- Les évaluations des variables devraient idéalement être effectuées dans le code de fonction.

Configuration des variables d'environnement (console)

Vous pouvez configurer des variables d'environnement pour votre API AWS AppSync GraphQL en créant la variable et en définissant sa paire clé-valeur. Vos résolveurs et fonctions utiliseront le nom de clé de la variable d'environnement pour récupérer la valeur lors de l'exécution. Pour définir des variables d'environnement dans la AWS AppSync console :

1. Connectez-vous à la AppSyncconsole AWS Management Console et ouvrez-la.
2. Sur la page API, choisissez le nom d'une API GraphQL.
3. Sur la page d'accueil de votre API, dans le volet de navigation, sélectionnez Paramètres.
4. Sous Variables d'environnement, choisissez Ajouter une variable d'environnement.
5. Choisissez Ajouter une variable d’environnement.

6. Entrez une clé et une valeur.
7. Si nécessaire, répétez les étapes 5 et 6 pour ajouter d'autres valeurs clés. Si vous devez supprimer une valeur de clé, choisissez l'option Supprimer et la ou les clés à supprimer.
8. Sélectionnez Envoyer.

> **Tip**
>
> Vous devez suivre quelques règles lors de la création de clés et de valeurs :
>
> - Les clés doivent commencer par une lettre.
> - Les clés doivent comporter au moins deux caractères.
> - Les touches ne peuvent contenir que des lettres, des chiffres et le caractère de soulignement (_).
> - Les valeurs peuvent comporter jusqu'à 512 caractères.
> - Vous pouvez configurer jusqu'à 50 paires clé-valeur dans une API GraphQL.

Configuration des variables d'environnement (API)

Pour définir une variable d'environnement à l'aide d'API, vous pouvez utiliser`PutGraphqlApiEnvironmentVariables`. La commande CLI correspondante est`put-graphql-api-environment-variables`.

Pour récupérer une variable d'environnement à l'aide d'API, vous pouvez utiliser`GetGraphqlApiEnvironmentVariables`. La commande CLI correspondante est`get-graphql-api-environment-variables`.

La commande doit contenir l'ID de l'API et la liste des variables d'environnement :

```
aws appsync put-graphql-api-environment-variables \
  --api-id "<api-id>" \
  --environment-variables '{"key1":"value1","key2":"value2", …}'
```

L'exemple suivant définit deux variables d'environnement dans une API avec l'ID d'`abcdefghijklmnopqrstuvwxyz`utilisation de la `put-graphql-api-environment-variables` commande :

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true"}'
```

Notez que lorsque vous appliquez des variables d'environnement à l'aide de la `put-graphql-api-environment-variables` commande, le contenu de la structure des variables d'environnement est remplacé ; cela signifie que les variables d'environnement existantes seront perdues. Pour conserver les variables d'environnement existantes lorsque vous en ajoutez de nouvelles, incluez toutes les paires clé-valeur existantes ainsi que les nouvelles dans votre demande. À l'aide de l'exemple ci-dessus, si vous souhaitez ajouter`"EMPTY":""`, vous pouvez effectuer les opérations suivantes :

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true", "EMPTY":""}'
```

Pour récupérer la configuration actuelle, utilisez la `get-graphql-api-environment-variables` commande :

```
aws appsync get-graphql-api-environment-variables --api-id "<api-id>"
```

En utilisant l'exemple ci-dessus, vous pouvez utiliser la commande suivante :

```
aws appsync get-graphql-api-environment-variables --api-id "abcdefghijklmnopqrstuvwxyz"
```

Le résultat affichera la liste des variables d'environnement ainsi que leurs valeurs clés :

```
{
    "environmentVariables": {
        "USER_TABLE": "users_prod",
        "DEBUG": "true",
        "EMPTY": ""
    }
}
```

Configuration des variables d'environnement (CFN)

Vous pouvez utiliser le modèle ci-dessous pour créer des variables d'environnement :

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  GraphQLApiWithEnvVariables:
    Type: "AWS::AppSync::GraphQLApi"
    Properties:
      Name: "MyApiWithEnvVars"
      AuthenticationType: "AWS_IAM"
      EnvironmentVariables:
        EnvKey1: "non-empty"
        EnvKey2: ""
```

variables d'environnement et API fusionnées

Les variables d'environnement définies dans les API source sont également disponibles dans vos API fusionnées. Les variables d'environnement dans les API fusionnées sont en lecture seule et ne peuvent pas être mises à jour. Notez que les clés de vos variables d'environnement doivent être uniques dans toutes les API source pour que vos fusions aboutissent ; les clés dupliquées entraîneront toujours un échec de fusion.

Récupération de variables d'environnement

Pour récupérer des variables d'environnement dans votre code de fonction, récupérez la valeur de l'`ctx.env`objet dans vos résolveurs et fonctions. Vous trouverez ci-dessous quelques exemples de cela en action.

Publishing to Amazon SNS

Dans cet exemple, notre résolveur HTTP envoie un message à une rubrique Amazon SNS. L'ARN de la rubrique n'est connu qu'une fois que la pile qui définit l'API GraphQL et la rubrique ont été déployées.

```
/**
 * Sends a publish request to the SNS topic
 */
export function request(ctx) {
    const TOPIC_ARN = ctx.env.TOPIC_ARN;
    const { input: values } = ctx.args;
    // this custom function sends values to the SNS topic
    return publishToSNSRequest(TOPIC_ARN, values);
}
```

Transactions with DynamoDB

Dans cet exemple, les noms de la table DynamoDB sont différents si l'API est déployée à des fins de préparation ou si elle est déjà en production. Il n'est pas nécessaire de modifier le code du résolveur. Les valeurs des variables d'environnement sont mises à jour en fonction de l'endroit où l'API est déployée.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: ctx.env.POST_TABLE,
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({ postId }),
        // rest of the configuration
      },
      {
        table: ctx.env.AUTHOR_TABLE,
        operation: 'UpdateItem',
        key: util.dynamodb.toMapValues({ authorId }),
        // rest of the configuration
      },
    ],
  };
}
```

Autorisation et authentification

Cette section décrit les options dont vous disposez pour configurer la sécurité et la protection des données pour vos applications.

Types d'autorisation

Il existe cinq manières d'autoriser les applications à interagir avec votre API AWS AppSync GraphQL. Vous spécifiez le type d'autorisation que vous utilisez en spécifiant l'une des valeurs de type d'autorisation suivantes dans votre appel AWS AppSync d'API ou de CLI :

- **`API_KEY`**

  Pour utiliser des clés API.

- **`AWS_LAMBDA`**

  Pour utiliser une AWS Lambda fonction.

- **`AWS_IAM`**

  Pour utiliser les autorisations AWS Identity and Access Management (IAM).

- **`OPENID_CONNECT`**

  Pour utiliser votre fournisseur OpenID Connect.

- **`AMAZON_COGNITO_USER_POOLS`**

  Pour utiliser un groupe d'utilisateurs Amazon Cognito.

Ces types d'autorisation de base fonctionnent pour la plupart des développeurs. Pour des cas d'utilisation plus avancés, vous pouvez ajouter des modes d'autorisation supplémentaires via la console, la CLI et AWS CloudFormation. Pour les modes d'autorisation supplémentaires, AWS AppSync fournit un type d'autorisation qui prend les valeurs répertoriées ci-dessus (c'est-à-dire `API_KEYAWS_LAMBDA`,`AWS_IAM`,`OPENID_CONNECT`, et`AMAZON_COGNITO_USER_POOLS`).

Lorsque vous spécifiez `API_KEYAWS_LAMBDA`, ou `AWS_IAM` en tant que type d'autorisation principal ou par défaut, vous ne pouvez pas les spécifier à nouveau comme l'un des modes d'autorisation supplémentaires. De même, vous ne pouvez pas dupliquer `API_KEY` `AWS_LAMBDA` ou `AWS_IAM`

utiliser les modes d'autorisation supplémentaires. Vous pouvez utiliser plusieurs groupes d'utilisateurs Amazon Cognito et fournisseurs OpenID Connect. Toutefois, vous ne pouvez pas utiliser de groupes d'utilisateurs Amazon Cognito ou de fournisseurs OpenID Connect dupliqués entre le mode d'autorisation par défaut et l'un des modes d'autorisation supplémentaires. Vous pouvez spécifier différents clients pour votre groupe d'utilisateurs Amazon Cognito ou votre fournisseur OpenID Connect à l'aide de l'expression régulière de configuration correspondante.

Autorisation API_KEY

Les API non authentifiées nécessitent des limitations plus strictes que les API authentifiées. Pour contrôler les limitations des points de terminaison GraphQL non authentifiés, vous pouvez utiliser des clés API. Une clé d'API est une valeur codée en dur dans votre application qui est générée par le AWS AppSync service lorsque vous créez un point de terminaison GraphQL non authentifié. Vous pouvez faire pivoter les clés d'API depuis la console, depuis la CLI ou depuis la référence AWS AppSync d'API.

Console

1. Connectez-vous à la AppSyncconsole AWS Management Console et ouvrez-la.
   a. Dans le tableau de bord des API, choisissez votre API GraphQL.
   b. Dans la barre latérale, choisissez Réglages.
2. Sous Mode d'autorisation par défaut, choisissez la clé API.
3. Dans le tableau des clés d'API, choisissez Ajouter une clé d'API.

   Une nouvelle clé d'API sera générée dans le tableau.

   - Pour supprimer une ancienne clé d'API, sélectionnez la clé d'API dans le tableau, puis choisissez Supprimer.
4. En bas de la page, sélectionnez Save (Enregistrer).

CLI

1. Si ce n'est pas déjà fait, configurez votre accès à la AWS CLI. Pour plus d'informations, consultez la section Principes de base de la configuration.
2. Créez un objet d'API GraphQL en exécutant la `update-graphql-api`commande.

   Vous devez saisir deux paramètres pour cette commande particulière :

1. Celui `api-id` de votre API GraphQL.
2. La nouveauté `name` de votre API. Vous pouvez utiliser le même`name`.
3. Le`authentication-type`, qui sera`API_KEY`.

> **Note**
>
> Il existe d'autres paramètres tels `Region` que ceux qui doivent être configurés, mais ils seront généralement définis par défaut sur les valeurs de configuration de votre CLI.

Voici un exemple de commande :

```
aws appsync update-graphql-api --api-id abcdefghijklmnopqrstuvwxyz --name
 TestAPI --authentication-type API_KEY
```

Une sortie sera renvoyée dans la CLI. Voici un exemple en JSON :

```
{
    "graphqlApi": {
        "xrayEnabled": false,
        "name": "TestAPI",
        "authenticationType": "API_KEY",
        "tags": {},
        "apiId": "abcdefghijklmnopqrstuvwxyz",
        "uris": {
            "GRAPHQL": "https://s8i3kk3ufhe9034ujnv73r513e.appsync-api.us-
west-2.amazonaws.com/graphql",
            "REALTIME": "wss://s8i3kk3ufhe9034ujnv73r513e.appsync-realtime-
api.us-west-2.amazonaws.com/graphql"
        },
        "arn": "arn:aws:appsync:us-west-2:348581070237:apis/
abcdefghijklmnopqrstuvwxyz"
    }
}
```

Les clés API sont configurables pour une durée maximale de 365 jours et vous pouvez prolonger une date d'expiration existante de 365 jours maximum à partir de cette date. Les clés API sont recommandées pour le développement ou pour les cas d'utilisation où il est possible d'exposer une API publique en toute sécurité.

Sur le client, la clé API est spécifiée par l'en-tête `x-api-key`.

Par exemple, si votre `API_KEY` a pour valeur `'ABC123'`, vous pouvez envoyer une requête GraphQL via `curl` comme suit :

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "x-api-key:ABC123" -d
 '{ "query": "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

Autorisation AWS_LAMBDA

Vous pouvez implémenter votre propre logique d'autorisation d'API à l'aide d'une AWS Lambda fonction. Vous pouvez utiliser une fonction Lambda pour votre autorisateur principal ou secondaire, mais il ne peut y avoir qu'une seule fonction d'autorisation Lambda par API. Lorsque vous utilisez des fonctions Lambda pour l'autorisation, les règles suivantes s'appliquent :

- Si les modes d'`AWS_IAM`autorisation `AWS_LAMBDA` et sont activés dans l'API, la signature SigV4 ne peut pas être utilisée comme jeton `AWS_LAMBDA` d'autorisation.
- Si les modes `OPENID_CONNECT` d'autorisation `AWS_LAMBDA` et ou le mode d'autorisation sont activés sur l'`AMAZON_COGNITO_USER_POOLS`API, le jeton OIDC ne peut pas être utilisé comme jeton `AWS_LAMBDA` d'autorisation. Notez que le jeton OIDC peut être un schéma Bearer.
- Une fonction Lambda ne doit pas renvoyer plus de 5 Mo de données contextuelles pour les résolveurs.

Par exemple, si votre jeton d'autorisation l'est `'ABC123'`, vous pouvez envoyer une requête GraphQL via curl comme suit :

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "Authorization:ABC123" -d
 '{ "query":
        "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

Les fonctions Lambda sont appelées avant chaque requête ou mutation. La valeur de retour peut être mise en cache en fonction de l'ID d'API et du jeton d'authentification. Par défaut, la mise en cache

n'est pas activée, mais elle peut être activée au niveau de l'API ou en définissant la `ttlOverride` valeur dans la valeur de retour d'une fonction.

Une expression régulière qui valide les jetons d'autorisation avant l'appel de la fonction peut être spécifiée si vous le souhaitez. Ces expressions régulières sont utilisées pour valider qu'un jeton d'autorisation est au bon format avant que votre fonction ne soit appelée. Toute demande utilisant un jeton ne correspondant pas à cette expression régulière sera automatiquement refusée.

Les fonctions Lambda utilisées pour l'autorisation nécessitent qu'une politique principale leur soit appliquée `appsync.amazonaws.com` pour permettre de les AWS AppSync appeler. Cette action est effectuée automatiquement dans la AWS AppSync console ; la AWS AppSync console ne supprime pas la politique. Pour plus d'informations sur l'attachement de politiques aux fonctions Lambda, consultez la section Politiques basées sur les ressources dans le Guide du développeur. AWS Lambda

La fonction Lambda que vous spécifiez recevra un événement ayant la forme suivante :

```
{
    "authorizationToken": "ExampleAUTHtoken123123123",
    "requestContext": {
        "apiId": "aaaaaa123123123example123",
        "accountId": "111122223333",
        "requestId": "f4081827-1111-4444-5555-5cf4695f339f",
        "queryString": "mutation CreateEvent {...}\n\nquery MyQuery {...}\n",
        "operationName": "MyQuery",
        "variables": {}
    }
    "requestHeaders": {
        application request headers
    }
}
```

L'`event`objet contient les en-têtes envoyés dans la demande par le client d'application à AWS AppSync.

La fonction d'autorisation doit renvoyer au moins `isAuthorized` un booléen indiquant si la demande est autorisée. AWS AppSync reconnaît les clés suivantes renvoyées par les fonctions d'autorisation Lambda :

Liste des fonctions

`isAuthorized`(booléen, obligatoire)

Une valeur booléenne indiquant si la valeur in `authorizationToken` est autorisée à effectuer des appels à l'API GraphQL.

Si cette valeur est vraie, l'exécution de l'API GraphQL continue. Si cette valeur est fausse, un `UnauthorizedException` est augmenté

`deniedFields`(liste de chaînes, facultatif)

Une liste dont la liste est modifiée de force`null`, même si une valeur a été renvoyée par un résolveur.

Chaque élément est soit un ARN de champ entièrement qualifié sous la forme de, `arn:aws:appsync:`*`us-east-1`*`:`*`111122223333`*`:apis/`*`GraphQLApiId`*`/types/`*`TypeName`*`/fields/`*`FieldName`* soit une forme abrégée de*`TypeName`*`.`*`FieldName`*. Le formulaire ARN complet doit être utilisé lorsque deux API partagent un autorisateur de fonction Lambda et qu'il peut y avoir une ambiguïté entre les types et les champs communs entre les deux API.

`resolverContext`(Objet JSON, facultatif)

Un objet JSON visible comme `$ctx.identity.resolverContext` dans les modèles de résolveur. Par exemple, si la structure suivante est renvoyée par un résolveur :

```
{
  "isAuthorized":true
  "resolverContext": {
    "banana":"very yellow",
    "apple":"very green"
  }
}
```

La valeur des modèles intégrés `ctx.identity.resolverContext.apple` au résolveur sera « `very green` ». L'`resolverContext`objet ne prend en charge que les paires clé-valeur. Les clés imbriquées ne sont pas prises en charge.

> **Warning**
>
> La taille totale de cet objet JSON ne doit pas dépasser 5 Mo.

`ttlOverride`(entier, facultatif)

Le nombre de secondes pendant lesquelles la réponse doit être mise en cache. Si aucune valeur n'est renvoyée, la valeur de l'API est utilisée. S'il s'agit de 0, la réponse n'est pas mise en cache.

Les autorisateurs Lambda ont un délai d'expiration de 10 secondes. Nous vous recommandons de concevoir des fonctions à exécuter le plus rapidement possible afin d'optimiser les performances de votre API.

Plusieurs AWS AppSync API peuvent partager une seule fonction Lambda d'authentification. L'utilisation d'autorisations entre comptes n'est pas autorisée.

Lorsque vous partagez une fonction d'autorisation entre plusieurs API, sachez que les noms de champs abrégés (*`typename`*`.`*`fieldname`*) peuvent masquer des champs par inadvertance. Pour lever l'ambiguïté d'un champ dans`deniedFields`, vous pouvez spécifier un ARN de champ non ambigu sous la forme de. `arn:aws:appsync:`*`region`*`:`*`accountId`*`:apis/`*`GraphQLApiId`*`/types/`*`typeName`*`/fields/`*`fieldName`*

Pour ajouter une fonction Lambda comme mode d'autorisation par défaut dans : AWS AppSync

Console

1. Connectez-vous à la AWS AppSync console et accédez à l'API que vous souhaitez mettre à jour.
2. Accédez à la page des paramètres de votre API.

   Modifiez l'autorisation au niveau de l'API en. AWS Lambda
3. Choisissez l'ARN Région AWS et Lambda par rapport auxquels autoriser les appels d'API.

   > **Note**
   >
   > La politique principale appropriée sera ajoutée automatiquement, ce qui vous permettra AWS AppSync d'appeler votre fonction Lambda.
4. Définissez éventuellement le TTL de réponse et l'expression régulière de validation du jeton.

AWS CLI

1. Associez la politique suivante à la fonction Lambda utilisée :

```
aws lambda add-permission --function-name "my-function" --statement-id "appsync"
 --principal appsync.amazonaws.com --action lambda:InvokeFunction --output text
```

> **Important**
>
> Si vous souhaitez que la politique de la fonction soit limitée à une seule API GraphQL, vous pouvez exécuter cette commande :
>
> ```
> aws lambda add-permission --function-name “my-function” --
> statement-id “appsync” --principal appsync.amazonaws.com --action
>  lambda:InvokeFunction --source-arn “<my AppSync API ARN>” --output text
> ```

2. Mettez à jour votre AWS AppSync API pour utiliser l'ARN de la fonction Lambda donné comme autorisateur :

```
aws appsync update-graphql-api --api-id example2f0ur2oid7acexample --
name exampleAPI --authentication-type AWS_LAMBDA --lambda-authorizer-config
 authorizerUri="arn:aws:lambda:us-east-2:111122223333:function:my-function"
```

> **Note**
>
> Vous pouvez également inclure d'autres options de configuration, telles que l'expression régulière du jeton.

L'exemple suivant décrit une fonction Lambda qui illustre les différents états d'authentification et de défaillance qu'une fonction Lambda peut avoir lorsqu'elle est utilisée comme mécanisme d'autorisation : AWS AppSync

```
def handler(event, context):
  # This is the authorization token passed by the client
  token = event.get('authorizationToken')
  # If a lambda authorizer throws an exception, it will be treated as unauthorized.
  if 'Fail' in token:
    raise Exception('Purposefully thrown exception in Lambda Authorizer.')

  if 'Authorized' in token and 'ReturnContext' in token:
```

```
    return {
      'isAuthorized': True,
      'resolverContext': {
        'key': 'value'
      }
    }

  # Authorized with no f
  if 'Authorized' in token:
    return {
      'isAuthorized': True
    }
  # Partial authorization
  if 'Partial' in token:
    return {
      'isAuthorized': True,
      'deniedFields':['user.favoriteColor']
    }
  if 'NeverCache' in token:
    return {
      'isAuthorized': True,
      'ttlOverride': 0
    }
  if 'Unauthorized' in token:
    return {
      'isAuthorized': False
    }
  # if nothing is returned, then the authorization fails.
  return {}
```

Contourner les limites d'autorisation des jetons SigV4 et OIDC

Les méthodes suivantes peuvent être utilisées pour contourner le problème de l'impossibilité d'utiliser votre signature SigV4 ou votre jeton OIDC comme jeton d'autorisation Lambda lorsque certains modes d'autorisation sont activés.

Si vous souhaitez utiliser la signature SigV4 comme jeton d'autorisation Lambda lorsque `AWS_IAM` les modes d'autorisation `AWS_LAMBDA` et sont activés AWS AppSync pour l'API, procédez comme suit :

- Pour créer un nouveau jeton d'autorisation Lambda, ajoutez des suffixes et/ou des préfixes aléatoires à la signature SigV4.

- Pour récupérer la signature SigV4 d'origine, mettez à jour votre fonction Lambda en supprimant les préfixes et/ou suffixes aléatoires du jeton d'autorisation Lambda. Utilisez ensuite la signature SigV4 d'origine pour l'authentification.

Si vous souhaitez utiliser le jeton OIDC comme jeton d'autorisation Lambda lorsque `OPENID_CONNECT` le mode d'autorisation ou `AMAZON_COGNITO_USER_POOLS` les modes d'autorisation `AWS_LAMBDA` et sont activés AWS AppSync pour l'API, procédez comme suit :

- Pour créer un nouveau jeton d'autorisation Lambda, ajoutez des suffixes et/ou des préfixes aléatoires au jeton OIDC. Le jeton d'autorisation Lambda ne doit pas contenir de préfixe de schéma Bearer.
- Pour récupérer le jeton OIDC d'origine, mettez à jour votre fonction Lambda en supprimant les préfixes et/ou suffixes aléatoires du jeton d'autorisation Lambda. Utilisez ensuite le jeton OIDC d'origine pour l'authentification.

Autorisation AWS_IAM

Ce type d'autorisation applique le processus de AWS signature de la version 4 de la signature sur l'API GraphQL. Vous pouvez associer les stratégies d'accès Identity and Access Management (IAM) à ce type d'autorisation. Votre application peut tirer parti de cette association en utilisant une clé d'accès (composée d'un identifiant de clé d'accès et d'une clé d'accès secrète) ou en utilisant des informations d'identification temporaires de courte durée fournies par Amazon Cognito Federated Identities.

Si vous souhaitez un rôle pouvant effectuer toutes les opérations de données :

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "appsync:GraphQL"
         ],
         "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/*"
         ]
      }
   ]
```

```
}
```

Vous pouvez le trouver sur la page principale `YourGraphQLApiId` de liste des API de la AppSync console, directement sous le nom de votre API. Vous pouvez aussi récupérer cet élément via l'interface de ligne de commande : `aws appsync list-graphql-apis`

Si vous souhaitez limiter l'accès à un nombre limité d'opérations GraphQL, vous pouvez le faire pour les champs racine `Query`, `Mutation` et `Subscription`.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "appsync:GraphQL"
         ],
         "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Query/
fields/<Field-1>",
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Query/
fields/<Field-2>",
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/
Mutation/fields/<Field-1>",
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/
Subscription/fields/<Field-1>"
         ]
     }
   ]
}
```

Par exemple, supposons que vous ayez le schéma suivant et que vous souhaitiez limiter l'accès à l'obtention de tous les billets de blog :

```
schema {
   query: Query
   mutation: Mutation
}

type Query {
   posts:[Post!]!
}
```

```
type Mutation {
   addPost(id:ID!, title:String!):Post!
}
```

La politique IAM correspondante pour un rôle (que vous pourriez associer à un pool d'identités Amazon Cognito, par exemple) se présente comme suit :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
            "appsync:GraphQL"
            ],
            "Resource": [
                "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/
Query/fields/posts"
            ]
        }
    ]
}
```

Autorisation OPENID_CONNECT

Ce type d'autorisation applique les jetons OpenID connect (OIDC) fournis par un service conforme à l'OIDC. Votre application peut tirer parti des utilisateurs et des privilèges définis par votre fournisseur OIDC pour le contrôle des accès.

L'URL de l'émetteur est la seule valeur de configuration requise que vous fournissez à AWS AppSync (par exemple,`https://auth.example.com`). Cette URL doit être adressable via HTTPS. AWS AppSync s'ajoute `/.well-known/openid-configuration` à l'URL de l'émetteur et localise la configuration OpenID conformément à la spécification OpenID Connect `https://auth.example.com/.well-known/openid-configuration` Discovery. Vous devez normalement récupérer un document JSON compatible avec la norme RFC5785 au niveau de cette URL. Ce document JSON doit contenir une `jwks_uri` clé pointant vers le document JSON Web Key Set (JWKS) contenant les clés de signature. AWS AppSync nécessite que le JWKS contienne les champs JSON de etkty. `kid`

AWS AppSync prend en charge un large éventail d'algorithmes de signature.

Algorithmes de signature
RS256
RS384
RS512
PS256
PS384
PS512
HS256
HS384
HS512
ES256
ES384
ES512

Nous vous recommandons d'utiliser les algorithmes RSA. Les jetons émis par le fournisseur doivent inclure l'heure d'émission du jeton (`iat`) et peuvent inclure son heure d'authentification (`auth_time`). Vous pouvez fournir des valeurs de durée de vie (TTL) pour l'heure d'émission (`iatTTL`) et l'heure d'authentification (`authTTL`) dans votre configuration OpenID Connect afin de renforcer la validation. Si votre fournisseur autorise plusieurs applications, vous pouvez également fournir une expression régulière (`clientId`) qui est utilisée pour les autorisations par ID client. Lorsque le `clientId` est présent dans votre configuration OpenID Connect, AWS AppSync valide la réclamation en demandant qu'il corresponde `clientId` à la réclamation `aud` ou à la `azp` réclamation figurant dans le jeton.

Pour valider plusieurs identifiants clients, utilisez l'opérateur de pipeline (« | ») qui est un « ou » dans une expression régulière. Par exemple, si votre application OIDC possède quatre clients avec des

identifiants clients tels que 0A1S2D, 1F4G9H, 1J6L4B, 6GS5MG, pour valider uniquement les trois premiers ID client, vous devez placer 1F4G9H|1J6L4B|6GS5MG dans le champ ID client.

Autorisation AMAZON_COGNITO_USER_POOLS

Ce type d'autorisation applique les jetons OIDC fournis par les groupes d'utilisateurs Amazon Cognito. Votre application peut exploiter les utilisateurs et les groupes de vos groupes d'utilisateurs et ceux d'un autre AWS compte et les associer à des champs GraphQL pour contrôler l'accès.

Lorsque vous utilisez les groupes d'utilisateurs Amazon Cognito, vous pouvez créer des groupes auxquels les utilisateurs appartiennent. Ces informations sont codées dans un jeton JWT que votre application envoie AWS AppSync dans un en-tête d'autorisation lors de l'envoi d'opérations GraphQL. Vous pouvez utiliser les directives GraphQL sur le schéma pour contrôler les groupes pouvant appeler des résolveurs sur un champ, ainsi que les résolveurs pouvant être appelés, ce qui permet à vos clients de bénéficier d'un accès plus contrôlé.

Supposons, par exemple, que vous ayez le schéma GraphQL suivant :

```
schema {
   query: Query
   mutation: Mutation
}

type Query {
   posts:[Post!]!
}

type Mutation {
   addPost(id:ID!, title:String!):Post!
}
...
```

Si vous avez deux groupes dans les groupes d'utilisateurs d'Amazon Cognito (les blogueurs et les lecteurs) et que vous souhaitez restreindre le nombre de lecteurs afin qu'ils ne puissent pas ajouter de nouvelles entrées, votre schéma doit ressembler à ceci :

```
schema {
   query: Query
   mutation: Mutation
}
```

```
type Query {
   posts:[Post!]!
   @aws_auth(cognito_groups: ["Bloggers", "Readers"])
}

type Mutation {
   addPost(id:ID!, title:String!):Post!
   @aws_auth(cognito_groups: ["Bloggers"])
}
...
```

Notez que vous pouvez omettre la `@aws_auth` directive si vous souhaitez utiliser par défaut une grant-or-deny stratégie d'accès spécifique. Vous pouvez spécifier la grant-or-deny stratégie dans la configuration du groupe d'utilisateurs lorsque vous créez votre API GraphQL via la console ou via la commande CLI suivante :

```
$ aws appsync --region us-west-2 create-graphql-api --authentication-
type AMAZON_COGNITO_USER_POOLS  --name userpoolstest --user-pool-config
 '{ "userPoolId":"test", "defaultEffect":"ALLOW", "awsRegion":"us-west-2"}'
```

Utilisation de modes d'autorisation supplémentaires

Lorsque vous ajoutez des modes d'autorisation supplémentaires, vous pouvez configurer directement le paramètre d'autorisation au niveau de l'API AWS AppSync GraphQL (c'est-à-dire le `authenticationType` champ que vous pouvez configurer directement sur l'`GraphqlApi`objet) et il agit comme paramètre par défaut sur le schéma. Cela signifie que tout type n'ayant pas de directive spécifique doit transmettre le paramètre d'autorisation au niveau de l'API.

Au niveau du schéma, vous pouvez spécifier des modes d'autorisation supplémentaires à l'aide de directives sur le schéma. Vous pouvez spécifier des modes d'autorisation sur des champs spécifiques du schéma. Par exemple, pour l'autorisation `API_KEY`, vous devez utiliser `@aws_api_key` sur les définitions/champs de type d'objet de schéma. Les directives suivantes sont prises en charge sur les champs de schéma et les définitions de types d'objet :

- `@aws_api_key` : permet de spécifier que le champ a l'autorisation `API_KEY`.
- `@aws_iam` : permet de spécifier que le champ a l'autorisation `AWS_IAM`.
- `@aws_oidc` : permet de spécifier que le champ a l'autorisation `OPENID_CONNECT`.

- `@aws_cognito_user_pools` : permet de spécifier que le champ a l'autorisation `AMAZON_COGNITO_USER_POOLS`.
- `@aws_lambda` : permet de spécifier que le champ a l'autorisation `AWS_LAMBDA`.

Vous ne pouvez pas utiliser la directive `@aws_auth` avec des modes d'autorisation supplémentaires. `@aws_auth` fonctionne uniquement dans le contexte de l'autorisation `AMAZON_COGNITO_USER_POOLS` sans mode d'autorisation supplémentaire. Cependant, vous pouvez utiliser la directive `@aws_cognito_user_pools` à la place de la directive `@aws_auth`, en utilisant les mêmes arguments. La principale différence entre les deux est que vous pouvez spécifier `@aws_cognito_user_pools` sur n'importe quelle définition de champ et de type d'objet.

Pour comprendre comment fonctionnent les modes d'autorisation supplémentaires et comment ils peuvent être spécifiés sur un schéma, examinons le schéma suivant :

```
schema {
   query: Query
   mutation: Mutation
}

type Query {
   getPost(id: ID): Post
   getAllPosts(): [Post]
   @aws_api_key
}

type Mutation {
   addPost(
      id: ID!
      author: String!
      title: String!
      content: String!
      url: String!
   ): Post!
}

type Post @aws_api_key @aws_iam {
   id: ID!
   author: String
   title: String
   content: String
   url: String
```

```
   ups: Int!
   downs: Int!
   version: Int!
}
...
```

Pour ce schéma, supposons qu'il `AWS_IAM` s'agit du type d'autorisation par défaut sur l'API AWS AppSync GraphQL. Cela signifie que les champs qui n'ont pas de directive sont protégés à l'aide de `AWS_IAM`. Par exemple, c'est le cas pour le champ `getPost` sur le type `Query`. Les directives de schéma vous permettent d'utiliser plusieurs modes d'autorisation. Par exemple, vous pouvez avoir `API_KEY` configuré un mode d'autorisation supplémentaire sur l'API AWS AppSync GraphQL, et vous pouvez marquer un champ à l'aide de la `@aws_api_key` directive (par exemple, `getAllPosts` dans cet exemple). Les directives fonctionnent au niveau du champ. Vous devez donc également accorder à `API_KEY` l'accès au type `Post`. Vous pouvez le faire en marquant chaque champ du type `Post` avec une directive ou en marquant le type `Post` avec la directive `@aws_api_key`.

Pour limiter davantage l'accès aux champs du type `Post`, vous pouvez utiliser des directives sur les champs individuels du type `Post`, comme illustré ci-après.

Par exemple, vous pouvez ajouter un champ `restrictedContent` au type `Post` et limiter l'accès à celui-ci à l'aide de la directive `@aws_iam`. Les demandes `AWS_IAM` authentifiées pourront accéder à `restrictedContent`, mais les demandes `API_KEY` ne pourront pas y accéder.

```
type Post @aws_api_key @aws_iam{
   id: ID!
   author: String
   title: String
   content: String
   url: String
   ups: Int!
   downs: Int!
   version: Int!
   restrictedContent: String!
   @aws_iam
}
...
```

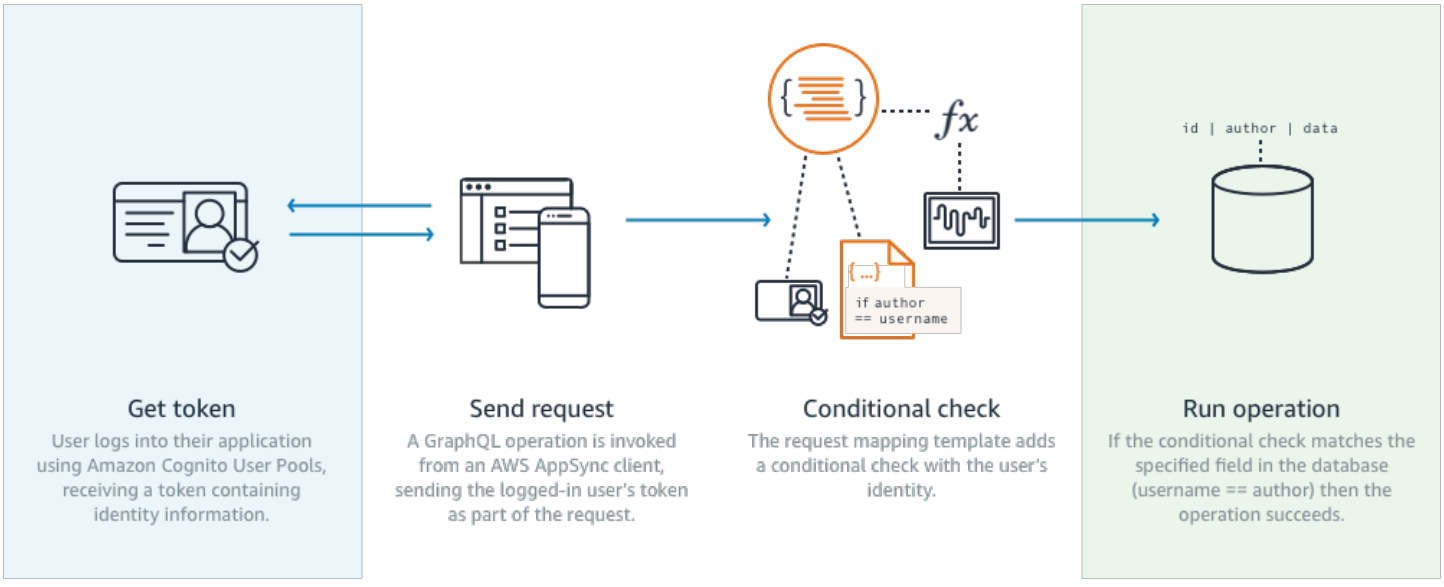
Contrôle précis des accès

Les informations précédentes montrent comment limiter ou accorder l'accès à certains champs GraphQL. Si vous voulez définir des contrôles d'accès sur les données en fonction de certaines conditions (par exemple, en fonction de l'utilisateur effectuant l'appel et du fait qu'il est ou non propriétaire des données), vous pouvez utiliser des modèles de mappage dans vos résolveurs. Vous pouvez également utiliser une logique métier plus complexe, dont vous trouverez la description dans Filtrage des informations.

Cette section explique comment définir les contrôles d'accès à vos données à l'aide d'un modèle de mappage du résolveur DynamoDB.

Avant de poursuivre, si vous n'êtes pas familiarisé avec les modèles de mappage dans AWS AppSync, vous pouvez consulter la référence du modèle de mappage Resolver et la référence du modèle de mappage Resolver pour DynamoDB.

Dans l'exemple suivant utilisant DynamoDB, supposons que vous utilisiez le schéma de billet de blog précédent et que seuls les utilisateurs ayant créé un article soient autorisés à le modifier. Le processus d'évaluation devrait consister pour l'utilisateur à obtenir des informations d'identification dans son application, à l'aide de groupes d'utilisateurs Amazon Cognito, par exemple, puis à transmettre ces informations d'identification dans le cadre d'une opération GraphQL. Le modèle de mappage remplace alors une valeur des informations d'identification (comme le nom utilisateur) dans une instruction conditionnelle qui sera ensuite comparée à une valeur dans votre base de données.



Pour ajouter cette fonctionnalité, ajoutez un champ GraphQL `editPost` comme suit :

```
schema {
   query: Query
   mutation: Mutation
}

type Query {
   posts:[Post!]!
}

type Mutation {
   editPost(id:ID!, title:String, content:String):Post
   addPost(id:ID!, title:String!):Post!
}
...
```

Le modèle de mappage de résolveur pour `editPost` (illustré dans un exemple à la fin de cette section) doit effectuer une vérification logique par rapport à votre magasin de données afin que seul l'utilisateur ayant créé un billet puisse le modifier. Comme il s'agit d'une opération de modification, elle correspond à une opération `UpdateItem` dans DynamoDB. Vous pouvez effectuer une vérification conditionnelle avant d'exécuter cette action, en utilisant le contexte transmis pour la validation de l'identité de l'utilisateur. Ce dernier se trouve dans un objet `Identity` qui a les valeurs suivantes :

```
{
   "accountId" : "12321434323",
   "cognitoIdentityPoolId" : "",
   "cognitoIdentityId" : "",
   "sourceIP" : "",
   "caller" : "ThisistheprincipalARN",
   "username" : "username",
   "userArn" : "Sameasabove"
}
```

Pour utiliser cet objet dans un appel DynamoDB`UpdateItem`, vous devez enregistrer les informations d'identité de l'utilisateur dans le tableau à des fins de comparaison. Tout d'abord, votre mutation `addPost` doit contenir le créateur. Ensuite, votre mutation `editPost` doit effectuer la vérification conditionnelle avant la mise à jour.

Voici un exemple de code de résolution `addPost` qui stocke l'identité de l'utilisateur sous forme de `Author` colonne :

```
import { util, Context } from '@aws-appsync/utils';
import { put } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const { id: postId, ...item } = ctx.args;
 return put({
  key: { postId },
  item: { ...item, Author: ctx.identity.username },
  condition: { postId: { attributeExists: false } },
 });
}

export const response = (ctx) => ctx.result;
```

Notez que l'attribut `Author` est renseigné à partir de l'objet `Identity`, qui provient de l'application.

Enfin, voici un exemple de code de résolution pour`editPost`, qui met à jour le contenu du billet de blog uniquement si la demande provient de l'utilisateur qui a créé le billet :

```
import { util, Context } from '@aws-appsync/utils';
import { put } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const { id, ...item } = ctx.args;
 return put({
  key: { id },
  item,
  condition: { author: { contains: ctx.identity.username } },
 });
}

export const response = (ctx) => ctx.result;
```

Cet exemple utilise un `PutItem` qui remplace toutes les valeurs plutôt que an`UpdateItem`, mais le même concept s'applique au bloc d'`condition`instructions.

Filtrer les informations

Il peut arriver que vous ne puissiez pas contrôler la réponse de votre source de données, mais que vous ne souhaitiez pas envoyer des informations inutiles aux clients sur la réussite d'une opération

de lecture ou d'écriture sur la source de données. Dans ce cas, vous pouvez filtrer les informations à l'aide d'un modèle de mappage de réponse.

Supposons, par exemple, que vous ne disposiez pas d'un index approprié dans la table DynamoDB de votre billet de blog (tel qu'un index sur). `Author` Vous pouvez utiliser le résolveur suivant :

```
import { util, Context } from '@aws-appsync/utils';
import { get } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 return get({ key: { ctx.args.id } });
}

export function response(ctx) {
 if (ctx.result.author === ctx.identity.username) {
  return ctx.result;
 }
 return null;
}
```

Le gestionnaire de demandes récupère l'élément même si l'appelant n'est pas l'auteur qui a créé le message. Pour éviter que cela ne renvoie toutes les données, le gestionnaire de réponses vérifie que l'appelant correspond à l'auteur de l'article. Si l'appelant ne correspond pas à ce contrôle, seule une réponse null est renvoyée.

Accès à une source de données

AWS AppSync communique avec les sources de données à l'aide des rôles Identity and Access Management (IAM) et des politiques d'accès. Si vous utilisez un rôle existant, une politique de confiance doit être ajoutée AWS AppSync pour pouvoir assumer le rôle. La relation d'approbation doit ressembler à ce qui suit :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "appsync.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
```

```
        }
    ]
}
```

Il est important de définir la stratégie d'accès en fonction du rôle, afin de n'avoir que les autorisations nécessaires pour agir sur l'ensemble minimal de ressources nécessaires. Lorsque vous utilisez la AppSync console pour créer une source de données et créer un rôle, cela se fait automatiquement pour vous. Toutefois, lorsque vous utilisez un exemple de modèle intégré à partir de la console IAM pour créer un rôle en dehors de la AWS AppSync console, les autorisations ne seront pas automatiquement limitées à une ressource et vous devez effectuer cette action avant de mettre votre application en production.

Cas d'utilisation de l'autorisation

Dans la section Sécurité, vous avez découvert les différents modes d'autorisation pour protéger votre API et avez bénéficié d'une présentation des mécanismes d'autorisation granulaire pour comprendre les concepts et le flux. Comme il vousAWS AppSync permet d'effectuer des opérations logiques complètes sur les données grâce à l'utilisation de modèles de mappage GraphQL Resolver, vous pouvez protéger les données en lecture ou en écriture de manière très flexible en combinant identité utilisateur, conditions et injection de données.

Si vous n'êtes pas familiarisé avec la modification des résolveurs AWS AppSync , passez en revue le guide de programmation.

Présentation

L'accès aux données d'un système se fait traditionnellement par le biais d'une matrice de contrôle d'accès où l'intersection d'une ligne (ressource) et d'une colonne (utilisateur/rôle) correspond aux autorisations accordées.

AWS AppSync utilise les ressources de votre propre compte et intègre les informations d'identité (utilisateur/rôle) dans la requête et la réponse GraphQL en tant qu'objet de contexte, que vous pouvez utiliser dans le résolveur. Cela signifie que les autorisations peuvent être accordées de façon appropriée sur des opérations de lecture ou d'écriture en fonction de la logique du résolveur. Si cette logique se situe au niveau des ressources, par exemple, seuls certains utilisateurs ou groupes nommés peuvent lire/écrire sur une ligne de base de données spécifique, ces « métadonnées d'autorisation » doivent être stockées. AWS AppSync ne stocke aucune donnée, vous devez donc stocker ces métadonnées d'autorisation avec les ressources afin que les autorisations puissent être calculées. Les métadonnées d'autorisation sont généralement un attribut (colonne) d'une table

DynamoDB, tel qu'un propriétaire ou une liste d'utilisateurs/groupes. Par exemple, il pourrait y avoir des attributs Readers et Writers.

Depuis un niveau élevé, cela signifie que si vous lisez un élément individuel à partir d'une source de données, vous devez effectuer une déclaration conditionnelle `#if () ... #end` dans le modèle de réponse après que le résolveur a lu à partir de la source de données. Le contrôle utilisera normalement les valeurs d'utilisateur ou de groupe dans `$context.identity` pour les contrôles d'appartenance par rapport aux métadonnées d'autorisation renvoyées par une opération de lecture. Pour plusieurs enregistrements, tels que les listes renvoyées à partir d'une table `Scan` ou `Query`, vous envoyez le contrôle de la condition à la source de données, comme partie intégrante de l'opération, à l'aide de valeurs d'utilisateur ou de groupe similaires.

De même, lors de l'écriture des données, vous appliquerez une instruction conditionnelle à l'action (comme un `PutItem` ou `UpdateItem` pour voir si l'utilisateur ou le groupe effectuant la mutation a l'autorisation). L'instruction conditionnelle utilisera à nouveau à plusieurs reprises une valeur dans `$context.identity` à titre de comparaison avec les métadonnées d'autorisation sur cette ressource. Pour les modèles de demande et de réponse, vous pouvez également utiliser des en-têtes personnalisés provenant des clients pour effectuer les contrôles de validation.

Lecture de données

Comme indiqué ci-dessus, les métadonnées d'autorisation pour effectuer un contrôle doivent être stockées avec une ressource ou transmises à la demande GraphQL (identité, en-tête, etc.). Pour illustrer cela, supposons que vous ayez la table DynamoDB ci-dessous :

ID	Data	PeopleCanAccess	GroupsCanAccess	Owner
123	{my: data,...}	[Mary, Joe]	[Admins, Editors]	Nadia

La clé primaire est `id` et les données qui doivent être accessibles `Data`. Les autres colonnes sont des exemples de vérifications que vous pouvez effectuer pour obtenir une autorisation. `Owner`prendrait un`String` certain temps`PeopleCanAccess` et`GroupsCanAccess` serait`String` `Sets` comme indiqué dans la référence du modèle de mappage Resolver pour DynamoDB.

Dans la présentation du modèle de mappage des résolveurs, le diagramme montre comment le modèle de réponse contient non seulement l'objet de contexte, mais aussi les résultats de la source de données. Pour les requêtes GraphQL des objets individuels, vous pouvez utiliser le modèle de réponse pour vérifier si l'utilisateur est autorisé à voir les résultats ou à renvoyer un message d'erreur

relatif à l'autorisation. Cet élément est parfois appelé « filtre d'autorisation ». Pour les requêtes GraphQL renvoyant des listes, à l'aide d'une requête (Query) ou d'une analyse (Scan), il est plus performant d'effectuer le contrôle sur le modèle de demande et de ne renvoyer les données que si une condition d'autorisation est remplie. L'implémentation se présente alors ainsi :

1. GetItem - contrôle d'autorisation pour les dossiers individuels. Fait à l'aide d'instructions `#if() ... #end`.
2. Opérations Query/Scan – Le contrôle d'autorisation est une déclaration `"filter": {"expression":...}`. Les contrôles courants concernent l'égalité (`attribute = :input`) ou la vérification de la présence d'une valeur dans une liste (`contains(attribute, :input)`).

Dans #2, l'attribut `attribute` des deux déclarations représente le nom de colonne de l'enregistrement dans une table, comme `Owner` dans notre exemple ci-dessus. Vous pouvez créer un alias à l'aide du signe # et utiliser `"expressionNames":{...}`, mais ce n'est pas obligatoire. L'élément `:input` est une référence à la valeur que vous comparez à l'attribut de base de données, que vous définissez dans `"expressionValues":{...}`. Voyez les exemples ci-dessous.

Cas d'utilisation : le propriétaire sait lire

À l'aide du tableau ci-dessus, si vous souhaitez uniquement renvoyer les données si `Owner == Nadia` dans le cas d'une opération de lecture (`GetItem`), votre modèle se présente comme suit :

```
#if($context.result["Owner"] == $context.identity.username)
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

Quelques remarques à mentionner ici seront réutilisées dans les sections restantes. Tout d'abord, le contrôle utilise`$context.identity.username` le nom convivial d'inscription de l'utilisateur si les groupes d'utilisateurs Amazon Cognito sont utilisés et l'identité de l'utilisateur si l'IAM est utilisé (y compris les identités fédérées Amazon Cognito). Il existe d'autres valeurs à enregistrer pour un propriétaire, telles que la valeur unique « identité Amazon Cognito », qui est utile lors de la fédération des connexions depuis plusieurs emplacements. Vous devriez consulter les options disponibles dans la référence contextuelle du modèle de mappage du résolveur.

Deuxièmement, le contrôle conditionnel « else » répondant avec `$util.unauthorized()` est totalement facultatif, mais recommandé comme bonne pratique lors de la conception de votre API GraphQL.

Cas d'utilisation : accès spécifique au hardcode

```
// This checks if the user is part of the Admin group and makes the call
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #if($group == "Admin")
        #set($inCognitoGroup = true)
    #end
#end
#if($inCognitoGroup)
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "attributeValues" : {
        "owner" : $util.dynamodb.toDynamoDBJson($context.identity.username)
        #foreach( $entry in $context.arguments.entrySet() )
            ,"${entry.key}" : $util.dynamodb.toDynamoDBJson($entry.value)
        #end
    }
}
#else
    $utils.unauthorized()
#end
```

Cas d'utilisation : filtrage d'une liste de résultats

Dans l'exemple précédent, vous avez été en mesure de contrôler directement `$context.result` tandis qu'il renvoyait un seul élément ; cependant, certaines opérations telles qu'une analyse renvoient plusieurs éléments dans `$context.result.items` où vous devez exécuter le filtre d'autorisation de filtre et retourner uniquement les résultats que l'utilisateur est autorisé à afficher. Supposons que`owner` l'IdentityID Amazon Cognito soit défini cette fois sur l'enregistrement, vous pouvez ensuite utiliser le modèle de mappage des réponses suivant pour filtrer afin d'afficher uniquement les enregistrements appartenant à l'utilisateur :

```
#set($myResults = [])
```

```
#foreach($item in $context.result.items)
    ##For userpools use $context.identity.username instead
    #if($item.Owner == $context.identity.cognitoIdentityId)
        #set($added = $myResults.add($item))
    #end
#end
$utils.toJson($myResults)
```

Cas d'utilisation : plusieurs personnes peuvent lire

Une autre option d'autorisation populaire consiste à autoriser un groupe de personnes à pouvoir lire les données. Dans l'exemple ci-dessous, `"filter":{"expression":...}` renvoie uniquement les valeurs d'une analyse de table si l'utilisateur exécutant la requête GraphQL est répertorié dans l'ensemble `PeopleCanAccess`.

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "limit": #if(${context.arguments.count}) $util.toJson($context.arguments.count)
 #else 20 #end,
    "nextToken": #if(${context.arguments.nextToken})
 $util.toJson($context.arguments.nextToken) #else null #end,
    "filter":{
        "expression": "contains(#peopleCanAccess, :value)",
        "expressionNames": {
                "#peopleCanAccess": "peopleCanAccess"
        },
        "expressionValues": {
                ":value": $util.dynamodb.toDynamoDBJson($context.identity.username)
        }
    }
}
```

Cas d'utilisation : le groupe peut lire

Comme pour le dernier scénario, il se peut que seules les personnes d'un ou de plusieurs groupes aient les droits pour lire certains éléments d'une base de données. L'utilisation de l'opération `"expression": "contains()"` est similaire ; cependant, le fait qu'un utilisateur puisse faire partie de ce qui doit être pris en compte dans l'appartenance à l'ensemble relève d'un OR logique de tous les groupes. Dans ce cas, nous créons une instruction `$expression` ci-dessous pour chaque groupe dont fait partie l'utilisateur, puis la transmettons au filtre :

```
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
    #set( $val = {})
    #set( $test = $val.put("S", $group))
    #set( $values = $expressionValues.put(":var$foreach.count", $val))
    #if ( $foreach.hasNext )
    #set( $expression = "${expression} OR" )
    #end
#end
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "limit": #if(${context.arguments.count}) $util.toJson($context.arguments.count)
 #else 20 #end,
    "nextToken": #if(${context.arguments.nextToken})
 $util.toJson($context.arguments.nextToken) #else null #end,
    "filter":{
        "expression": "$expression",
        "expressionValues": $utils.toJson($expressionValues)
    }
}
```

Écrire des données

L'écriture de données sur les mutations est toujours contrôlée sur le modèle de mappage de la demande. Dans le cas des sources de données DynamoDB, la clé consiste à utiliser un `"condition":{"expression"...}"` approprié, qui effectue la validation par rapport aux métadonnées d'autorisation de la table. Dans Sécurité, nous avons fourni un exemple que vous pouvez utiliser pour vérifier le champ `Author` dans une table. Les cas d'utilisation de cette section explorent d'autres scénarios.

Cas d'utilisation : plusieurs propriétaires

À l'aide de l'exemple de schéma de table précédent, imaginons la liste `PeopleCanAccess`

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
```

```
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "update" : {
        "expression" : "SET meta = :meta",
        "expressionValues": {
            ":meta" : $util.dynamodb.toDynamoDBJson($ctx.args.meta)
        }
    },
    "condition" : {
        "expression"       : "contains(Owner,:expectedOwner)",
        "expressionValues" : {
            ":expectedOwner" :
 $util.dynamodb.toDynamoDBJson($context.identity.username)
        }
    }
}
```

Cas d'utilisation : le groupe peut créer un nouvel enregistrement

```
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
    #set( $val = {})
    #set( $test = $val.put("S", $group))
    #set( $values = $expressionValues.put(":var$foreach.count", $val))
    #if ( $foreach.hasNext )
    #set( $expression = "${expression} OR" )
    #end
#end
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        ## If your table's hash key is not named 'id', update it here. **
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
        ## If your table has a sort key, add it as an item here. **
    },
    "attributeValues" : {
        ## Add an item for each field you would like to store to Amazon DynamoDB. **
        "title" : $util.dynamodb.toDynamoDBJson($ctx.args.title),
```

```
        "content": $util.dynamodb.toDynamoDBJson($ctx.args.content),
        "owner": $util.dynamodb.toDynamoDBJson($context.identity.username)
    },
    "condition" : {
        "expression": $util.toJson("attribute_not_exists(id) AND $expression"),
        "expressionValues": $utils.toJson($expressionValues)
    }
}
```

Cas d'utilisation : le groupe peut mettre à jour un enregistrement existant

```
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
    #set( $val = {})
    #set( $test = $val.put("S", $group))
    #set( $values = $expressionValues.put(":var$foreach.count", $val))
    #if ( $foreach.hasNext )
    #set( $expression = "${expression} OR" )
    #end
#end
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "update":{
                "expression" : "SET title = :title, content = :content",
        "expressionValues": {
            ":title" : $util.dynamodb.toDynamoDBJson($ctx.args.title),
            ":content" : $util.dynamodb.toDynamoDBJson($ctx.args.content)
        }
    },
    "condition" : {
        "expression": $util.toJson($expression),
        "expressionValues": $utils.toJson($expressionValues)
    }
}
```

Dossiers publics et privés

Avec les filtres conditionnels, vous pouvez également choisir de marquer les données comme privées, publiques ou booléennes. Elles peuvent ensuite être combinées dans le cadre d'un filtre d'autorisation à l'intérieur du modèle de réponse. L'utilisation de ce contrôle est un bon moyen de masquer les données temporairement sans tenter de contrôler l'appartenance au groupe.

Par exemple, imaginons que vous ayez ajouté un attribut sur chaque élément de votre table DynamoDB appelé `public` avec la valeur `yes` ou `no`. Le modèle de réponse suivant peut être utilisé sur un appel `GetItem` pour afficher les données uniquement si l'utilisateur se trouve dans un groupe qui a accès ET si ces données sont marquées comme publiques :

```
#set($permissions = $context.result.GroupsCanAccess)
#set($claimPermissions = $context.identity.claims.get("cognito:groups"))

#foreach($per in $permissions)
    #foreach($cgroups in $claimPermissions)
        #if($cgroups == $per)
            #set($hasPermission = true)
        #end
    #end
#end

#if($hasPermission && $context.result.public == 'yes')
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

Le code ci-dessus peut également utiliser un OR logique (`||`) pour autoriser les personnes à lire si elles sont autorisés à accéder à un enregistrement ou s'il est public :

```
#if($hasPermission || $context.result.public == 'yes')
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

En général, vous trouverez les opérateurs standard `==`, `!=`, `&&` et `||` utiles lors de l'exécution des contrôles d'autorisation.

Données en temps réel

Vous pouvez appliquer les contrôles d'accès détaillé aux abonnements GraphQL au moment où un client effectue un abonnement, en utilisant les mêmes techniques que celles décrites plus haut dans la documentation. Vous attachez un résolveur au champ d'abonnement et pouvez alors interroger les données à partir d'une source de données et exécuter une logique conditionnelle dans le modèle de mappage de la demande ou de la réponse. Vous pouvez également renvoyer des données supplémentaires au client, telles que les résultats initiaux d'un abonnement, aussi longtemps que la structure des données correspond à celle du type retourné dans votre abonnement GraphQL.

Cas d'utilisation : l'utilisateur peut s'abonner à des conversations spécifiques uniquement

Un cas d'utilisation courant pour les données en temps réel avec les abonnements GraphQL consiste à créer une application de messagerie ou chat privé. Lors de la création d'une application de chat qui comporte plusieurs utilisateurs, les conversations peuvent se produire entre deux ou plusieurs personnes. Celles-ci peuvent être regroupées en « salles », qui sont privées ou publiques. À ce titre, vous souhaitez uniquement autoriser un utilisateur à s'abonner à une conversation (qui pourrait être en face à face ou au sein d'un groupe) pour laquelle l'accès lui a été accordé. À des fins de démonstration, l'exemple ci-dessous illustre un simple scénario d'un utilisateur envoyant un message privé à un autre utilisateur. L'installation comporte deux tables Amazon DynamoDB :

- Table Messages : (clé primaire) `toUser`, (clé de tri) `id`
- Table Permissions : (clé primaire) `username`

La table Messages stocke les messages réels envoyés via une mutation GraphQL. La table Permissions est contrôlée par l'abonnement GraphQL pour l'autorisation au moment de la connexion du client. L'exemple suivant suppose que vous utilisez le schéma GraphQL suivant :

```
input CreateUserPermissionsInput {
    user: String!
    isAuthorizedForSubscriptions: Boolean
}

type Message {
    id: ID
    toUser: String
    fromUser: String
    content: String
```

```
}

type MessageConnection {
    items: [Message]
    nextToken: String
}

type Mutation {
    sendMessage(toUser: String!, content: String!): Message
    createUserPermissions(input: CreateUserPermissionsInput!): UserPermissions
    updateUserPermissions(input: UpdateUserPermissionInput!): UserPermissions
}

type Query {
    getMyMessages(first: Int, after: String): MessageConnection
    getUserPermissions(user: String!): UserPermissions
}

type Subscription {
    newMessage(toUser: String!): Message
        @aws_subscribe(mutations: ["sendMessage"])
}

input UpdateUserPermissionInput {
    user: String!
    isAuthorizedForSubscriptions: Boolean
}

type UserPermissions {
    user: String
    isAuthorizedForSubscriptions: Boolean
}

schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
}
```

Certaines opérations standard, telles que`createUserPermissions()`, ne sont pas abordées ci-dessous pour illustrer les résolveurs par abonnement, mais sont des implémentations standard des résolveurs DynamoDB. Au lieu de cela, nous allons nous concentrer sur les flux d'autorisation d'abonnement avec les résolveurs. Pour envoyer un message d'un utilisateur à un autre, attachez un

résolveur au champ `sendMessage()` et sélectionnez la source de données de la table Messages avec le modèle de demande suivant :

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "toUser" : $util.dynamodb.toDynamoDBJson($ctx.args.toUser),
        "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
    },
    "attributeValues" : {
        "fromUser" : $util.dynamodb.toDynamoDBJson($context.identity.username),
        "content" : $util.dynamodb.toDynamoDBJson($ctx.args.content),
    }
}
```

Dans cet exemple, nous utilisons `$context.identity.username`. Cela renvoie les informations utilisateur pour les utilisateursAWS Identity and Access Management d'Amazon Cognito. Le modèle de réponse est une simple transmission de `$util.toJson($ctx.result)`. Enregistrez et revenez à la page du schéma. Ensuite, attachez un résolveur pour l'abonnement `newMessage()`, à l'aide de la table Permissions comme source de données et du modèle de mappage de demande suivant :

```
{
    "version": "2018-05-29",
    "operation": "GetItem",
    "key": {
        "username": $util.dynamodb.toDynamoDBJson($ctx.identity.username),
    },
}
```

Ensuite, utilisez le modèle de mappage de réponse suivant pour exécuter vos contrôles d'autorisation à l'aide des données de la table Permissions :

```
#if(! ${context.result})
    $utils.unauthorized()
#elseif(${context.identity.username} != ${context.arguments.toUser})
    $utils.unauthorized()
#elseif(! ${context.result.isAuthorizedForSubscriptions})
    $utils.unauthorized()
#else
##User is authorized, but we return null to continue
```

```
    null
#end
```

Dans ce cas, vous effectuez trois contrôles d'autorisation. Le premier garantit qu'un résultat est retourné. Le second garantit que l'utilisateur ne s'abonne pas aux messages destinés à une autre personne. La troisième garantit que l'utilisateur est autorisé à s'abonner à n'importe quel champ, en cochant un attribut DynamoDB s'il est`isAuthorizedForSubscriptions` stocké sous forme de`BOOL`.

Pour tester les choses, vous pouvez vous connecter à laAWS AppSync console à l'aide des groupes d'utilisateurs Amazon Cognito et d'un utilisateur nommé « Nadia », puis exécuter l'abonnement GraphQL suivant :

```
subscription AuthorizedSubscription {
    newMessage(toUser: "Nadia") {
        id
        toUser
        fromUser
        content
    }
}
```

Si, dans la table Permissions, il y a un enregistrement pour l'attribut clé `username` de `Nadia` avec `isAuthorizedForSubscriptions` défini sur `true`, vous obtiendrez une réponse positive. Si vous essayez un autre `username` dans la requête `newMessage()` ci-dessus, une erreur est renvoyée.

Utilisation de AWS WAF pour protéger vos API

AWS WAF est un pare-feu d'application web qui aide à protéger les applications web et les API contre les attaques. Il vous permet de configurer un ensemble de règles, appelé liste de contrôle d'accès Web (ACL Web), qui autorise, bloque ou surveille (compte) les requêtes Web sur la base de règles et de conditions de sécurité Web personnalisables que vous définissez. Lorsque vous intégrez votreAWS AppSyncAPI avecAWS WAF, vous gagnez en contrôle et en visibilité sur le trafic HTTP accepté par votre API. Pour en savoir plus surAWS WAF, voirCommentAWS WAFŒuvresdans leAWS WAFGuide du développeur.

Vous pouvez l'utiliser AWS WAF pour protéger votre API AppSync contre les attaques web courantes, telles que l'injection SQL et les attaques XSS (cross-site scripting). Celles-ci peuvent

affecter la disponibilité et les performances des API, compromettre la sécurité ou consommer des ressources excessives. Par exemple, vous pouvez créer des règles pour autoriser ou bloquer les demandes provenant de plages d'adresses IP spécifiées, les demandes provenant de blocs CIDR, les demandes provenant d'un pays ou d'une région spécifique, les demandes contenant du code SQL malveillant ou les demandes contenant des scripts malveillants.

Vous pouvez également créer des règles qui correspondent à une chaîne spécifiée ou un modèle d'expression régulière dans les en-têtes HTTP, la méthode, la chaîne de requête, l'URI et le corps de la demande (limité aux 8 premiers Ko). De plus, vous pouvez créer des règles pour bloquer les attaques émanant d'agents utilisateurs spécifiques, de robots malveillants et d'extracteurs de contenu. Par exemple, vous pouvez utiliser des règles basées sur le débit pour spécifier le nombre de requêtes web que chaque adresse IP du client est autorisée à envoyer au cours d'une période de 5 minutes mise à jour en continu.

Pour en savoir plus sur les types de règles pris en charge et les informations supplémentairesAWS WAFfonctionnalités, consultez leAWS WAFGuide du développeuret leAWS WAFRéférence d'API.

> **Important**
>
> AWS WAF est la première ligne de défense contre les menaces web. QuandAWS WAFest activé sur une API,AWS WAFles règles sont évaluées avant les autres fonctionnalités de contrôle d'accès, telles que l'autorisation par clé d'API, les politiques IAM, les jetons OIDC et les groupes d'utilisateurs Amazon Cognito.

Intégrez unAppSyncAPI avecAWS WAF

Vous pouvez intégrer une API Appsync àAWS WAFà l'aide duAWS Management Console, leAWS CLI,AWS CloudFormation, ou tout autre client compatible.

Pour intégrer unAWS AppSyncAPI avecAWS WAF

1. Créez unAWS WAFACL Web. Pour des étapes détaillées à l'aide duAWS WAFConsole, voirCréation d'une ACL Web.
2. Définissez les règles de l'ACL Web. Une ou plusieurs règles sont définies lors de la création de l'ACL Web. Pour plus d'informations sur la structure des règles, voirAWS WAFrègles. Vous trouverez des exemples de règles utiles que vous pouvez définir pour votreAWS AppSyncAPI, voirCréation de règles pour une ACL Web.

3. Associez l'ACL Web à unAWS AppSyncAPI. Vous pouvez effectuer cette étape dansAWS WAFConsoleou dans leAppSyncConsole.
   - Pour associer l'ACL Web à unAWS AppSyncAPI dans leAWS WAFConsole, suivez les instructions pourAssocier ou dissocier une ACL Web avec unAWSressourcedans leAWS WAFGuide du développeur.
   - Pour associer l'ACL Web à unAWS AppSyncAPI dans leAWS AppSyncConsole
     a. Connectez-vous auAWS Management Consoleet ouvrez leAppSyncConsole.
     b. Choisissez l'API que vous souhaitez associer à une ACL Web.
     c. Dans le panneau de navigation, sélectionnez Settings (Paramètres).
     d. Dans lePare-feu pour applications Websection, activerActiverAWS WAF.
     e. Dans leACL Webliste déroulante, choisissez le nom de l'ACL Web à associer à votre API.
     f. ChoisissezEnregistrerpour associer l'ACL Web à votre API.

> **Note**
>
> Après avoir créé une ACL Web dansAWS WAFConsole, la mise à disposition de la nouvelle ACL Web peut prendre quelques minutes. Si vous ne voyez pas de nouvelle ACL Web dans lePare-feu pour applications Webmenu, attendez quelques minutes et réessayez les étapes pour associer l'ACL Web à votre API.

> **Note**
>
> AWS WAFl'intégration ne prend en charge que`Subscription registration message`événement pour les points de terminaison en temps réel.AWS AppSyncrépondra par un message d'erreur au lieu d'un`start_ack`message pour tous`Subscription registration message`bloqué parAWS WAF.

Après avoir associé une ACL Web à unAWS AppSyncAPI, vous allez gérer l'ACL Web à l'aide duAWS WAFAPI. Il n'est pas nécessaire de réassocier l'ACL Web à votreAWS AppSyncAPI, sauf si vous souhaitez associerAWS AppSyncAPI avec une ACL Web différente.

Création de règles pour une ACL Web

Les règles définissent comment inspecter les requêtes Web et ce qu'il faut faire lorsqu'une demande Web correspond aux critères d'inspection. Les règles n'existent pas dans AWS WAF par elles-mêmes. Vous pouvez accéder à une règle par son nom dans un groupe de règles ou dans l'ACL Web où elle est définie. Pour plus d'informations, voirAWS WAFrègles. Les exemples suivants montrent comment définir et associer des règles utiles pour protéger unAppSyncAPI.

Example règle ACL Web pour limiter la taille du corps de la demande

Voici un exemple de règle qui limite la taille du corps des demandes. Cela serait inscrit dans leÉditeur JSON de règleslors de la création d'une ACL Web dansAWS WAFConsole.

```
{
    "Name": "BodySizeRule",
    "Priority": 1,
    "Action": {
        "Block": {}
    },
    "Statement": {
        "SizeConstraintStatement": {
            "ComparisonOperator": "GE",
            "FieldToMatch": {
                "Body": {}
            },
            "Size": 1024,
            "TextTransformations": [
                {
                    "Priority": 0,
                    "Type": "NONE"
                }
             ]
          }
       },
       "VisibilityConfig": {
           "CloudWatchMetricsEnabled": true,
           "MetricName": "BodySizeRule",
           "SampledRequestsEnabled": true
        }
}
```

Après avoir créé votre ACL Web à l'aide de l'exemple de règle précédent, vous devez l'associer à votreAppSyncAPI. Comme alternative à l'utilisation duAWS Management Console, vous pouvez effectuer cette étape dansAWS CLIen exécutant la commande suivante.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

La propagation des modifications peut prendre quelques minutes, mais après avoir exécuté cette commande, les demandes dont le corps est supérieur à 1024 octets seront rejetées parAWS AppSync.

> **Note**
>
> Après avoir créé une nouvelle ACL Web dansAWS WAFConsole, quelques minutes peuvent être nécessaires pour que l'ACL Web soit disponible pour être associée à une API. Si vous exécutez la commande CLI et obtenez un`WAFUnavailableEntityException`erreur, attendez quelques minutes et réessayez d'exécuter la commande.

Example règle ACL Web pour limiter les demandes provenant d'une seule adresse IP

Voici un exemple de règle qui limite unAppSyncAPI pour 100 demandes à partir d'une seule adresse IP. Cela serait inscrit dans leÉditeur JSON de règleslors de la création d'une ACL Web avec une règle basée sur le taux dans leAWS WAFConsole.

```
{
  "Name": "Throttle",
  "Priority": 0,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "Throttle"
  },
  "Statement": {
    "RateBasedStatement": {
      "Limit": 100,
      "AggregateKeyType": "IP"
    }
```

```
  }
}
```

Après avoir créé votre ACL Web à l'aide de l'exemple de règle précédent, vous devez l'associer à votreAppSyncAPI. Vous pouvez effectuer cette étape dansAWS CLIen exécutant la commande suivante.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

Example règle ACL Web pour empêcher les requêtes d'introspection GraphQL __schema vers une API

Voici un exemple de règle qui empêche les requêtes d'introspection GraphQL __schema vers une API. Tout corps HTTP contenant la chaîne « __schema » sera bloqué. Cela serait inscrit dans leÉditeur JSON de règleslors de la création d'une ACL Web dansAWS WAFConsole.

```
{
  "Name": "BodyRule",
  "Priority": 5,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "BodyRule"
  },
  "Statement": {
    "ByteMatchStatement": {
      "FieldToMatch": {
        "Body": {}
      },
      "PositionalConstraint": "CONTAINS",
      "SearchString": "__schema",
      "TextTransformations": [
        {
          "Type": "NONE",
          "Priority": 0
        }
      ]
    }
  }
```

```
}
```

Après avoir créé votre ACL Web à l'aide de l'exemple de règle précédent, vous devez l'associer à votreAppSyncAPI. Vous pouvez effectuer cette étape dansAWS CLIen exécutant la commande suivante.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

Sécurité dans AWS AppSync

La sécurité du cloud AWS est la priorité absolue. En tant que AWS client, vous bénéficiez de centres de données et d'architectures réseau conçus pour répondre aux exigences des entreprises les plus sensibles en matière de sécurité.

La sécurité est une responsabilité partagée entre vous AWS et vous. Le modèle de responsabilité partagée décrit cela comme la sécurité du cloud et la sécurité dans le cloud :

- Sécurité du cloud : AWS est chargée de protéger l'infrastructure qui exécute les AWS services dans le AWS cloud. AWS vous fournit également des services que vous pouvez utiliser en toute sécurité. Des auditeurs tiers testent et vérifient régulièrement l'efficacité de notre sécurité dans le cadre des programmes de AWS conformité Programmes de de conformité. Pour en savoir plus sur les programmes de conformité qui s'appliquent à AWS AppSync, voir AWS Services concernés par programme de conformitéAWS .
- Sécurité dans le cloud — Votre responsabilité est déterminée par le AWS service que vous utilisez. Vous êtes également responsable d'autres facteurs, y compris de la sensibilité de vos données, des exigences de votre entreprise, ainsi que de la législation et de la réglementation applicables.

Cette documentation vous aide à comprendre comment appliquer le modèle de responsabilité partagée lors de son utilisation AWS AppSync. Les rubriques suivantes expliquent comment procéder à la configuration AWS AppSync pour atteindre vos objectifs de sécurité et de conformité. Vous apprendrez également à utiliser d'autres AWS services qui vous aident à surveiller et à sécuriser vos AWS AppSync ressources.

Rubriques

Protection des données dans AWS AppSync

Le modèle de responsabilité AWS partagée de s'applique à la protection des données dans AWS AppSync. Comme décrit dans ce modèle, AWS est chargé de protéger l'infrastructure mondiale qui gère tous les AWS Cloud. La gestion du contrôle de votre contenu hébergé sur cette infrastructure relève de votre responsabilité. Vous êtes également responsable des tâches de configuration et de gestion de la sécurité des Services AWS que vous utilisez. Pour plus d'informations sur la confidentialité des données, consultez Questions fréquentes (FAQ) sur la confidentialité des données. Pour en savoir plus sur la protection des données en Europe, consultez le billet de blog Modèle de responsabilité partagée AWS et RGPD (Règlement général sur la protection des données) sur le Blog de sécuritéAWS .

À des fins de protection des données, nous vous recommandons de protéger les Compte AWS informations d'identification et de configurer les utilisateurs individuels avec AWS IAM Identity Center ou AWS Identity and Access Management (IAM). Ainsi, chaque utilisateur se voit attribuer uniquement les autorisations nécessaires pour exécuter ses tâches. Nous vous recommandons également de sécuriser vos données comme indiqué ci-dessous :

- Utilisez l'authentification multifactorielle (MFA) avec chaque compte.
- Utilisez le protocole SSL/TLS pour communiquer avec les ressources. AWS Nous exigeons TLS 1.2 et recommandons TLS 1.3.
- Configurez l'API et la journalisation de l'activité des utilisateurs avec AWS CloudTrail.
- Utilisez des solutions de AWS chiffrement, ainsi que tous les contrôles de sécurité par défaut qu'ils contiennent Services AWS.
- Utilisez des services de sécurité gérés avancés tels qu'Amazon Macie, qui contribuent à la découverte et à la sécurisation des données sensibles stockées dans Amazon S3.
- Si vous avez besoin de modules cryptographiques validés par la norme FIPS 140-2 pour accéder AWS via une interface de ligne de commande ou une API, utilisez un point de terminaison FIPS. Pour plus d'informations sur les points de terminaison FIPS (Federal Information Processing Standard) disponibles, consultez Federal Information Processing Standard (FIPS) 140-2 (Normes de traitement de l'information fédérale).

Nous vous recommandons fortement de ne jamais placer d'informations confidentielles ou sensibles, telles que les adresses e-mail de vos clients, dans des balises ou des champs de texte libre tels que le champ Name (Nom). Cela inclut lorsque vous travaillez avec AWS AppSync ou d'autres Services AWS utilisateurs de la console, de l'API ou AWS des SDK. AWS CLI Toutes les données que vous

entrez dans des balises ou des champs de texte de forme libre utilisés pour les noms peuvent être utilisées à des fins de facturation ou dans les journaux de diagnostic. Si vous fournissez une adresse URL à un serveur externe, nous vous recommandons fortement de ne pas inclure d'informations d'identification dans l'adresse URL permettant de valider votre demande adressée à ce serveur.

Chiffrement en mouvement

AWS AppSync, comme tous les AWS services, utilise TLS1.2 et les versions ultérieures pour communiquer lors de l'utilisation des API et des AWS SDK publiés.

L'utilisation AWS AppSync avec d'autres AWS services tels qu'Amazon DynamoDB garantit le chiffrement en transit : AWS tous les services utilisent le protocole TLS 1.2 et les versions ultérieures pour communiquer entre eux, sauf indication contraire. Pour les résolveurs utilisant Amazon EC2 CloudFront ou Amazon EC2, il est de votre responsabilité de vérifier que le protocole TLS (HTTPS) est configuré et sécurisé. Pour plus d'informations sur la configuration du protocole HTTPS dans Amazon EC2, consultez la section Configuration du protocole SSL/TLS sur Amazon Linux 2 dans le guide de l'utilisateur d'Amazon EC2. Pour plus d'informations sur la configuration du protocole HTTPS CloudFront activé, consultez la section HTTPS sur Amazon CloudFront dans le guide de CloudFront l'utilisateur.

Validation de conformité pour AWS AppSync

Des auditeurs tiers évaluent la sécurité et AWS AppSync la conformité de plusieurs programmes de AWS conformité. AWS AppSync est conforme aux programmes SOC, PCI, HIPAA/HIPAA BAA, IRAP, C5, ENS High, OSPAR et HITRUST CSF.

Pour savoir si un programme Services AWS de conformité Service AWS s'inscrit dans le champ d'application de programmes de conformité spécifiques, consultez Services AWS la section de conformité et sélectionnez le programme de conformité qui vous intéresse. Pour des informations générales, voir Programmes de AWS conformité Programmes AWS de .

Vous pouvez télécharger des rapports d'audit tiers à l'aide de AWS Artifact. Pour plus d'informations, voir Téléchargement de rapports dans AWS Artifact .

Votre responsabilité en matière de conformité lors de l'utilisation Services AWS est déterminée par la sensibilité de vos données, les objectifs de conformité de votre entreprise et les lois et réglementations applicables. AWS fournit les ressources suivantes pour faciliter la mise en conformité :

- Guides de démarrage rapide sur la sécurité et la conformité : ces guides de déploiement abordent les considérations architecturales et indiquent les étapes à suivre pour déployer des environnements de base axés sur AWS la sécurité et la conformité.
- Architecture axée sur la sécurité et la conformité HIPAA sur Amazon Web Services : ce livre blanc décrit comment les entreprises peuvent créer des applications AWS conformes à la loi HIPAA.

> **Note**
>
> Tous ne Services AWS sont pas éligibles à la loi HIPAA. Pour plus d'informations, consultez le HIPAA Eligible Services Reference.

- AWS Ressources de https://aws.amazon.com/compliance/resources/ de conformité — Cette collection de classeurs et de guides peut s'appliquer à votre secteur d'activité et à votre région.
- AWS Guides de conformité destinés aux clients — Comprenez le modèle de responsabilité partagée sous l'angle de la conformité. Les guides résument les meilleures pratiques en matière de sécurisation Services AWS et décrivent les directives relatives aux contrôles de sécurité dans de nombreux cadres (notamment le National Institute of Standards and Technology (NIST), le Payment Card Industry Security Standards Council (PCI) et l'Organisation internationale de normalisation (ISO)).
- Évaluation des ressources à l'aide des règles du guide du AWS Config développeur : le AWS Config service évalue dans quelle mesure les configurations de vos ressources sont conformes aux pratiques internes, aux directives du secteur et aux réglementations.
- AWS Security Hub— Cela Service AWS fournit une vue complète de votre état de sécurité interne AWS. Security Hub utilise des contrôles de sécurité pour évaluer vos ressources AWS et vérifier votre conformité par rapport aux normes et aux bonnes pratiques du secteur de la sécurité. Pour obtenir la liste des services et des contrôles pris en charge, consultez Référence des contrôles Security Hub.
- Amazon GuardDuty — Cela Service AWS détecte les menaces potentielles qui pèsent sur vos charges de travail Comptes AWS, vos conteneurs et vos données en surveillant votre environnement pour détecter toute activité suspecte et malveillante. GuardDuty peut vous aider à répondre à diverses exigences de conformité, telles que la norme PCI DSS, en répondant aux exigences de détection des intrusions imposées par certains cadres de conformité.
- AWS Audit Manager— Cela vous Service AWS permet d'auditer en permanence votre AWS utilisation afin de simplifier la gestion des risques et la conformité aux réglementations et aux normes du secteur.

Sécurité de l'infrastructure dans AWS AppSync

En tant que service géré, AWS AppSync il est protégé par la sécurité du réseau AWS mondial. Pour plus d'informations sur les services AWS de sécurité et sur la manière dont AWS l'infrastructure est protégée, consultez la section Sécurité du AWS cloud. Pour concevoir votre AWS environnement en utilisant les meilleures pratiques en matière de sécurité de l'infrastructure, consultez la section Protection de l'infrastructure dans le cadre AWS bien architecturé du pilier de sécurité.

Vous utilisez des appels d'API AWS publiés pour accéder AWS AppSync via le réseau. Les clients doivent prendre en charge les éléments suivants :

- Protocole TLS (Transport Layer Security). Nous exigeons TLS 1.2 et recommandons TLS 1.3.
- Ses suites de chiffrement PFS (Perfect Forward Secrecy) comme DHE (Ephemeral Diffie-Hellman) ou ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). La plupart des systèmes modernes tels que Java 7 et les versions ultérieures prennent en charge ces modes.

En outre, les demandes doivent être signées à l'aide d'un ID de clé d'accès et d'une clé d'accès secrète associée à un principal IAM. Vous pouvez également utiliser AWS Security Token Service (AWS STS) pour générer des informations d'identification de sécurité temporaires et signer les demandes.

Résilience dans AWS AppSync

L'infrastructure AWS mondiale est construite autour des AWS régions et des zones de disponibilité. AWS Les régions fournissent plusieurs zones de disponibilité physiquement séparées et isolées, connectées par un réseau à faible latence, à haut débit et hautement redondant. Avec les zones de disponibilité, vous pouvez concevoir et exploiter des applications et des bases de données qui basculent automatiquement d'une zone à l'autre sans interruption. Les zones de disponibilité sont davantage disponibles, tolérantes aux pannes et ont une plus grande capacité de mise à l'échelle que les infrastructures traditionnelles à un ou plusieurs centres de données.

Pour plus d'informations sur AWS les régions et les zones de disponibilité, consultez la section Infrastructure AWS mondiale.

Outre l'infrastructure AWS globale, elle AWS AppSync permet de définir la plupart des ressources à l'aide de AWS CloudFormation modèles ; pour un exemple d'utilisation de AWS CloudFormation modèles pour déclarer des AWS AppSync ressources, voir Cas d'utilisation pratiques pour les

résolveurs de AWS AppSync pipeline sur le AWS blog et le guide de l'AWS CloudFormation utilisateur.

Gestion des identités et des accès pour AWS AppSync

AWS Identity and Access Management (IAM) est un outil Service AWS qui permet à un administrateur de contrôler en toute sécurité l'accès aux AWS ressources. Les administrateurs IAM contrôlent qui peut être authentifié (connecté) et autorisé (autorisé) à utiliser AWS AppSync les ressources. IAM est un Service AWS outil que vous pouvez utiliser sans frais supplémentaires.

Rubriques

Public ciblé

La façon dont vous utilisez AWS Identity and Access Management (IAM) varie en fonction du travail que vous effectuez. AWS AppSync

Utilisateur du service : si vous utilisez le AWS AppSync service pour effectuer votre travail, votre administrateur vous fournit les informations d'identification et les autorisations dont vous avez besoin. Au fur et à mesure que vous utilisez de nouvelles AWS AppSync fonctionnalités pour effectuer votre travail, vous aurez peut-être besoin d'autorisations supplémentaires. En comprenant bien la gestion des accès, vous saurez demander les autorisations appropriées à votre administrateur. Si vous ne pouvez pas accéder à une fonctionnalité dans AWS AppSync, consultezRésolution des problèmes AWS AppSync d'identité et d'accès.

Administrateur du service — Si vous êtes responsable des AWS AppSync ressources de votre entreprise, vous avez probablement un accès complet à AWS AppSync. C'est à vous de déterminer les AWS AppSync fonctionnalités et les ressources auxquelles les utilisateurs de votre service

doivent accéder. Vous devez ensuite soumettre les demandes à votre administrateur IAM pour modifier les autorisations des utilisateurs de votre service. Consultez les informations sur cette page pour comprendre les concepts de base d'IAM. Pour en savoir plus sur la manière dont votre entreprise peut utiliser IAM avec AWS AppSync, voirComment AWS AppSync fonctionne avec IAM.

Administrateur IAM — Si vous êtes administrateur IAM, vous souhaiterez peut-être en savoir plus sur la manière dont vous pouvez rédiger des politiques pour gérer l'accès à. AWS AppSync Pour consulter des exemples de politiques AWS AppSync basées sur l'identité que vous pouvez utiliser dans IAM, consultez. Stratégies basées sur l'identité pour AWS AppSync

Authentification par des identités

L'authentification est la façon dont vous vous connectez à AWS l'aide de vos informations d'identification. Vous devez être authentifié (connecté à AWS) en tant qu'utilisateur IAM ou en assumant un rôle IAM. Utilisateur racine d'un compte AWS

Vous pouvez vous connecter en AWS tant qu'identité fédérée en utilisant les informations d'identification fournies par le biais d'une source d'identité. AWS IAM Identity Center Les utilisateurs (IAM Identity Center), l'authentification unique de votre entreprise et vos informations d'identification Google ou Facebook sont des exemples d'identités fédérées. Lorsque vous vous connectez avec une identité fédérée, votre administrateur aura précédemment configuré une fédération d'identités avec des rôles IAM. Lorsque vous accédez à AWS l'aide de la fédération, vous assumez indirectement un rôle.

Selon le type d'utilisateur que vous êtes, vous pouvez vous connecter au portail AWS Management Console ou au portail AWS d'accès. Pour plus d'informations sur la connexion à AWS, consultez la section Comment vous connecter à votre compte Compte AWS dans le guide de Connexion à AWS l'utilisateur.

Si vous y accédez AWS par programmation, AWS fournit un kit de développement logiciel (SDK) et une interface de ligne de commande (CLI) pour signer cryptographiquement vos demandes à l'aide de vos informations d'identification. Si vous n'utilisez pas d' AWS outils, vous devez signer vous-même les demandes. Pour plus d'informations sur l'utilisation de la méthode recommandée pour signer vous-même les demandes, consultez la section Signature des demandes AWS d'API dans le guide de l'utilisateur IAM.

Quelle que soit la méthode d'authentification que vous utilisez, vous devrez peut-être fournir des informations de sécurité supplémentaires. Par exemple, il vous AWS recommande d'utiliser l'authentification multifactorielle (MFA) pour renforcer la sécurité de votre compte. Pour en savoir

plus, consultez Authentification multifactorielle dans le Guide de l'utilisateur AWS IAM Identity Center et Utilisation de l'authentification multifactorielle (MFA) dans l'interface AWS dans le Guide de l'utilisateur IAM.

Compte AWS utilisateur root

Lorsque vous créez un Compte AWS, vous commencez par une identité de connexion unique qui donne un accès complet à toutes Services AWS les ressources du compte. Cette identité est appelée utilisateur Compte AWS root et est accessible en vous connectant avec l'adresse e-mail et le mot de passe que vous avez utilisés pour créer le compte. Il est vivement recommandé de ne pas utiliser l'utilisateur racine pour vos tâches quotidiennes. Protégez vos informations d'identification d'utilisateur racine et utilisez-les pour effectuer les tâches que seul l'utilisateur racine peut effectuer. Pour obtenir la liste complète des tâches qui vous imposent de vous connecter en tant qu'utilisateur root, consultez Tâches nécessitant des informations d'identification d'utilisateur root dans le Guide de l'utilisateur IAM.

Identité fédérée

La meilleure pratique consiste à obliger les utilisateurs humains, y compris ceux qui ont besoin d'un accès administrateur, à utiliser la fédération avec un fournisseur d'identité pour accéder à l'aide Services AWS d'informations d'identification temporaires.

Une identité fédérée est un utilisateur de l'annuaire des utilisateurs de votre entreprise, d'un fournisseur d'identité Web AWS Directory Service, du répertoire Identity Center ou de tout utilisateur qui y accède à l'aide des informations d'identification fournies Services AWS par le biais d'une source d'identité. Lorsque des identités fédérées y accèdent Comptes AWS, elles assument des rôles, qui fournissent des informations d'identification temporaires.

Pour une gestion des accès centralisée, nous vous recommandons d'utiliser AWS IAM Identity Center. Vous pouvez créer des utilisateurs et des groupes dans IAM Identity Center, ou vous pouvez vous connecter et synchroniser avec un ensemble d'utilisateurs et de groupes dans votre propre source d'identité afin de les utiliser dans toutes vos applications Comptes AWS et applications. Pour obtenir des informations sur IAM Identity Center, consultez Qu'est-ce que IAM Identity Center ? dans le Guide de l'utilisateur AWS IAM Identity Center .

Utilisateurs et groupes IAM

Un utilisateur IAM est une identité au sein de vous Compte AWS qui possède des autorisations spécifiques pour une seule personne ou une seule application. Dans la mesure du possible, nous

vous recommandons de vous appuyer sur des informations d'identification temporaires plutôt que de créer des utilisateurs IAM ayant des informations d'identification à long terme tels que les clés d'accès. Toutefois, si certains cas d'utilisation spécifiques nécessitent des informations d'identification à long terme avec les utilisateurs IAM, nous vous recommandons de faire pivoter les clés d'accès. Pour plus d'informations, consultez Rotation régulière des clés d'accès pour les cas d'utilisation nécessitant des informations d'identification dans le Guide de l'utilisateur IAM.

Un groupe IAM est une identité qui concerne un ensemble d'utilisateurs IAM. Vous ne pouvez pas vous connecter en tant que groupe. Vous pouvez utiliser les groupes pour spécifier des autorisations pour plusieurs utilisateurs à la fois. Les groupes permettent de gérer plus facilement les autorisations pour de grands ensembles d'utilisateurs. Par exemple, vous pouvez avoir un groupe nommé IAMAdmins et accorder à ce groupe les autorisations d'administrer des ressources IAM.

Les utilisateurs sont différents des rôles. Un utilisateur est associé de manière unique à une personne ou une application, alors qu'un rôle est conçu pour être endossé par tout utilisateur qui en a besoin. Les utilisateurs disposent d'informations d'identification permanentes, mais les rôles fournissent des informations d'identification temporaires. Pour en savoir plus, consultez Quand créer un utilisateur IAM (au lieu d'un rôle) dans le Guide de l'utilisateur IAM.

Rôles IAM

Un rôle IAM est une identité au sein de vous Compte AWS dotée d'autorisations spécifiques. Le concept ressemble à celui d'utilisateur IAM, mais le rôle IAM n'est pas associé à une personne en particulier. Vous pouvez assumer temporairement un rôle IAM dans le en AWS Management Console changeant de rôle. Vous pouvez assumer un rôle en appelant une opération d' AWS API AWS CLI ou en utilisant une URL personnalisée. Pour plus d'informations sur les méthodes d'utilisation des rôles, consultez Utilisation de rôles IAM dans le Guide de l'utilisateur IAM.

Les rôles IAM avec des informations d'identification temporaires sont utiles dans les cas suivants :

- Accès utilisateur fédéré – Pour attribuer des autorisations à une identité fédérée, vous créez un rôle et définissez des autorisations pour le rôle. Quand une identité externe s'authentifie, l'identité est associée au rôle et reçoit les autorisations qui sont définies par celui-ci. Pour obtenir des informations sur les rôles pour la fédération, consultez Création d'un rôle pour un fournisseur d'identité tiers (fédération) dans le Guide de l'utilisateur IAM. Si vous utilisez IAM Identity Center, vous configurez un jeu d'autorisations. IAM Identity Center met en corrélation le jeu d'autorisations avec un rôle dans IAM afin de contrôler à quoi vos identités peuvent accéder après leur authentification. Pour plus d'informations sur les jeux d'autorisations, consultez la rubrique Jeux d'autorisations dans le Guide de l'utilisateur AWS IAM Identity Center .

- Autorisations d'utilisateur IAM temporaires : un rôle ou un utilisateur IAM peut endosser un rôle IAM pour profiter temporairement d'autorisations différentes pour une tâche spécifique.
- Accès intercompte : vous pouvez utiliser un rôle IAM pour permettre à un utilisateur (principal de confiance) d'un compte différent d'accéder aux ressources de votre compte. Les rôles constituent le principal moyen d'accorder l'accès intercompte. Toutefois, dans certains Services AWS cas, vous pouvez associer une politique directement à une ressource (au lieu d'utiliser un rôle comme proxy). Pour en savoir plus sur la différence entre les rôles et les politiques basées sur les ressources pour l'accès intercompte, consultez Différence entre les rôles IAM et les politiques basées sur les ressources dans le Guide de l'utilisateur IAM.
- Accès multiservices — Certains Services AWS utilisent des fonctionnalités dans d'autres Services AWS. Par exemple, lorsque vous effectuez un appel dans un service, il est courant que ce service exécute des applications dans Amazon EC2 ou stocke des objets dans Amazon S3. Un service peut le faire en utilisant les autorisations d'appel du principal, un rôle de service ou un rôle lié au service.
  - Sessions d'accès direct (FAS) : lorsque vous utilisez un utilisateur ou un rôle IAM pour effectuer des actions AWS, vous êtes considéré comme un mandant. Lorsque vous utilisez certains services, vous pouvez effectuer une action qui initie une autre action dans un autre service. FAS utilise les autorisations du principal appelant et Service AWS, associées Service AWS à la demande, pour adresser des demandes aux services en aval. Les demandes FAS ne sont effectuées que lorsqu'un service reçoit une demande qui nécessite des interactions avec d'autres personnes Services AWS ou des ressources pour être traitée. Dans ce cas, vous devez disposer d'autorisations nécessaires pour effectuer les deux actions. Pour plus de détails sur la politique relative à la transmission de demandes FAS, consultez Sessions de transmission d'accès.
  - Rôle de service : il s'agit d'un rôle IAM attribué à un service afin de réaliser des actions en votre nom. Un administrateur IAM peut créer, modifier et supprimer une fonction du service à partir d'IAM. Pour plus d'informations, consultez Création d'un rôle pour la délégation d'autorisations à un Service AWS dans le Guide de l'utilisateur IAM.
  - Rôle lié à un service — Un rôle lié à un service est un type de rôle de service lié à un. Service AWS Le service peut endosser le rôle afin d'effectuer une action en votre nom. Les rôles liés au service apparaissent dans votre Compte AWS fichier et appartiennent au service. Un administrateur IAM peut consulter, mais ne peut pas modifier, les autorisations concernant les rôles liés à un service.
- Applications exécutées sur Amazon EC2 : vous pouvez utiliser un rôle IAM pour gérer les informations d'identification temporaires pour les applications qui s'exécutent sur une instance EC2 et qui envoient des demandes d'API. AWS CLI AWS Cette solution est préférable au stockage

des clés d’accès au sein de l’instance EC2. Pour attribuer un AWS rôle à une instance EC2 et le mettre à la disposition de toutes ses applications, vous devez créer un profil d'instance attaché à l'instance. Un profil d’instance contient le rôle et permet aux programmes qui s’exécutent sur l’instance EC2 d’obtenir des informations d’identification temporaires. Pour plus d’informations, consultez Utilisation d’un rôle IAM pour accorder des autorisations à des applications s’exécutant sur des instances Amazon EC2 dans le Guide de l’utilisateur IAM.

Pour savoir dans quel cas utiliser des rôles ou des utilisateurs IAM, consultez Quand créer un rôle IAM (au lieu d’un utilisateur) dans le Guide de l’utilisateur IAM.

Gestion des accès à l’aide de politiques

Vous contrôlez l'accès en AWS créant des politiques et en les associant à AWS des identités ou à des ressources. Une politique est un objet AWS qui, lorsqu'il est associé à une identité ou à une ressource, définit leurs autorisations. AWS évalue ces politiques lorsqu'un principal (utilisateur, utilisateur root ou session de rôle) fait une demande. Les autorisations dans les politiques déterminent si la demande est autorisée ou refusée. La plupart des politiques sont stockées AWS sous forme de documents JSON. Pour plus d’informations sur la structure et le contenu des documents de politique JSON, consultez Vue d’ensemble des politiques JSON dans le Guide de l’utilisateur IAM.

Les administrateurs peuvent utiliser les politiques AWS JSON pour spécifier qui a accès à quoi. C’est-à-dire, quel principal peut effectuer des actions sur quelles ressources et dans quelles conditions.

Par défaut, les utilisateurs et les rôles ne disposent d’aucune autorisation. Pour octroyer aux utilisateurs des autorisations d’effectuer des actions sur les ressources dont ils ont besoin, un administrateur IAM peut créer des politiques IAM. L’administrateur peut ensuite ajouter les politiques IAM aux rôles et les utilisateurs peuvent assumer les rôles.

Les politiques IAM définissent les autorisations d’une action, quelle que soit la méthode que vous utilisez pour exécuter l’opération. Par exemple, supposons que vous disposiez d’une politique qui autorise l’action `iam:GetRole`. Un utilisateur appliquant cette politique peut obtenir des informations sur le rôle à partir de AWS Management Console AWS CLI, de ou de l' AWS API.

Politiques basées sur l’identité

Les politiques basées sur l’identité sont des documents de politique d’autorisations JSON que vous pouvez attacher à une identité telle qu’un utilisateur, un groupe d’utilisateurs ou un rôle IAM. Ces

politiques contrôlent quel type d'actions des utilisateurs et des rôles peuvent exécuter, sur quelles ressources et dans quelles conditions. Pour découvrir comment créer une politique basée sur l'identité, consultez Création de politiques IAM dans le Guide de l'utilisateur IAM.

Les politiques basées sur l'identité peuvent être classées comme des politiques en ligne ou des politiques gérées. Les politiques en ligne sont intégrées directement à un utilisateur, groupe ou rôle. Les politiques gérées sont des politiques autonomes que vous pouvez associer à plusieurs utilisateurs, groupes et rôles au sein de votre Compte AWS. Les politiques gérées incluent les politiques AWS gérées et les politiques gérées par le client. Pour découvrir comment choisir entre une politique gérée et une politique en ligne, consultez Choix entre les politiques gérées et les politiques en ligne dans le Guide de l'utilisateur IAM.

politiques basées sur les ressources

Les politiques basées sur les ressources sont des documents de politique JSON que vous attachez à une ressource. Des politiques basées sur les ressources sont, par exemple, les politiques de confiance de rôle IAM et des politiques de compartiment. Dans les services qui sont compatibles avec les politiques basées sur les ressources, les administrateurs de service peuvent les utiliser pour contrôler l'accès à une ressource spécifique. Pour la ressource dans laquelle se trouve la politique, cette dernière définit quel type d'actions un principal spécifié peut effectuer sur cette ressource et dans quelles conditions. Vous devez spécifier un principal dans une politique basée sur les ressources. Les principaux peuvent inclure des comptes, des utilisateurs, des rôles, des utilisateurs fédérés ou. Services AWS

Les politiques basées sur les ressources sont des politiques en ligne situées dans ce service. Vous ne pouvez pas utiliser les politiques AWS gérées par IAM dans une stratégie basée sur les ressources.

Listes de contrôle d'accès (ACL)

Les listes de contrôle d'accès (ACL) vérifie quels principals (membres de compte, utilisateurs ou rôles) ont l'autorisation d'accéder à une ressource. Les listes de contrôle d'accès sont similaires aux politiques basées sur les ressources, bien qu'elles n'utilisent pas le format de document de politique JSON.

Amazon S3 et Amazon VPC sont des exemples de services qui prennent en charge les ACL. AWS WAF Pour en savoir plus sur les listes de contrôle d'accès, consultez Vue d'ensemble des listes de contrôle d'accès (ACL) dans le Guide du développeur Amazon Simple Storage Service.

Autres types de politique

AWS prend en charge d'autres types de politiques moins courants. Ces types de politiques peuvent définir le nombre maximum d'autorisations qui vous sont accordées par des types de politiques plus courants.

- Limite d'autorisations : une limite d'autorisations est une fonctionnalité avancée dans laquelle vous définissez le nombre maximal d'autorisations qu'une politique basée sur l'identité peut accorder à une entité IAM (utilisateur ou rôle IAM). Vous pouvez définir une limite d'autorisations pour une entité. Les autorisations en résultant représentent la combinaison des politiques basées sur l'identité d'une entité et de ses limites d'autorisation. Les politiques basées sur les ressources qui spécifient l'utilisateur ou le rôle dans le champ `Principal` ne sont pas limitées par les limites d'autorisations. Un refus explicite dans l'une de ces politiques remplace l'autorisation. Pour plus d'informations sur les limites d'autorisations, consultez Limites d'autorisations pour des entités IAM dans le Guide de l'utilisateur IAM.
- Politiques de contrôle des services (SCP) — Les SCP sont des politiques JSON qui spécifient les autorisations maximales pour une organisation ou une unité organisationnelle (UO) dans. AWS Organizations AWS Organizations est un service permettant de regrouper et de gérer de manière centralisée Comptes AWS les multiples propriétés de votre entreprise. Si vous activez toutes les fonctionnalités d'une organisation, vous pouvez appliquer les politiques de contrôle des services (SCP) à l'un ou à l'ensemble de vos comptes. Le SCP limite les autorisations pour les entités figurant dans les comptes des membres, y compris chacune Utilisateur racine d'un compte AWS d'entre elles. Pour plus d'informations sur les organisations et les SCP, consultez Fonctionnement des SCP dans le Guide de l'utilisateur AWS Organizations .
- Politiques de séance : les politiques de séance sont des politiques avancées que vous utilisez en tant que paramètre lorsque vous créez par programmation une séance temporaire pour un rôle ou un utilisateur fédéré. Les autorisations de séance en résultant sont une combinaison des politiques basées sur l'identité de l'utilisateur ou du rôle et des politiques de séance. Les autorisations peuvent également provenir d'une politique basée sur les ressources. Un refus explicite dans l'une de ces politiques annule l'autorisation. Pour plus d'informations, consultez politiques de séance dans le Guide de l'utilisateur IAM.

Plusieurs types de politique

Lorsque plusieurs types de politiques s'appliquent à la requête, les autorisations en résultant sont plus compliquées à comprendre. Pour savoir comment AWS déterminer s'il faut autoriser

une demande lorsque plusieurs types de politiques sont impliqués, consultez la section Logique d'évaluation des politiques dans le guide de l'utilisateur IAM.

Comment AWS AppSync fonctionne avec IAM

Avant d'utiliser IAM pour gérer l'accès à AWS AppSync, découvrez les fonctionnalités IAM disponibles. AWS AppSync

Fonctionnalités IAM que vous pouvez utiliser avec AWS AppSync

Fonction IAM	AWS AppSync soutien
Politiques basées sur l’identité	Oui
Politiques basées sur les ressources	Non
Actions de politique	Oui
Ressources de politique	Oui
Clés de condition d’une politique	Non
ACL	Non
ABAC (identifications dans les politiques)	Partielle
Informations d’identification temporaires	Oui
Transfert des sessions d’accès (FAS)	Partielle
Fonctions du service	Non
Rôles liés à un service	Partielle

Pour obtenir une vue d'ensemble de la façon dont AWS AppSync les autres AWS services fonctionnent avec la plupart des fonctionnalités IAM, consultez la section AWS Services compatibles avec IAM dans le Guide de l'utilisateur IAM.

Politiques basées sur l'identité pour AWS AppSync

Prend en charge les politiques basées sur l'identité	Oui

Les politiques basées sur l'identité sont des documents de politique d'autorisations JSON que vous pouvez attacher à une identité telle qu'un utilisateur, un groupe d'utilisateurs ou un rôle IAM. Ces politiques contrôlent quel type d'actions des utilisateurs et des rôles peuvent exécuter, sur quelles ressources et dans quelles conditions. Pour découvrir comment créer une politique basée sur l'identité, consultez Création de politiques IAM dans le Guide de l'utilisateur IAM.

Avec les politiques IAM basées sur l'identité, vous pouvez spécifier des actions et ressources autorisées ou refusées, ainsi que les conditions dans lesquelles les actions sont autorisées ou refusées. Vous ne pouvez pas spécifier le principal dans une politique basée sur une identité car celle-ci s'applique à l'utilisateur ou au rôle auquel elle est attachée. Pour découvrir tous les éléments que vous utilisez dans une politique JSON, consultez Références des éléments de politique JSON IAM dans le Guide de l'utilisateur IAM.

Exemples de politiques basées sur l'identité pour AWS AppSync

Pour consulter des exemples de politiques AWS AppSync basées sur l'identité, consultez. Stratégies basées sur l'identité pour AWS AppSync

Politiques basées sur les ressources au sein de AWS AppSync

Prend en charge les politiques basées sur les ressources	Non

Les politiques basées sur les ressources sont des documents de politique JSON que vous attachez à une ressource. Des politiques basées sur les ressources sont, par exemple, les politiques de confiance de rôle IAM et des politiques de compartiment. Dans les services qui sont compatibles avec les politiques basées sur les ressources, les administrateurs de service peuvent les utiliser pour contrôler l'accès à une ressource spécifique. Pour la ressource dans laquelle se trouve la politique, cette dernière définit quel type d'actions un principal spécifié peut effectuer sur cette ressource et dans quelles conditions. Vous devez spécifier un principal dans une politique basée sur les

ressources. Les principaux peuvent inclure des comptes, des utilisateurs, des rôles, des utilisateurs fédérés ou. Services AWS

Pour permettre un accès intercompte, vous pouvez spécifier un compte entier ou des entités IAM dans un autre compte en tant que principal dans une politique basée sur les ressources. L'ajout d'un principal entre comptes à une politique basée sur les ressources ne représente qu'une partie de l'instauration de la relation d'approbation. Lorsque le principal et la ressource sont différents Comptes AWS, un administrateur IAM du compte sécurisé doit également accorder à l'entité principale (utilisateur ou rôle) l'autorisation d'accéder à la ressource. Pour ce faire, il attache une politique basée sur une identité à l'entité. Toutefois, si une politique basée sur des ressources accorde l'accès à un principal dans le même compte, aucune autre politique basée sur l'identité n'est requise. Pour plus d'informations, consultez Différence entre les rôles IAM et les politiques basées sur une ressource dans le Guide de l'utilisateur IAM.

Actions politiques pour AWS AppSync

Prend en charge les actions de politique	Oui

Les administrateurs peuvent utiliser les politiques AWS JSON pour spécifier qui a accès à quoi. C'est-à-dire, quel principal peut effectuer des actions sur quelles ressources et dans quelles conditions.

L'élément `Action` d'une politique JSON décrit les actions que vous pouvez utiliser pour autoriser ou refuser l'accès à une politique. Les actions de stratégie portent généralement le même nom que l'opération AWS d'API associée. Il existe quelques exceptions, telles que les actions avec autorisations uniquement qui n'ont pas d'opération API correspondante. Certaines opérations nécessitent également plusieurs actions dans une politique. Ces actions supplémentaires sont nommées actions dépendantes.

Intégration d'actions dans une stratégie afin d'accorder l'autorisation d'exécuter les opérations associées.

Pour consulter la liste des AWS AppSync actions, reportez-vous à la section Actions définies par AWS AppSync dans la référence d'autorisation de service.

Les actions de politique en AWS AppSync cours utilisent le préfixe suivant avant l'action :

```
appsync
```

Pour indiquer plusieurs actions dans une seule déclaration, séparez-les par des virgules.

```
"Action": [
      "appsync:action1",
      "appsync:action2"
         ]
```

Pour consulter des exemples de politiques AWS AppSync basées sur l'identité, consultez. Stratégies basées sur l’identité pour AWS AppSync

Ressources politiques pour AWS AppSync

Prend en charge les ressources de politique	Oui

Les administrateurs peuvent utiliser les politiques AWS JSON pour spécifier qui a accès à quoi. C’est-à-dire, quel principal peut effectuer des actions sur quelles ressources et dans quelles conditions.

L’élément de politique JSON `Resource` indique le ou les objets auxquels l’action s’applique. Les instructions doivent inclure un élément `Resource` ou `NotResource`. Il est recommandé de définir une ressource à l’aide de son Amazon Resource Name (ARN). Vous pouvez le faire pour des actions qui prennent en charge un type de ressource spécifique, connu sous la dénomination autorisations de niveau ressource.

Pour les actions qui ne sont pas compatibles avec les autorisations de niveau ressource, telles que les opérations de liste, utilisez un caractère générique (*) afin d’indiquer que l’instruction s’applique à toutes les ressources.

```
"Resource": "*"
```

Pour consulter la liste des types de AWS AppSync ressources et de leurs ARN, voir Ressources définies par AWS AppSync dans la référence d'autorisation de service. Pour savoir avec quelles actions vous pouvez spécifier l'ARN de chaque ressource, consultez la section Actions définies par AWS AppSync.

Pour consulter des exemples de politiques AWS AppSync basées sur l'identité, consultez. Stratégies basées sur l’identité pour AWS AppSync

Clés de conditions de politique pour AWS AppSync

Prend en charge les clés de condition de politique spécifiques au service	Non

Les administrateurs peuvent utiliser les politiques AWS JSON pour spécifier qui a accès à quoi. C’est-à-dire, quel principal peut effectuer des actions sur quelles ressources et dans quelles conditions.

L’élément `Condition` (ou le bloc `Condition`) vous permet de spécifier des conditions lorsqu’une instruction est appliquée. L’élément `Condition` est facultatif. Vous pouvez créer des expressions conditionnelles qui utilisent des opérateurs de condition, tels que les signes égal ou inférieur à, pour faire correspondre la condition de la politique aux valeurs de la demande.

Si vous spécifiez plusieurs éléments `Condition` dans une instruction, ou plusieurs clés dans un seul élément `Condition`, AWS les évalue à l’aide d’une opération `AND` logique. Si vous spécifiez plusieurs valeurs pour une seule clé de condition, AWS évalue la condition à l'aide d'une `OR` opération logique. Toutes les conditions doivent être remplies avant que les autorisations associées à l’instruction ne soient accordées.

Vous pouvez aussi utiliser des variables d’espace réservé quand vous spécifiez des conditions. Par exemple, vous pouvez accorder à un utilisateur IAM l’autorisation d’accéder à une ressource uniquement si elle est balisée avec son nom d’utilisateur IAM. Pour plus d’informations, consultez Éléments d’une politique IAM : variables et identifications dans le Guide de l’utilisateur IAM.

AWS prend en charge les clés de condition globales et les clés de condition spécifiques au service. Pour voir toutes les clés de condition AWS globales, voir les clés de contexte de condition AWS globales dans le guide de l'utilisateur IAM.

Pour consulter la liste des clés de AWS AppSync condition, reportez-vous à la section Clés de condition pour AWS AppSync la référence d'autorisation de service. Pour savoir avec quelles actions et ressources vous pouvez utiliser une clé de condition, consultez la section Actions définies par AWS AppSync.

Pour consulter des exemples de politiques AWS AppSync basées sur l'identité, consultez. Stratégies basées sur l’identité pour AWS AppSync

Listes de contrôle d’accès (ACL) dans AWS AppSync

Prend en charge les listes ACL	Non

Les listes de contrôle d’accès (ACL) vérifient quels principaux (membres de compte, utilisateurs ou rôles) ont l’autorisation d’accéder à une ressource. Les listes de contrôle d’accès sont similaires aux politiques basées sur les ressources, bien qu’elles n’utilisent pas le format de document de politique JSON.

Contrôle d'accès basé sur les attributs (ABAC) avec AWS AppSync

Prise en charge d’ABAC (identifications dans les politiques)	Partielle

Le contrôle d’accès basé sur les attributs (ABAC) est une politique d’autorisation qui définit des autorisations en fonction des attributs. Dans AWS, ces attributs sont appelés balises. Vous pouvez associer des balises aux entités IAM (utilisateurs ou rôles) et à de nombreuses AWS ressources. L’étiquetage des entités et des ressources est la première étape d’ABAC. Vous concevez ensuite des politiques ABAC pour autoriser des opérations quand l’identification du principal correspond à celle de la ressource à laquelle il tente d’accéder.

L’ABAC est utile dans les environnements qui connaissent une croissance rapide et pour les cas où la gestion des politiques devient fastidieuse.

Pour contrôler l’accès basé sur des étiquettes, vous devez fournir les informations d’étiquette dans l’élément de condition d’une politique utilisant les clés de condition `aws:ResourceTag/`*`key-name`*, `aws:RequestTag/`*`key-name`* ou `aws:TagKeys`.

Si un service prend en charge les trois clés de condition pour tous les types de ressources, alors la valeur pour ce service est Oui. Si un service prend en charge les trois clés de condition pour certains types de ressources uniquement, la valeur est Partielle.

Pour plus d’informations sur l’ABAC, consultez Qu’est-ce que le contrôle d’accès basé sur les attributs (ABAC) ? dans le Guide de l’utilisateur IAM. Pour accéder à un didacticiel décrivant les

étapes de configuration de l'ABAC, consultez Utilisation du contrôle d'accès par attributs (ABAC) dans le Guide de l'utilisateur IAM.

Utilisation d'informations d'identification temporaires avec AWS AppSync

Prend en charge les informations d'identification temporaires	Oui

Certains Services AWS ne fonctionnent pas lorsque vous vous connectez à l'aide d'informations d'identification temporaires. Pour plus d'informations, y compris celles qui Services AWS fonctionnent avec des informations d'identification temporaires, consultez Services AWS la section relative à l'utilisation d'IAM dans le guide de l'utilisateur d'IAM.

Vous utilisez des informations d'identification temporaires si vous vous connectez à l' AWS Management Console aide d'une méthode autre qu'un nom d'utilisateur et un mot de passe. Par exemple, lorsque vous accédez à AWS l'aide du lien d'authentification unique (SSO) de votre entreprise, ce processus crée automatiquement des informations d'identification temporaires. Vous créez également automatiquement des informations d'identification temporaires lorsque vous vous connectez à la console en tant qu'utilisateur, puis changez de rôle. Pour plus d'informations sur le changement de rôle, consultez Changement de rôle (console) dans le Guide de l'utilisateur IAM.

Vous pouvez créer manuellement des informations d'identification temporaires à l'aide de l' AWS API AWS CLI or. Vous pouvez ensuite utiliser ces informations d'identification temporaires pour y accéder AWS. AWS recommande de générer dynamiquement des informations d'identification temporaires au lieu d'utiliser des clés d'accès à long terme. Pour plus d'informations, consultez Informations d'identification de sécurité temporaires dans IAM.

Transférer les sessions d'accès pour AWS AppSync

Prend en charge les sessions d'accès direct (FAS)	Partielle

Lorsque vous utilisez un utilisateur ou un rôle IAM pour effectuer des actions AWS, vous êtes considéré comme un mandant. Lorsque vous utilisez certains services, vous pouvez effectuer une action qui initie une autre action dans un autre service. FAS utilise les autorisations du principal appelant et Service AWS, associées Service AWS à la demande, pour adresser des demandes aux

services en aval. Les demandes FAS ne sont effectuées que lorsqu'un service reçoit une demande qui nécessite des interactions avec d'autres personnes Services AWS ou des ressources pour être traitée. Dans ce cas, vous devez disposer d'autorisations nécessaires pour effectuer les deux actions. Pour plus de détails sur une politique lors de la formulation de demandes FAS, consultez Transmission des sessions d'accès.

Fonctions du service pour AWS AppSync

Prend en charge les fonctions de service	Non

Une fonction du service est un rôle IAM qu'un service endosse pour accomplir des actions en votre nom. Un administrateur IAM peut créer, modifier et supprimer une fonction du service à partir d'IAM. Pour plus d'informations, consultez Création d'un rôle pour la délégation d'autorisations à un Service AWS dans le Guide de l'utilisateur IAM.

> **Warning**
>
> La modification des autorisations associées à un rôle de service peut perturber AWS AppSync les fonctionnalités. Modifiez les rôles de service uniquement lorsque AWS AppSync vous êtes invité à le faire.

Rôles liés à un service pour AWS AppSync

Prend en charge les rôles liés à un service	Partielle

Un rôle lié à un service est un type de rôle de service lié à un. Service AWS Le service peut endosser le rôle afin d'effectuer une action en votre nom. Les rôles liés au service apparaissent dans votre Compte AWS fichier et appartiennent au service. Un administrateur IAM peut consulter, mais ne peut pas modifier, les autorisations concernant les rôles liés à un service.

Pour plus de détails sur la création ou la gestion des rôles liés à un service, consultez la section AWS Services compatibles avec IAM dans le Guide de l'utilisateur d'IAM. Recherchez un service dans le tableau qui inclut un `Yes` dans la colonne Rôle lié à un service. Choisissez le lien Oui pour consulter la documentation du rôle lié à ce service.

Stratégies basées sur l’identité pour AWS AppSync

Par défaut, les utilisateurs et les rôles ne sont pas autorisés à créer ou à modifier AWS AppSync des ressources. Ils ne peuvent pas non plus effectuer de tâches à l'aide de l'API AWS Management Console, AWS Command Line Interface (AWS CLI) ou de AWS l'API. Pour octroyer aux utilisateurs des autorisations d’effectuer des actions sur les ressources dont ils ont besoin, un administrateur IAM peut créer des politiques IAM. L’administrateur peut ensuite ajouter les politiques IAM aux rôles et les utilisateurs peuvent assumer les rôles.

Pour apprendre à créer une politique basée sur l’identité IAM à l’aide de ces exemples de documents de politique JSON, consultez Création de politiques dans l’onglet JSON dans le Guide de l’utilisateur IAM.

Pour plus de détails sur les actions et les types de ressources définis par AWS AppSync, y compris le format des ARN pour chacun des types de ressources, voir Actions, ressources et clés de condition AWS AppSync dans la référence d'autorisation de service.

Pour connaître les meilleures pratiques en matière de création et de configuration de politiques basées sur l'identité IAM, consultez. the section called “Bonnes pratiques en matière de politique IAM”

Pour obtenir la liste des politiques basées sur l'identité IAM pour AWS AppSync, voir. AWS politiques gérées pour AWS AppSync

Rubriques

- Utilisation de la console AWS AppSync
- Autorisation accordée aux utilisateurs pour afficher leurs propres autorisations
- Accès à un compartiment Amazon S3
- Afficher AWS AppSync les widgets en fonction des balises
- AWS politiques gérées pour AWS AppSync

Utilisation de la console AWS AppSync

Pour accéder à la AWS AppSync console, vous devez disposer d'un ensemble minimal d'autorisations. Ces autorisations doivent vous permettre de répertorier et d'afficher les détails AWS AppSync des ressources de votre Compte AWS. Si vous créez une stratégie basée sur l’identité qui est plus restrictive que l’ensemble minimum d’autorisations requis, la console ne fonctionnera pas comme prévu pour les entités (utilisateurs ou rôles) tributaires de cette stratégie.

Il n'est pas nécessaire d'accorder des autorisations de console minimales aux utilisateurs qui appellent uniquement l'API AWS CLI ou l' AWS API. Autorisez plutôt l'accès à uniquement aux actions qui correspondent à l'opération d'API qu'ils tentent d'effectuer.

Pour garantir que les utilisateurs et les rôles IAM peuvent toujours utiliser la AWS AppSync console, associez également la politique AWS AppSync `ConsoleAccess` ou la politique `ReadOnly` AWS gérée aux entités. Pour plus d'informations, consultez Ajout d'autorisations à un utilisateur dans le Guide de l'utilisateur IAM.

Autorisation accordée aux utilisateurs pour afficher leurs propres autorisations

Cet exemple montre comment créer une politique qui permet aux utilisateurs IAM d'afficher les politiques en ligne et gérées attachées à leur identité d'utilisateur. Cette politique inclut les autorisations permettant d'effectuer cette action sur la console ou par programmation à l'aide de l'API AWS CLI or AWS .

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
```

```
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Accès à un compartiment Amazon S3

Dans cet exemple, vous souhaitez accorder à un utilisateur IAM de votre AWS compte l'accès à l'un de vos compartiments Amazon S3. `examplebucket` Vous souhaitez également autoriser l'utilisateur à ajouter, mettre à jour et supprimer des objets.

En plus de l'octroi des autorisations `s3:PutObject`, `s3:GetObject` et `s3:DeleteObject` à l'utilisateur, la stratégie octroie aussi les autorisations `s3:ListAllMyBuckets`, `s3:GetBucketLocation` et `s3:ListBucket`. Ces conditions supplémentaires sont requises par la console. De la même manière, les actions `s3:PutObjectAcl` et `s3:GetObjectAcl` sont nécessaires pour que les objets puissent être copiés, coupés et collés dans la console. Pour un exemple de procédure pas à pas qui accorde des autorisations aux utilisateurs et les teste à l'aide de la console, consultez Un exemple de procédure pas à pas : utilisation de politiques utilisateur pour contrôler l'accès à votre compartiment.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"ListBucketsInConsole",
         "Effect":"Allow",
         "Action":[
            "s3:ListAllMyBuckets"
         ],
         "Resource":"arn:aws:s3:::*"
      },
      {
         "Sid":"ViewSpecificBucketInfo",
         "Effect":"Allow",
         "Action":[
            "s3:ListBucket",
            "s3:GetBucketLocation"
         ],
         "Resource":"arn:aws:s3:::examplebucket"
```

```
      },
      {
         "Sid":"ManageBucketContents",
         "Effect":"Allow",
         "Action":[
            "s3:PutObject",
            "s3:PutObjectAcl",
            "s3:GetObject",
            "s3:GetObjectAcl",
            "s3:DeleteObject"
         ],
         "Resource":"arn:aws:s3:::examplebucket/*"
      }
   ]
}
```

Afficher AWS AppSync *les widgets* en fonction des balises

Vous pouvez utiliser des conditions dans votre politique basée sur l'identité pour contrôler l'accès aux AWS AppSync ressources en fonction de balises. Cet exemple montre comment créer une politique qui autorise l'affichage d'un *widget*. Toutefois, l'autorisation n'est accordée que si la balise du *widget* `Owner` a la valeur du nom d'utilisateur de cet utilisateur. Cette politique accorde également les autorisations nécessaires pour réaliser cette action sur la console.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ListWidgetsInConsole",
            "Effect": "Allow",
            "Action": "appsync:ListWidgets",
            "Resource": "*"
        },
        {
            "Sid": "ViewWidgetIfOwner",
            "Effect": "Allow",
            "Action": "appsync:GetWidget",
            "Resource": "arn:aws:appsync:*:*:widget/*",
            "Condition": {
                "StringEquals": {"aws:ResourceTag/Owner": "${aws:username}"}
            }
        }
    ]
```

```
}
```

Vous pouvez rattacher cette politique aux utilisateurs IAM de votre compte. *Si un utilisateur nommé richard-roe tente d'afficher un AWS AppSync widget, celui-ci doit être étiqueté Owner=richard-roe ouowner=richard-roe.* Dans le cas contraire, l'utilisateur se voit refuser l'accès. La clé de condition d'étiquette `0wner` correspond à la fois à `0wner` et à `owner`, car les noms de clé de condition ne sont pas sensibles à la casse. Pour plus d'informations, consultez Conditions pour éléments de politique JSON IAM dans le Guide de l'utilisateur IAM.

AWS politiques gérées pour AWS AppSync

Pour ajouter des autorisations aux utilisateurs, aux groupes et aux rôles, il est plus facile d'utiliser des politiques AWS gérées que de les rédiger vous-même. Il faut du temps et de l'expertise pour créer des politiques gérées par le client IAM qui ne fournissent à votre équipe que les autorisations dont elle a besoin. Pour démarrer rapidement, vous pouvez utiliser nos politiques AWS gérées. Ces politiques couvrent des cas d'utilisation courants et sont disponibles dans votre Compte AWS. Pour plus d'informations sur les politiques AWS gérées, voir les politiques AWS gérées dans le guide de l'utilisateur IAM.

AWS les services maintiennent et mettent à jour les politiques AWS gérées. Vous ne pouvez pas modifier les autorisations dans les politiques AWS gérées. Les services ajoutent parfois des autorisations supplémentaires à une politique AWS gérée pour prendre en charge de nouvelles fonctionnalités. Ce type de mise à jour affecte toutes les identités (utilisateurs, groupes et rôles) auxquelles la politique est attachée. Les services sont plus susceptibles de mettre à jour une politique AWS gérée lorsqu'une nouvelle fonctionnalité est lancée ou lorsque de nouvelles opérations sont disponibles. Les services ne suppriment pas les autorisations d'une politique AWS gérée. Les mises à jour des politiques n'endommageront donc pas vos autorisations existantes.

En outre, AWS prend en charge les politiques gérées pour les fonctions professionnelles qui couvrent plusieurs services. Par exemple, la politique ReadOnlyAccess AWS gérée fournit un accès en lecture seule à tous les AWS services et ressources. Lorsqu'un service lance une nouvelle fonctionnalité, il AWS ajoute des autorisations en lecture seule pour les nouvelles opérations et ressources. Pour obtenir la liste des politiques de fonctions professionnelles et leurs descriptions, consultez la page politiques gérées par AWS pour les fonctions de tâche dans le Guide de l'utilisateur IAM.

AWS politique gérée : AWSAppSyncInvokeFullAccess

Utilisez la politique `AWSAppSyncInvokeFullAccess` AWS gérée pour permettre à vos administrateurs d'accéder au AWS AppSync service via la console ou de manière indépendante.

Vous pouvez associer la politique `AWSAppSyncInvokeFullAccess` à vos identités IAM.

Détails de l'autorisation

Cette politique inclut les autorisations suivantes.

- `AWS AppSync`— Permet un accès administratif complet à toutes les ressources de AWS AppSync

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "appsync:GraphQL",
                "appsync:GetGraphqlApi",
                "appsync:ListGraphqlApis",
                "appsync:ListApiKeys"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS politique gérée : AWSAppSyncSchemaAuthor

Utilisez la politique `AWSAppSyncSchemaAuthor` AWS gérée pour autoriser les utilisateurs IAM à accéder à leurs schémas GraphQL pour créer, mettre à jour et interroger leurs schémas GraphQL. Pour plus d'informations sur ce que les utilisateurs peuvent faire avec ces autorisations, consultezConception d'API GraphQL.

Vous pouvez associer la politique `AWSAppSyncSchemaAuthor` à vos identités IAM.

Détails de l'autorisation

Cette politique inclut les autorisations suivantes.

- `AWS AppSync`— Permet les actions suivantes :
  - Création de schémas GraphQL
  - Permettre la création, la modification et la suppression de types, de résolveurs et de fonctions GraphQL
  - Évaluation de la logique des modèles de demande et de réponse
  - Évaluation du code à l'aide d'un environnement d'exécution et d'un contexte
  - Envoi de requêtes GraphQL aux API GraphQL
  - Récupération de données GraphQL

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "appsync:GraphQL",
                "appsync:CreateResolver",
                "appsync:CreateType",
                "appsync:DeleteResolver",
                "appsync:DeleteType",
                "appsync:GetResolver",
                "appsync:GetType",
                "appsync:GetDataSource",
                "appsync:GetSchemaCreationStatus",
                "appsync:GetIntrospectionSchema",
                "appsync:GetGraphqlApi",
                "appsync:ListTypes",
                "appsync:ListApiKeys",
                "appsync:ListResolvers",
                "appsync:ListDataSources",
                "appsync:ListGraphqlApis",
                "appsync:StartSchemaCreation",
                "appsync:UpdateResolver",
                "appsync:UpdateType",
                "appsync:TagResource",
                "appsync:UntagResource",
                "appsync:ListTagsForResource",
                "appsync:CreateFunction",
```

```
                "appsync:UpdateFunction",
                "appsync:GetFunction",
                "appsync:DeleteFunction",
                "appsync:ListFunctions",
                "appsync:ListResolversByFunction",
                "appsync:EvaluateMappingTemplate",
                "appsync:EvaluateCode"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS politique gérée : AWSAppSyncPushToCloudWatchLogs

AWS AppSync utilise Amazon CloudWatch pour surveiller les performances de votre application en générant des journaux que vous pouvez utiliser pour résoudre et optimiser vos requêtes GraphQL. Pour plus d'informations, consultez Surveillance et journalisation.

Utilisez la politique `AWSAppSyncPushToCloudWatchLogs` AWS gérée pour autoriser le transfert des journaux AWS AppSync vers le CloudWatch compte d'un utilisateur IAM.

Vous pouvez associer la politique `AWSAppSyncPushToCloudWatchLogs` à vos identités IAM.

Détails de l'autorisation

Cette politique inclut les autorisations suivantes.

- `CloudWatch Logs`— Permet AWS AppSync de créer des groupes de journaux et des flux avec des noms spécifiés. AWS AppSynctransmet les événements du journal au flux de journal spécifié.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
```

```
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS politique gérée : AWSAppSyncAdministrator

Utilisez la politique `AWSAppSyncAdministrator` AWS gérée pour permettre à vos administrateurs d'accéder à tout AWS AppSync sauf à la AWS console.

Vous pouvez attacher `AWSAppSyncAdministrator` à vos entités IAM. AWS AppSync associe également cette politique à un rôle de service qui lui permet d'effectuer des actions en votre nom.

Détails de l’autorisation

Cette politique inclut les autorisations suivantes.

- `AWS AppSync`— Permet un accès administratif complet à toutes les ressources de AWS AppSync
- `IAM`— Permet les actions suivantes :
  - Création de rôles liés à un service pour permettre AWS AppSync d'analyser les ressources d'autres services en votre nom
  - Supprimer des rôles liés à un service
  - Transmission de rôles liés à un service à d'autres AWS services pour qu'ils assument le rôle ultérieurement et exécutent des actions en votre nom

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "appsync:*"
```

```
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": [
                        "appsync.amazonaws.com"
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "appsync.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:DeleteServiceLinkedRole",
                "iam:GetServiceLinkedRoleDeletionStatus"
            ],
            "Resource": "arn:aws:iam::*:role/aws-service-role/appsync.amazonaws.com/
AWSServiceRoleForAppSync*"
        }
    ]
}
```

AWS politique gérée : AWSAppSyncServiceRolePolicy

Utilisez la politique `AWSAppSyncServiceRolePolicy` AWS gérée pour autoriser l'accès aux AWS services et aux ressources qui AWS AppSync utilisent ou gèrent.

Vous ne pouvez pas joindre de `AWSAppSyncServiceRolePolicy` à vos entités IAM. Cette politique est associée à un rôle lié à un service qui permet d' AWS AppSync effectuer des actions en votre nom. Pour plus d'informations, consultez Rôles liés à un service pour AWS AppSync.

Détails de l'autorisation

Cette politique inclut les autorisations suivantes.

- `X-Ray`— AWS AppSync utilise AWS X-Ray pour collecter des données sur les demandes effectuées dans le cadre de votre application. Pour plus d'informations, consultez Suivi avecAWS X-Ray.

  Cette politique permet les actions suivantes :

  - Récupération des règles d'échantillonnage et de leurs résultats
  - Envoi de données de trace au daemon X-Ray

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:PutTraceSegments",
                "xray:PutTelemetryRecords",
                "xray:GetSamplingTargets",
                "xray:GetSamplingRules",
                "xray:GetSamplingStatisticSummaries"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

AWS AppSync mises à jour des politiques AWS gérées

Consultez les détails des mises à jour des politiques AWS gérées AWS AppSync depuis que ce service a commencé à suivre ces modifications. Pour recevoir des alertes automatiques concernant les modifications apportées à cette page, abonnez-vous au flux RSS sur la page Historique du AWS AppSync document.

Modification	Description	Date
AWSAppSyncSchemaAuthor : mise à jour d'une stratégie existante	Ajout d'une action `EvaluateCode` de politique permettant aux utilisateurs d'évaluer le code à l'aide d'un environnement d'exécution et d'un contexte.	7 février 2023
AWSAppSyncSchemaAuthor : mise à jour d'une stratégie existante	Des actions de politique ont été ajoutées pour autoriser les fonctions de liste, d'obtention, de création, de mise à jour et de suppression pour une API. Ajout d'une action `EvaluateMappingTemplate` de politique permettant aux utilisateurs d'évaluer la logique du modèle de mappage du résolveur de demandes et de réponses. Des actions politiques ont été ajoutées pour permettre le balisage des ressources.	25 août 2022
AWS AppSync a commencé à suivre les modifications	AWS AppSync a commencé à suivre les modifications apportées AWS à ses politiques gérées.	25 août 2022

Résolution des problèmes AWS AppSync d'identité et d'accès

Utilisez les informations suivantes pour vous aider à diagnostiquer et à résoudre les problèmes courants que vous pouvez rencontrer lorsque vous travaillez avec AWS AppSync IAM.

Je ne suis pas autorisé à effectuer une action dans AWS AppSync

S'il vous AWS Management Console indique que vous n'êtes pas autorisé à effectuer une action, vous devez contacter votre administrateur pour obtenir de l'aide. Votre administrateur est la personne qui vous a fourni votre nom d'utilisateur et votre mot de passe.

L'exemple d'erreur suivant se produit lorsque l'utilisateur IAM `mateojackson` essaie d'utiliser la console pour afficher les détails d'une *`my-example-widget`* ressource fictive, mais qu'il ne dispose pas des `appsync:`*`GetWidget`* autorisations fictives.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 appsync:GetWidget on resource: my-example-widget
```

Dans ce cas, Mateo demande à son administrateur de mettre à jour ses politiques pour lui permettre d'accéder à la ressource *`my-example-widget`* à l'aide de l'action `appsync:`*`GetWidget`*.

Je ne suis pas autorisé à effectuer iam : PassRole

Si vous recevez un message d'erreur indiquant que vous n'êtes pas autorisé à effectuer l'`iam:PassRole`action, vos politiques doivent être mises à jour pour vous permettre de transmettre un rôle AWS AppSync.

Certains vous Services AWS permettent de transmettre un rôle existant à ce service au lieu de créer un nouveau rôle de service ou un rôle lié à un service. Pour ce faire, un utilisateur doit disposer des autorisations nécessaires pour transmettre le rôle au service.

L'exemple d'erreur suivant se produit lorsqu'un utilisateur IAM nommé `marymajor` essaie d'utiliser la console pour effectuer une action dans AWS AppSync. Toutefois, l'action nécessite que le service ait des autorisations accordées par un rôle de service. Mary ne dispose pas des autorisations nécessaires pour transférer le rôle au service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

Dans ce cas, les politiques de Mary doivent être mises à jour pour lui permettre d'exécuter l'action `iam:PassRole`.

Si vous avez besoin d'aide, contactez votre AWS administrateur. Votre administrateur vous a fourni vos informations de connexion.

Je veux afficher mes clés d'accès

Une fois les clés d'accès utilisateur IAM créées, vous pouvez afficher votre ID de clé d'accès à tout moment. Toutefois, vous ne pouvez pas revoir votre clé d'accès secrète. Si vous perdez votre clé d'accès secrète, vous devez créer une nouvelle paire de clés.

Les clés d'accès se composent de deux parties : un ID de clé d'accès (par exemple, `AKIAIOSFODNN7EXAMPLE`) et une clé d'accès secrète (par exemple, `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`). À l'instar d'un nom d'utilisateur et un mot de passe, vous devez utiliser à la fois l'ID de clé d'accès et la clé d'accès secrète pour authentifier vos demandes. Gérez vos clés d'accès de manière aussi sécurisée que votre nom d'utilisateur et votre mot de passe.

> **Important**
>
> Ne communiquez pas vos clés d'accès à un tiers, même pour qu'il vous aide à trouver votre ID utilisateur canonique. Ce faisant, vous pourriez donner à quelqu'un un accès permanent à votre Compte AWS.

Lorsque vous créez une paire de clé d'accès, enregistrez l'ID de clé d'accès et la clé d'accès secrète dans un emplacement sécurisé. La clé d'accès secrète est accessible uniquement au moment de sa création. Si vous perdez votre clé d'accès secrète, vous devez ajouter de nouvelles clés d'accès pour votre utilisateur IAM. Vous pouvez avoir un maximum de deux clés d'accès. Si vous en avez déjà deux, vous devez supprimer une paire de clés avant d'en créer une nouvelle. Pour afficher les instructions, consultez Gestion des clés d'accès dans le Guide de l'utilisateur IAM.

Je suis administrateur et je souhaite autoriser d'autres personnes à accéder AWS AppSync

Pour autoriser d'autres personnes à y accéder AWS AppSync, vous devez créer une entité IAM (utilisateur ou rôle) pour la personne ou l'application qui a besoin d'un accès. Ils utiliseront les informations d'identification de cette entité pour accéder à AWS. Vous devez ensuite associer une politique à l'entité qui lui accorde les autorisations appropriées AWS AppSync.

Pour démarrer immédiatement, consultez Création de votre premier groupe et utilisateur délégué IAM dans le Guide de l'utilisateur IAM.

Je souhaite autoriser des personnes extérieures à mon AWS compte à accéder à mes AWS AppSync ressources

Vous pouvez créer un rôle que les utilisateurs provenant d'autres comptes ou les personnes extérieures à votre organisation pourront utiliser pour accéder à vos ressources. Vous pouvez spécifier qui est autorisé à assumer le rôle. Pour les services qui prennent en charge les politiques basées sur les ressources ou les listes de contrôle d'accès (ACL), vous pouvez utiliser ces politiques pour donner l'accès à vos ressources.

Pour en savoir plus, consultez les éléments suivants :

- Pour savoir si ces fonctionnalités sont prises AWS AppSync en charge, consultezComment AWS AppSync fonctionne avec IAM.
- Pour savoir comment fournir l'accès à vos ressources sur celles Comptes AWS que vous possédez, consultez la section Fournir l'accès à un utilisateur IAM dans un autre utilisateur Compte AWS que vous possédez dans le Guide de l'utilisateur IAM.
- Pour savoir comment fournir l'accès à vos ressources à des tiers Comptes AWS, consultez la section Fournir un accès à des ressources Comptes AWS détenues par des tiers dans le guide de l'utilisateur IAM.
- Pour savoir comment fournir un accès par le biais de la fédération d'identité, consultez Fournir un accès à des utilisateurs authentifiés en externe (fédération d'identité) dans le Guide de l'utilisateur IAM.
- Pour découvrir quelle est la différence entre l'utilisation des rôles et l'utilisation des politiques basées sur les ressources pour l'accès entre comptes, consultez Différence entre les rôles IAM et les politiques basées sur les ressources dans le Guide de l'utilisateur IAM.

Journalisation des appels d' AWS AppSync API avec AWS CloudTrail

AWS AppSync est intégré à AWS CloudTrail un service qui fournit un enregistrement des actions entreprises par un utilisateur, un rôle ou un AWS service dans AWS AppSync. CloudTrail capture les appels d'API AWS AppSync sous forme d'événements. Les appels capturés incluent des appels provenant de la AWS AppSync console et des appels de code vers les opérations de l'

AWS AppSync API. Si vous créez un suivi, vous pouvez activer la diffusion continue d' CloudTrail événements vers un compartiment Amazon S3, y compris les événements pour AWS AppSync. Si vous ne configurez pas de suivi, vous pouvez toujours consulter les événements les plus récents dans la CloudTrail console dans Historique des événements. À l'aide des informations collectées par CloudTrail, vous pouvez déterminer la demande qui a été faite AWS AppSync, l'adresse IP à partir de laquelle la demande a été faite, qui a fait la demande, quand elle a été faite et des détails supplémentaires.

Pour en savoir plus CloudTrail, consultez le guide de AWS CloudTrail l'utilisateur.

AWS AppSync informations dans CloudTrail

CloudTrail est activé sur votre AWS compte lorsque vous le créez. Lorsqu'une activité se produit dans AWS AppSync, cette activité est enregistrée dans un CloudTrail événement avec d'autres événements de AWS service dans l'historique des événements. Vous pouvez consulter, rechercher et télécharger les événements récents dans votre AWS compte. Pour plus d'informations, consultez la section Affichage des événements à l'aide de l'historique des CloudTrail événements.

Pour un enregistrement continu des événements de votre AWS compte, y compris des événements pour AWS AppSync, créez un parcours. Un suivi permet CloudTrail de fournir des fichiers journaux à un compartiment Amazon S3. Par défaut, lorsque vous créez un parcours dans la console, celui-ci s'applique à toutes les AWS régions. Le journal enregistre les événements de toutes les régions de la AWS partition et transmet les fichiers journaux au compartiment Amazon S3 que vous spécifiez. En outre, vous pouvez configurer d'autres AWS services pour analyser plus en détail les données d'événements collectées dans les CloudTrail journaux et agir en conséquence. Pour plus d’informations, consultez les ressources suivantes :

- Présentation de la création d’un journal de suivi
- CloudTrail services et intégrations pris en charge
- Configuration des notifications Amazon SNS pour CloudTrail
- Réception de fichiers CloudTrail journaux de plusieurs régions et réception de fichiers CloudTrail journaux de plusieurs comptes

AWS AppSync prend en charge la journalisation des appels effectués via l' AWS AppSync API. À l'heure actuelle, les appels à vos API, ainsi que les appels passés aux résolveurs, ne sont pas connectés. AWS AppSync CloudTrail

Chaque événement ou entrée de journal contient des informations sur la personne ayant initié la demande. Les informations relatives à l'identité permettent de déterminer les éléments suivants :

- Si la demande a été faite avec les informations d'identification de l'utilisateur root ou AWS Identity and Access Management (IAM).
- Si la demande a été effectuée avec les informations d'identification de sécurité temporaires d'un rôle ou d'un utilisateur fédéré.
- Si la demande a été faite par un autre AWS service.

Pour plus d'informations, consultez l'élément CloudTrail UserIdentity.

Comprendre les entrées du fichier AWS AppSync journal

Un suivi est une configuration qui permet de transmettre des événements sous forme de fichiers journaux à un compartiment Amazon S3 que vous spécifiez. CloudTrail les fichiers journaux contiennent une ou plusieurs entrées de journal. Un événement représente une demande unique provenant de n'importe quelle source et inclut des informations sur l'action demandée, la date et l'heure de l'action, les paramètres de la demande, etc. CloudTrail les fichiers journaux ne constituent pas une trace ordonnée des appels d'API publics, ils n'apparaissent donc pas dans un ordre spécifique.

L'exemple suivant montre une entrée de CloudTrail journal qui illustre l'`GetGraphqlApi`action effectuée via la AWS AppSync console :

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "ABCDEFXAMPLEPRINCIPAL:nikkiwolf",
        "arn": "arn:aws:sts::111122223333:assumed-role/admin/nikkiwolf",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AIDAJ45Q7YFFAREXAMPLE",
                "arn": "arn:aws:iam::111122223333:role/admin",
                "accountId": "111122223333",
                "userName": "admin"
            },
```

```
            "webIdFederationData": {},
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2021-03-12T22:41:48Z"
            }
        }
    },
    "eventTime": "2021-03-12T22:46:18Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "GetGraphqlApi",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "203.0.113.69",
    "userAgent": "aws-internal/3 aws-sdk-java/1.11.942
 Linux/4.9.230-0.1.ac.223.84.332.metal1.x86_64 OpenJDK_64-Bit_Server_VM/25.282-b08
 java/1.8.0_282 vendor/Oracle_Corporation",
    "requestParameters": {
        "apiId": "xhxt3typtfnmidkhcexampleid"
    },
    "responseElements": null,
    "requestID": "2fc43a35-a552-4b5d-be6e-12553a03dd12",
    "eventID": "b95b0ad9-8c71-4252-a2ec-5dc2fe5f8ae8",
    "readOnly": true,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "eventCategory": "Management",
    "recipientAccountId": "111122223333"
}
```

L'exemple suivant montre une entrée de CloudTrail journal qui illustre l'`CreateApikey`action effectuée par le biais de AWS CLI :

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "ABCDEFXAMPLEPRINCIPAL",
        "arn": "arn:aws:iam::111122223333:user/nikkiwolf",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "nikkiwolf"
    },
    "eventTime": "2021-03-12T22:49:10Z",
    "eventSource": "appsync.amazonaws.com",
```

```
    "eventName": "CreateApiKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "203.0.113.69",
    "userAgent": "aws-cli/2.0.11 Python/3.7.4 Darwin/18.7.0 botocore/2.0.0dev15",
    "requestParameters": {
        "apiId": "xhxt3typtfnmidkhcexampleid"
    },
    "responseElements": {
        "apiKey": {
            "id": "***",
            "expires": 1616191200,
            "deletes": 1621375200
        }
    },
    "requestID": "e152190e-04ba-4d0a-ae7b-6bfc0bcea6af",
    "eventID": "ba3f39e0-9d87-41c5-abbb-2000abcb6013",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "eventCategory": "Management",
    "recipientAccountId": "111122223333"
}
```

Bonnes pratiques de sécurité pour AWS AppSync

Pour sécuriser AWS AppSync , il ne suffit pas d'actionner quelques leviers ou de configurer la journalisation. Les sections suivantes décrivent les meilleures pratiques en matière de sécurité qui varient en fonction de la manière dont vous utilisez le service.

Comprendre les méthodes d'authentification

AWS AppSync fournit plusieurs méthodes pour authentifier vos utilisateurs auprès de vos API GraphQL. Chaque méthode comporte des compromis en termes de sécurité, d'auditabilité et de facilité d'utilisation.

Les méthodes d'authentification courantes suivantes sont disponibles :

- Les groupes d'utilisateurs Amazon Cognito permettent à votre API GraphQL d'utiliser les attributs utilisateur pour un contrôle d'accès et un filtrage précis.
- Les jetons d'API ont une durée de vie limitée et conviennent aux systèmes automatisés, tels que les systèmes d'intégration continue et l'intégration avec des API externes.

- AWS Identity and Access Management (IAM) convient aux applications internes gérées dans votre Compte AWS.
- OpenID Connect vous permet de contrôler et de fédérer les accès grâce au protocole OpenID Connect.

Pour plus d'informations sur l'authentification et l'autorisation dans AWS AppSync, consultezAutorisation et authentification.

Utiliser le protocole TLS pour les résolveurs HTTP

Lorsque vous utilisez des résolveurs HTTP, veillez à utiliser des connexions sécurisées par TLS (HTTPS) dans la mesure du possible. Pour obtenir la liste complète des certificats TLS approuvés AWS AppSync , consultezAutorités de certification (CA) reconnues parAWS AppSyncpour les points de terminaison HTTPS.

Utilisez des rôles avec le moins d'autorisations possible

Lorsque vous utilisez des résolveurs tels que le résolveur DynamoDB, utilisez des rôles qui fournissent la vue la plus restrictive de vos ressources, tels que vos tables Amazon DynamoDB.

Bonnes pratiques en matière de politique IAM

Les politiques basées sur l'identité déterminent si quelqu'un peut créer, accéder ou supprimer AWS AppSync des ressources dans votre compte. Ces actions peuvent entraîner des frais pour votre Compte AWS. Lorsque vous créez ou modifiez des politiques basées sur l’identité, suivez ces instructions et recommandations :

- Commencez AWS par les politiques gérées et passez aux autorisations du moindre privilège : pour commencer à accorder des autorisations à vos utilisateurs et à vos charges de travail, utilisez les politiques AWS gérées qui accordent des autorisations pour de nombreux cas d'utilisation courants. Ils sont disponibles dans votre Compte AWS. Nous vous recommandons de réduire davantage les autorisations en définissant des politiques gérées par les AWS clients spécifiques à vos cas d'utilisation. Pour plus d’informations, consultez politiques gérées par AWS ou politiques gérées par AWS pour les activités professionnelles dans le Guide de l’utilisateur IAM.
- Accorder les autorisations de moindre privilège : lorsque vous définissez des autorisations avec des politiques IAM, accordez uniquement les autorisations nécessaires à l’exécution d’une seule tâche. Pour ce faire, vous définissez les actions qui peuvent être entreprises sur des

ressources spécifiques dans des conditions spécifiques, également appelées autorisations de moindre privilège. Pour plus d'informations sur l'utilisation de IAM pour appliquer des autorisations, consultez politiques et autorisations dans IAM dans le Guide de l'utilisateur IAM.

- Utiliser des conditions dans les politiques IAM pour restreindre davantage l'accès : vous pouvez ajouter une condition à vos politiques afin de limiter l'accès aux actions et aux ressources. Par exemple, vous pouvez écrire une condition de politique pour spécifier que toutes les demandes doivent être envoyées via SSL. Vous pouvez également utiliser des conditions pour accorder l'accès aux actions de service si elles sont utilisées par le biais d'un service spécifique Service AWS, tel que AWS CloudFormation. Pour plus d'informations, consultez Conditions pour éléments de politique JSON IAM dans le Guide de l'utilisateur IAM.
- Utilisez IAM Access Analyzer pour valider vos politiques IAM afin de garantir des autorisations sécurisées et fonctionnelles : IAM Access Analyzer valide les politiques nouvelles et existantes de manière à ce que les politiques IAM respectent le langage de politique IAM (JSON) et les bonnes pratiques IAM. IAM Access Analyzer fournit plus de 100 vérifications de politiques et des recommandations exploitables pour vous aider à créer des politiques sécurisées et fonctionnelles. Pour plus d'informations, consultez Validation de politique IAM Access Analyzer dans le Guide de l'utilisateur IAM.
- Exiger l'authentification multifactorielle (MFA) : si vous avez un scénario qui nécessite des utilisateurs IAM ou un utilisateur root, activez l'authentification MFA pour une sécurité accrue. Compte AWS Pour exiger le MFA lorsque des opérations d'API sont appelées, ajoutez des conditions MFA à vos politiques. Pour plus d'informations, consultez Configuration de l'accès aux API protégé par MFA dans le Guide de l'utilisateur IAM.

Pour plus d'informations sur les bonnes pratiques dans IAM, consultez Bonnes pratiques de sécurité dans IAM dans le Guide de l'utilisateur IAM.

Référence du résolveur () JavaScript

Les sections suivantes décrivent le `APPSYNC_JS` moteur d'exécution et les JavaScript résolveurs.

Rubriques

JavaScript vue d'ensemble des résolveurs

AWS AppSync vous permet de répondre aux requêtes GraphQL en effectuant des opérations sur vos sources de données. Pour chaque champ GraphQL sur lequel vous souhaitez exécuter une requête, une mutation ou un abonnement, un résolveur doit être joint.

Les résolveurs sont les connecteurs entre GraphQL et une source de données. Ils expliquent AWS AppSync comment traduire une requête GraphQL entrante en instructions pour votre source de données principale et comment traduire la réponse de cette source de données en réponse GraphQL. Avec AWS AppSync, vous pouvez écrire vos résolveurs en les utilisant JavaScript et les exécuter dans l'environnement AWS AppSync (`APPSYNC_JS`).

AWS AppSync vous permet d'écrire des résolveurs d'unités ou des résolveurs de pipeline composés de plusieurs AWS AppSync fonctions dans un pipeline.

Fonctionnalités d'exécution prises en charge

Le AWS AppSync JavaScript moteur d'exécution fournit un sous-ensemble de JavaScript bibliothèques, d'utilitaires et de fonctionnalités. Pour une liste complète des fonctionnalités prises en

charge par le APPSYNC_JS moteur d'exécution, voir Fonctionnalités JavaScript d'exécution pour les résolveurs et les fonctions.

Résolveurs d'unités

Un résolveur d'unités est composé d'un code qui définit un gestionnaire de demandes et de réponses qui sont exécutés sur une source de données. Le gestionnaire de demandes prend un objet de contexte comme argument et renvoie la charge utile de la demande utilisée pour appeler votre source de données. Le gestionnaire de réponses reçoit une charge utile en retour de la source de données avec le résultat de la demande exécutée. Le gestionnaire de réponse transforme la charge utile en réponse GraphQL pour résoudre le champ GraphQL. Dans l'exemple ci-dessous, un résolveur extrait un élément d'une source de données DynamoDB :

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  return ddb.get({ key: { id: ctx.args.id } });
}

export const response = (ctx) => ctx.result;
```

Anatomie d'un résolveur de JavaScript pipeline

Un résolveur de pipeline est composé d'un code qui définit un gestionnaire de requêtes et de réponses et d'une liste de fonctions. Chaque fonction possède un gestionnaire de requêtes et de réponses qu'elle exécute sur une source de données. Lorsqu'un résolveur de pipeline délègue des exécutions à une liste de fonctions, il n'est donc lié à aucune source de données. Les résolveurs d'unité et les fonctions sont des primitifs qui exécutent l'opération sur les sources de données.

Gestionnaire de requêtes Pipeline Resolver

Le gestionnaire de requêtes d'un résolveur de pipeline (étape précédente) vous permet d'exécuter une certaine logique de préparation avant d'exécuter les fonctions définies.

Liste des fonctions

La liste des fonctions d'un résolveur de pipeline est exécutée dans l'ordre. Le résultat de l'évaluation du gestionnaire de demandes du résolveur de pipeline est mis à la disposition de la première fonction sous forme de. `ctx.prev.result` Le résultat de l'évaluation de chaque fonction est disponible pour la fonction suivante sous forme de`ctx.prev.result`.

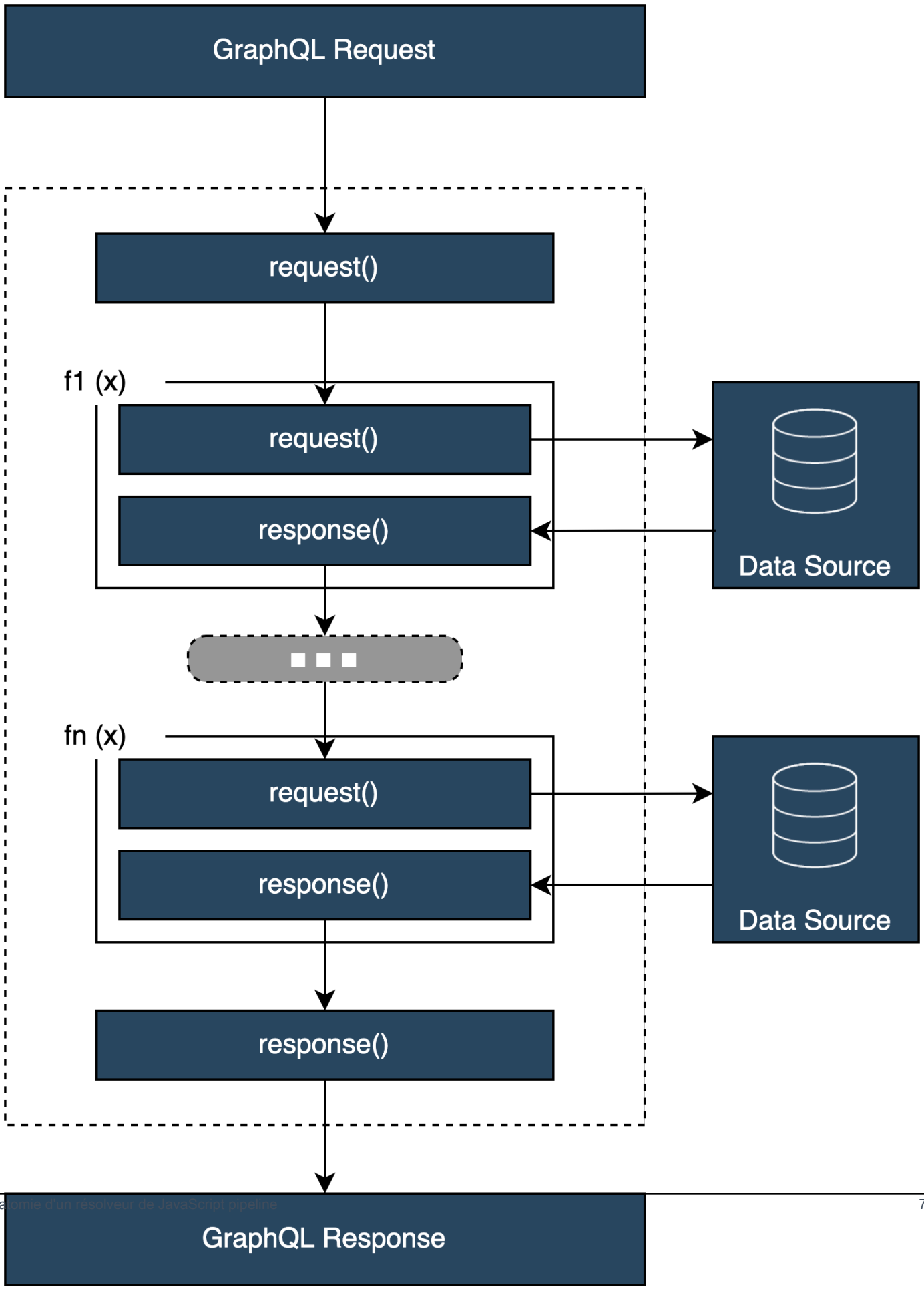
Gestionnaire de réponses Pipeline Resolver

Le gestionnaire de réponse d'un résolveur de pipeline vous permet d'exécuter une certaine logique finale entre la sortie de la dernière fonction et le type de champ GraphQL attendu. La sortie de la dernière fonction de la liste des fonctions est disponible dans le gestionnaire de réponse du résolveur de pipeline sous `ctx.prev.result` la forme ou. `ctx.result`

Flux d'exécution

Étant donné qu'un résolveur de pipeline est composé de deux fonctions, la liste ci-dessous représente le flux d'exécution lorsque le résolveur est invoqué :

1. Gestionnaire de requêtes Pipeline Resolver
2. Fonction 1 : gestionnaire de demandes de fonctions
3. Fonction 1 : Appel de source de données
4. Fonction 1 : gestionnaire de réponse fonctionnelle
5. Fonction 2 : gestionnaire de demandes de fonctions
6. Fonction 2 : Appel de source de données
7. Fonction 2 : gestionnaire de réponse fonctionnelle
8. Gestionnaire de réponses Pipeline Resolver


GraphQL Request
request()
f1 (x)
request()
response()
Data Source
fn (x)
request()
response()
Data Source
response()
GraphQL Response

Utilitaires intégrés utiles à l'**APPSYNC_JS**exécution

Les utilitaires suivants peuvent vous aider si vous travaillez avec des résolveurs de pipeline.

ctx.stash

Le stash est un objet mis à disposition dans chaque résolveur et chaque gestionnaire de demandes et de réponses de fonctions. La même instance de stash passe par une seule exécution du résolveur. Cela signifie que vous pouvez utiliser le stash pour transmettre des données arbitraires entre les gestionnaires de requêtes et de réponses et entre les fonctions d'un résolveur de pipeline. Vous pouvez tester la réserve comme un JavaScript objet normal.

ctx.prev.result

`ctx.prev.result` représente le résultat de l'opération précédente exécutée dans le pipeline. Si l'opération précédente était le gestionnaire de demandes du résolveur de pipeline, elle `ctx.prev.result` est alors mise à la disposition de la première fonction de la chaîne. Si l'opération précédente est la première fonction, alors `ctx.prev.result` représente le résultat de la première fonction et il est disponible pour la seconde fonction du pipeline. Si l'opération précédente était la dernière fonction, elle `ctx.prev.result` représente la sortie de la dernière fonction et est mise à la disposition du gestionnaire de réponse du résolveur de pipeline.

util.error

L'utilitaire `util.error` est utile pour envoyer une erreur de champ. `util.error`L'utilisation à l'intérieur d'un gestionnaire de demandes ou de réponses de fonction génère immédiatement une erreur de champ, ce qui empêche l'exécution des fonctions suivantes. Pour plus de détails et pour d'autres `util.error` signatures, consultez les fonctionnalités JavaScript d'exécution pour les résolveurs et les fonctions.

Erreur Util.AppendError

`util.appendError`est similaire à`util.error()`, avec la principale différence qu'il n'interrompt pas l'évaluation du gestionnaire. Au lieu de cela, il signale qu'une erreur s'est produite dans le champ, mais permet au gestionnaire d'être évalué et, par conséquent, de renvoyer des données. L'utilisation de `util.appendError` dans une fonction ne perturbera pas le flux d'exécution du pipeline. Pour plus de détails et pour d'autres `util.error` signatures, consultez les fonctionnalités JavaScript d'exécution relatives aux résolveurs et aux fonctions.

Temps d'exécution. Retour anticipé

La `runtime.earlyReturn` fonction vous permet de revenir prématurément à n'importe quelle fonction de requête. L'utilisation de `runtime.earlyReturn` l'intérieur d'un gestionnaire de demandes de résolution sera renvoyée par le résolveur. L'appeler depuis un gestionnaire de demandes de AWS AppSync fonction retournera depuis la fonction et poursuivra l'exécution vers la fonction suivante du pipeline ou vers le gestionnaire de réponse du résolveur.

Écrire des résolveurs de pipeline

Un résolveur de pipeline possède également un gestionnaire de requêtes et de réponses entourant l'exécution des fonctions du pipeline : son gestionnaire de demandes est exécuté avant la demande de la première fonction, et son gestionnaire de réponse est exécuté après la réponse de la dernière fonction. Le gestionnaire de requêtes Resolver peut configurer les données à utiliser par les fonctions du pipeline. Le gestionnaire de réponse du résolveur est chargé de renvoyer les données correspondant au type de sortie du champ GraphQL. Dans l'exemple ci-dessous, un gestionnaire de demandes de résolution définit `allowedGroups` ; les données renvoyées doivent appartenir à l'un de ces groupes. Cette valeur peut être utilisée par les fonctions du résolveur pour demander des données. Le gestionnaire de réponses du résolveur effectue une dernière vérification et filtre le résultat pour s'assurer que seuls les éléments appartenant aux groupes autorisés sont renvoyés.

```
import { util } from '@aws-appsync/utils';

/**
 * Called before the request function of the first AppSync function in the pipeline.
 *  @param ctx the context object holds contextual information about the function
 invocation.
 */
export function request(ctx) {
  ctx.stash.allowedGroups = ['admin'];
  ctx.stash.startedAt = util.time.nowISO8601();
  return {};
}
/**
 * Called after the response function of the last AppSync function in the pipeline.
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function response(ctx) {
  const result = [];
  for (const item of ctx.prev.result) {
```

```
    if (ctx.stash.allowedGroups.indexOf(item.group) > -1) result.push(item);
  }
  return result;
}
```

AWS AppSyncFonctions d'écriture

AWS AppSync les fonctions vous permettent d'écrire une logique commune que vous pouvez réutiliser dans plusieurs résolveurs de votre schéma. Par exemple, vous pouvez avoir une AWS AppSync fonction appelée `QUERY_ITEMS` qui est chargée de demander des éléments à partir d'une source de données Amazon DynamoDB. Pour les résolveurs avec lesquels vous souhaitez interroger des éléments, ajoutez simplement la fonction au pipeline du résolveur et fournissez l'index de requête à utiliser. La logique n'a pas besoin d'être réimplémentée.

Écrire du code

Supposons que vous souhaitiez associer un résolveur de pipeline à un champ nommé `getPost(id:ID!)` qui renvoie un `Post` type provenant d'une source de données Amazon DynamoDB avec la requête GraphQL suivante :

```
getPost(id:1){
    id
    title
    content
}
```

Tout d'abord, attachez un résolveur simple à l'`Query.getPost`aide du code ci-dessous. Il s'agit d'un exemple de code de résolution simple. Aucune logique n'est définie dans le gestionnaire de demandes, et le gestionnaire de réponse renvoie simplement le résultat de la dernière fonction.

```
/**
 * Invoked **before** the request handler of the first AppSync function in the
 pipeline.
 * The resolver `request` handler allows to perform some preparation logic
 * before executing the defined functions in your pipeline.
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function request(ctx) {
  return {}
```

```
}

/**
 * Invoked **after** the response handler of the last AppSync function in the pipeline.
 * The resolver `response` handler allows to perform some final evaluation logic
 * from the output of the last function to the expected GraphQL field type.
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function response(ctx) {
  return ctx.prev.result
}
```

Définissez ensuite la fonction GET_ITEM qui extrait un article de votre source de données :

```
import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'

/**
 * Request a single item from the attached DynamoDB table datasource
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function request(ctx) {
 const { id } = ctx.args
 return ddb.get({ key: { id } })
}

/**
 * Returns the result
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function response(ctx) {
 const { error, result } = ctx
 if (error) {
  return util.appendError(error.message, error.type, result)
 }
 return ctx.result
}
```

En cas d'erreur lors de la requête, le gestionnaire de réponse de la fonction ajoute une erreur qui sera renvoyée au client appelant dans la réponse GraphQL. Ajoutez la GET_ITEM fonction à la liste

des fonctions de votre résolveur. Lorsque vous exécutez la requête, le gestionnaire de requêtes de la `GET_ITEM` fonction utilise les utilitaires fournis par le module AWS AppSync DynamoDB pour créer une `DynamoDBGetItem` demande en utilisant la clé. `id` `ddb.get({ key: { id } })`génère l'`GetItem`opération appropriée :

```
{
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "1" }
    }
}
```

AWS AppSyncutilise la demande pour récupérer les données depuis Amazon DynamoDB. Une fois les données renvoyées, elles sont gérées par le gestionnaire de réponses de la `GET_ITEM` fonction, qui vérifie les erreurs puis renvoie le résultat.

```
{
  "result" : {
    "id": 1,
    "title": "hello world",
    "content": "<long story>"
  }
}
```

Enfin, le gestionnaire de réponses du résolveur renvoie directement le résultat.

Travailler avec des erreurs

Si une erreur se produit dans votre fonction lors d'une demande, elle sera disponible dans le gestionnaire de réponse de votre fonction dans`ctx.error`. Vous pouvez ajouter l'erreur à votre réponse GraphQL à l'aide `util.appendError` de l'utilitaire. Vous pouvez rendre l'erreur accessible aux autres fonctions du pipeline en utilisant le stash. Consultez l'exemple ci-dessous :

```
/**
 * Returns the result
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
```

```
    if (!ctx.stash.errors) ctx.stash.errors = []
    ctx.stash.errors.push(ctx.error)
    return util.appendError(error.message, error.type, result);
  }
  return ctx.result;
}
```

Utilitaires

AWS AppSyncfournit deux bibliothèques qui facilitent le développement de résolveurs avec le `APPSYNC_JS` moteur d'exécution :

- `@aws-appsync/eslint-plugin`- Détecte et corrige les problèmes rapidement pendant le développement.
- `@aws-appsync/utils`- Fournit la validation de type et l'autocomplétion dans les éditeurs de code.

Configuration du plugin eslint

ESLint est un outil qui analyse statiquement votre code pour détecter rapidement les problèmes. Vous pouvez exécuter ESLint dans le cadre de votre pipeline d'intégration continue. `@aws-appsync/eslint-plugin`est un plugin ESLint qui détecte une syntaxe non valide dans votre code lorsque vous tirez parti du `APPSYNC_JS` runtime. Le plugin vous permet d'obtenir rapidement des commentaires sur votre code pendant le développement sans avoir à transférer vos modifications dans le cloud.

`@aws-appsync/eslint-plugin`fournit deux ensembles de règles que vous pouvez utiliser pendant le développement.

« plugin : @aws -appsync/base » configure un ensemble de règles de base que vous pouvez utiliser dans votre projet :

Règle	Description
non asynchrone	Les processus et les promesses asynchrones ne sont pas pris en charge.
sans attente	Les processus et les promesses asynchrones ne sont pas pris en charge.

Règle	Description
pas de cours	Les cours ne sont pas pris en charge.
sans pour	`for`n'est pas pris en charge (sauf pour `for-in` et`for-of`, qui sont pris en charge)
sans continuer	`continue` n'est pas pris en charge.
sans groupes électrogènes	Les générateurs ne sont pas pris en charge.
aucun rendement	`yield` n'est pas pris en charge.
sans étiquettes	Les étiquettes ne sont pas prises en charge.
Non-ça	`this`le mot clé n'est pas pris en charge.
pas d'essai	La structure Try/Catch n'est pas prise en charge.
sans délai	Alors que les boucles ne sont pas prises en charge.
no-disallowed-unary-operators	`++--`, et les opérateurs ~ unaires ne sont pas autorisés.
no-disallowed-binary-operators	L'`instanceof` opérateur n'est pas autorisé.
sans promesse	Les processus et les promesses asynchrones ne sont pas pris en charge.

« plugin : @aws -appsync/recommended » fournit des règles supplémentaires mais vous oblige également à ajouter des TypeScript configurations à votre projet.

Règle	Description
absence de récursion	Les appels de fonction récursifs ne sont pas autorisés.

Règle	Description
no-disallowed-methods	Certaines méthodes ne sont pas autorisée s. Consultez la référence pour un ensemble complet de fonctions intégrées prises en charge.
no-function-passing	Il n'est pas permis de transmettre des fonctions en tant qu'arguments à des fonctions.
no-function-reassign	Les fonctions ne peuvent pas être réattribuées.
no-function-return	Les fonctions ne peuvent pas être la valeur de retour des fonctions.

Pour ajouter le plugin à votre projet, suivez les étapes d'installation et d'utilisation décrites dans Getting Started with ESLint. Ensuite, installez le plugin dans votre projet à l'aide de votre gestionnaire de packages de projet (par exemple, npm, yarn ou pnpm) :

```
$ npm install @aws-appsync/eslint-plugin
```

Dans votre `.eslintrc.{js,yml,json}` fichier, ajoutez « plugin : @aws -appsync/base » ou « plugin : @aws -appsync/recommended » à la propriété. `extends` L'extrait ci-dessous est un exemple de `.eslintrc` configuration de base pour : JavaScript

```
{
  "extends": ["plugin:@aws-appsync/base"]
}
```

Pour utiliser l'ensemble de règles « plugin : @aws -appsync/recommended », installez la dépendance requise :

```
$ npm install -D @typescript-eslint/parser
```

Créez ensuite un `.eslintrc.js` fichier :

```
{
  "parser": "@typescript-eslint/parser",
```

```
  "parserOptions": {
    "ecmaVersion": 2018,
    "project": "./tsconfig.json"
  },
  "extends": ["plugin:@aws-appsync/recommended"]
}
```

Regroupement TypeScript et cartes sources

Tirer parti des bibliothèques et regrouper votre code

Dans votre résolveur et votre code de fonction, vous pouvez tirer parti des bibliothèques personnalisées et externes à condition qu'elles soient conformes aux APPSYNC_JS exigences. Cela permet de réutiliser le code existant dans votre application. Pour utiliser des bibliothèques définies par plusieurs fichiers, vous devez utiliser un outil de regroupement, tel que esbuild, pour combiner votre code dans un seul fichier qui peut ensuite être enregistré dans votre AWS AppSync résolveur ou votre fonction.

Lorsque vous regroupez votre code, gardez les points suivants à l'esprit :

- APPSYNC_JSne prend en charge que les modules ECMAScript (ESM).
- @aws-appsync/*les modules sont intégrés dans votre code APPSYNC_JS et ne doivent pas être fournis avec celui-ci.
- L'environnement APPSYNC_JS d'exécution est similaire à NodeJS dans la mesure où le code ne s'exécute pas dans un environnement de navigateur.
- Vous pouvez inclure une carte source facultative. Cependant, n'incluez pas le contenu source.

  Pour en savoir plus sur les cartes sources, reportez-vous à la section Utilisation des cartes sources.

Par exemple, pour regrouper le code de votre résolveur situé à l'adressesrc/appsync/getPost.resolver.js, vous pouvez utiliser la commande ESbuild CLI suivante :

```
$ esbuild --bundle \
--sourcemap=inline \
--sources-content=false \
--target=esnext \
--platform=node \
--format=esm \
```

```
--external:@aws-appsync/utils \
--outdir=out/appsync \
 src/appsync/getPost.resolver.js
```

Construire votre code et travailler avec TypeScript

TypeScriptest un langage de programmation développé par Microsoft qui offre toutes JavaScript les fonctionnalités ainsi que le système de TypeScript saisie. Vous pouvez l'utiliser TypeScript pour écrire du code sécurisé et détecter les erreurs et les bogues au moment de la compilation avant d'enregistrer votre code dans. AWS AppSync Le `@aws-appsync/utils` package est entièrement dactylographié.

Le `APPSYNC_JS` moteur d'exécution ne prend pas TypeScript directement en charge. Vous devez d'abord transpiler votre TypeScript code dans un JavaScript code compatible avec le `APPSYNC_JS` moteur d'exécution avant de l'enregistrer. AWS AppSync Vous pouvez l'utiliser TypeScript pour écrire votre code dans votre environnement de développement intégré (IDE) local, mais notez que vous ne pouvez pas créer de TypeScript code dans la AWS AppSync console.

Pour commencer, assurez-vous que vous l'avez TypeScriptinstallé dans votre projet. Configurez ensuite vos paramètres de TypeScript transcompilation pour qu'ils fonctionnent avec le `APPSYNC_JS` moteur d'exécution à l'aide de TSConfig. Voici un exemple de `tsconfig.json` fichier de base que vous pouvez utiliser :

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "esnext",
    "module": "esnext",
   "noEmit": true,
   "moduleResolution": "node",
  }
}
```

Vous pouvez ensuite utiliser un outil de regroupement comme esbuild pour compiler et regrouper votre code. Par exemple, dans le cas d'un projet dont AWS AppSync le code se trouve à l'adresse`src/appsync`, vous pouvez utiliser la commande suivante pour compiler et regrouper votre code :

```
$ esbuild --bundle \
```

```
--sourcemap=inline \
--sources-content=false \
--target=esnext \
--platform=node \
--format=esm \
--external:@aws-appsync/utils \
--outdir=out/appsync \
 src/appsync/**/*.ts
```

Utilisation du codegen Amplify

Vous pouvez utiliser la CLI Amplify pour générer les types de votre schéma. Dans le répertoire où se trouve votre `schema.graphql` fichier, exécutez la commande suivante et passez en revue les instructions pour configurer votre codegen :

```
$  npx @aws-amplify/cli codegen add
```

Pour régénérer votre codegen dans certaines circonstances (par exemple, lorsque votre schéma est mis à jour), exécutez la commande suivante :

```
$ npx @aws-amplify/cli codegen
```

Vous pouvez ensuite utiliser les types générés dans le code de votre résolveur. Par exemple, selon le schéma suivant :

```
type Todo {
  id: ID!
  title: String!
  description: String
}

type Mutation {
  createTodo(title: String!, description: String): Todo
}

type Query {
  listTodos: Todo
}
```

Vous pouvez utiliser les types générés dans l'exemple de AWS AppSync fonction suivant :

```
import { Context, util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'
import { CreateTodoMutationVariables, Todo } from './API' // codegen

export function request(ctx: Context<CreateTodoMutationVariables>) {
 ctx.args.description = ctx.args.description ?? 'created on ' + util.time.nowISO8601()
 return ddb.put<Todo>({ key: { id: util.autoId() }, item: ctx.args })
}

export function response(ctx) {
 return ctx.result as Todo
}
```

Utilisation de génériques dans TypeScript

Vous pouvez utiliser des génériques avec plusieurs des types fournis. Par exemple, l'extrait ci-dessous est un Todo type :

```
export type Todo = {
  __typename: "Todo",
  id: string,
  title: string,
  description?: string | null,
};
```

Vous pouvez écrire un résolveur pour un abonnement qui utilise. Todo Dans votre IDE, les définitions de type et les conseils de saisie automatique vous aideront à utiliser correctement l'utilitaire de `toSubscriptionFilter` transformation :

```
import { util, Context, extensions } from '@aws-appsync/utils'
import { Todo } from './API'

export function request(ctx: Context) {
  return {}
}

export function response(ctx: Context) {
  const filter = util.transform.toSubscriptionFilter<Todo>({
    title: { beginsWith: 'hello' },
    description: { contains: 'created' },
  })
  extensions.setSubscriptionFilter(filter)
```

```
  return null
}
```

Linting vos paquets

Vous pouvez automatiquement pelucher vos bundles en important le `esbuild-plugin-eslint` plugin. Vous pouvez ensuite l'activer en fournissant une `plugins` valeur qui active les fonctionnalités eslint. Vous trouverez ci-dessous un extrait qui utilise l' JavaScript API esbuild dans un fichier appelé : `build.mjs`

```
/* eslint-disable */
import { build } from 'esbuild'
import eslint from 'esbuild-plugin-eslint'
import glob from 'glob'
const files = await glob('src/**/*.ts')

await build({
  format: 'esm',
  target: 'esnext',
  platform: 'node',
  external: ['@aws-appsync/utils'],
  outdir: 'dist/',
  entryPoints: files,
  bundle: true,
  plugins: [eslint({ useEslintrc: true })],
})
```

Utilisation de cartes sources

Vous pouvez fournir une carte source en ligne (`sourcemap`) avec votre JavaScript code. Les cartes sources sont utiles lorsque vous regroupez JavaScript ou TypeScript codez et que vous souhaitez voir des références à vos fichiers source d'entrée dans vos journaux et dans vos messages d' JavaScript erreur d'exécution.

Vous `sourcemap` devez apparaître à la fin de votre code. Il est défini par une seule ligne de commentaire qui suit le format suivant :

```
//# sourceMappingURL=data:application/json;base64,<base64 encoded string>
```

Voici un exemple:

```
//# sourceMappingURL=data:application/
json;base64,ewogICJ2ZXJzaW9uIjogMywKICAic291cmNlcyI6IFsibGliLmpzIiwgImNvZGUuanMiXSwKICAibWFwcGl
```

Les cartes sources peuvent être créées avec esbuild. L'exemple ci-dessous vous montre comment utiliser l' JavaScriptAPI esbuild pour inclure une carte source en ligne lors de la création et du regroupement du code :

```
/* eslint-disable */
import { build } from 'esbuild'
import eslint from 'esbuild-plugin-eslint'
import glob from 'glob'
const files = await glob('src/**/*.ts')

await build({
  sourcemap: 'inline',
  sourcesContent: false,

  format: 'esm',
  target: 'esnext',
  platform: 'node',
  external: ['@aws-appsync/utils'],
  outdir: 'dist/',
  entryPoints: files,
  bundle: true,
  plugins: [eslint({ useEslintrc: true })],
})
```

En particulier, les `sourcesContent` options `sourcemap` et spécifient qu'une carte source doit être ajoutée en ligne à la fin de chaque build, mais ne doit pas inclure le contenu source. Par convention, nous vous recommandons de ne pas inclure de contenu source dans votre`sourcemap`. Vous pouvez le désactiver dans esbuild en réglant `sources-content` sur`false`.

Pour illustrer le fonctionnement des cartes sources, consultez l'exemple suivant dans lequel un code de résolution fait référence à des fonctions d'assistance d'une bibliothèque d'assistance. Le code contient des instructions de journal dans le code du résolveur et dans la bibliothèque d'assistance :

. /src/default.resolver.ts (votre résolveur)

```
import { Context } from '@aws-appsync/utils'
import { hello, logit } from './helper'
```

```
export function request(ctx: Context) {
  console.log('start >')
  logit('hello world', 42, true)
  console.log('< end')
  return 'test'
}

export function response(ctx: Context): boolean {
  hello()
  return ctx.prev.result
}
```

.src/helper.ts (un fichier d'assistance)

```
export const logit = (...rest: any[]) => {
  // a special logger
  console.log('[logger]', ...rest.map((r) => `<${r}>`))
}

export const hello = () => {
  // This just returns a simple sentence, but it could do more.
  console.log('i just say hello..')
}
```

Lorsque vous créez et regroupez le fichier de résolution, votre code de résolution inclut une carte source en ligne. Lorsque votre résolveur s'exécute, les entrées suivantes apparaissent dans les CloudWatch journaux :

En regardant les entrées du CloudWatch journal, vous remarquerez que les fonctionnalités des deux fichiers ont été regroupées et s'exécutent simultanément. Le nom de fichier d'origine de chaque fichier est également clairement reflété dans les journaux.

Test

Vous pouvez utiliser la commande `EvaluateCode` API pour tester à distance votre résolveur et vos gestionnaires de fonctions avec des données simulées avant d'enregistrer votre code dans un résolveur ou une fonction. Pour commencer à utiliser la commande, assurez-vous d'avoir ajouté l'`appsync:evaluatecode`autorisation à votre politique. Par exemple :

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateCode",
            "Resource": "arn:aws:appsync:<region>:<account>:*"
        }
    ]
}
```

Vous pouvez utiliser la commande à l'aide de la AWSCLI ou AWSdes SDK. Par exemple, pour tester votre code à l'aide de la CLI, il vous suffit de pointer sur votre fichier, de fournir un contexte et de spécifier le gestionnaire que vous souhaitez évaluer :

```
aws appsync evaluate-code \
  --code file://code.js \
  --function request \
  --context file://context.json \
  --runtime name=APPSYNC_JS,runtimeVersion=1.0.0
```

La réponse contient un `evaluationResult` contenant la charge utile renvoyée par votre gestionnaire. Il contient également un `logs` objet contenant la liste des journaux générés par votre gestionnaire lors de l'évaluation. Cela permet de déboguer facilement l'exécution de votre code et de consulter les informations relatives à votre évaluation pour vous aider à résoudre le problème. Par exemple :

```
{
    "evaluationResult": "{\"operation\":\"PutItem\",\"key\":{\"id\":{\"S\":\"record-id
\"}},\"attributeValues\":{\"owner\":{\"S\":\"John doe\"},\"expectedVersion\":{\"N\":2},
\"authorId\":{\"S\":\"Sammy Davis\"}}}",
    "logs": [
        "INFO - code.js:5:3: \"current id\" \"record-id\"",
```

```
        "INFO - code.js:9:3: \"request evaluated\""
    ]
}
```

Le résultat de l'évaluation peut être analysé au format JSON, ce qui donne :

```
{
  "operation": "PutItem",
  "key": {
    "id": {
      "S": "record-id"
    }
  },
  "attributeValues": {
    "owner": {
      "S": "John doe"
    },
    "expectedVersion": {
      "N": 2
    },
    "authorId": {
      "S": "Sammy Davis"
    }
  }
}
```

À l'aide du SDK, vous pouvez facilement intégrer des tests issus de votre suite de tests pour valider le comportement de votre code. Notre exemple ici utilise le Jest Testing Framework, mais n'importe quelle suite de tests fonctionne. L'extrait suivant montre une exécution de validation hypothétique. Notez que nous nous attendons à ce que la réponse d'évaluation soit un JSON valide. Nous l'utilisons donc `JSON.parse` pour récupérer le JSON à partir de la réponse sous forme de chaîne :

```
const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })
const runtime = {name:'APPSYNC_JS',runtimeVersion:'1.0.0')

test('request correctly calls DynamoDB', async () => {
  const code = fs.readFileSync('./code.js', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)
```

```
  const response = await client.evaluateCode({ code, context, runtime, function:
 'request' }).promise()
  const result = JSON.parse(response.evaluationResult)

  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})
```

Cela donne le résultat suivant :

```
Ran all test suites.
> jest

PASS ./index.test.js
# request correctly calls DynamoDB (543 ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 totalTime: 1.511 s, estimated 2 s
```

Migration de VTL vers JavaScript

AWS AppSync vous permet d'écrire votre logique métier pour vos résolveurs et fonctions à l'aide de VTL ou. JavaScript Dans les deux langages, vous rédigez une logique qui indique au AWS AppSync service comment interagir avec vos sources de données. Avec VTL, vous écrivez des modèles de mappage qui doivent être évalués en une chaîne codée en JSON valide. Avec JavaScript, vous écrivez des gestionnaires de requêtes et de réponses qui renvoient des objets. Vous ne renvoyez pas de chaîne codée en JSON.

Par exemple, utilisez le modèle de mappage VTL suivant pour obtenir un élément Amazon DynamoDB :

```
{
    "operation": "GetItem",
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
    }
}
```

L'utilitaire `$util.dynamodb.toDynamoDBJson` renvoie une chaîne codée en JSON. S'il `$ctx.args.id` est défini sur<id>, le modèle est évalué comme une chaîne codée en JSON valide :

```
{
    "operation": "GetItem",
    "key": {
        "id": {"S": "<id>"},
    }
}
```

Lorsque vous travaillez avec JavaScript, vous n'avez pas besoin d'imprimer des chaînes brutes codées en JSON dans votre code, et l'utilisation d'un utilitaire de ce type n'`toDynamoDBJson`est pas nécessaire. Voici un exemple équivalent du modèle de mappage ci-dessus :

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: {id: util.dynamodb.toDynamoDB(ctx.args.id)}
  };
}
```

Une alternative consiste à utiliser`util.dynamodb.toMapValues`, qui est l'approche recommandée pour gérer un objet de valeurs :

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}
```

Cela permet d'évaluer ce qui suit :

```
{
  "operation": "GetItem",
  "key": {
    "id": {
      "S": "<id>"
    }
  }
}
```

> **Note**
>
> Nous recommandons d'utiliser le module DynamoDB avec les sources de données DynamoDB :
>
> ```
> import * as ddb from '@aws-appsync/utils/dynamodb'
>
> export function request(ctx) {
>  ddb.get({ key: { id: ctx.args.id } })
> }
> ```

Autre exemple, prenez le modèle de mappage suivant pour placer un élément dans une source de données Amazon DynamoDB :

```
{
    "operation" : "PutItem",
    "key" : {
        "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

Lors de son évaluation, cette chaîne de modèle de mappage doit produire une chaîne codée en JSON valide. Lors de l'utilisation JavaScript, votre code renvoie directement l'objet de la requête :

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { id = util.autoId(), ...values } = ctx.args;
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}
```

qui évalue à :

```
{
  "operation": "PutItem",
```

```
  "key": {
    "id": { "S": "2bff3f05-ff8c-4ed8-92b4-767e29fc4e63" }
  },
  "attributeValues": {
    "firstname": { "S": "Shaggy" },
    "age": { "N": 4 }
  }
}
```

> **Note**
>
> Nous recommandons d'utiliser le module DynamoDB avec les sources de données DynamoDB :
>
> ```
> import { util } from '@aws-appsync/utils'
> import * as ddb from '@aws-appsync/utils/dynamodb'
>
> export function request(ctx) {
>  const { id = util.autoId(), ...item } = ctx.args
>  return ddb.put({ key: { id }, item })
> }
> ```

Choisir entre un accès direct à une source de données ou un proxy via une source de données Lambda

Grâce à AWS AppSync l'`APPSYNC_JS`environnement d'exécution, vous pouvez écrire votre propre code qui implémente votre logique métier personnalisée en utilisant des AWS AppSync fonctions pour accéder à vos sources de données. Cela vous permet d'interagir directement avec des sources de données telles qu'Amazon DynamoDB, Aurora OpenSearch Serverless, Service, les API HTTP AWS et d'autres services sans avoir à déployer de services ou d'infrastructures informatiques supplémentaires. AWS AppSync facilite également l'interaction avec une AWS Lambda fonction en configurant une source de données Lambda. Les sources de données Lambda vous permettent d'exécuter une logique métier complexe en utilisant l'ensemble complet des fonctionnalités AWS Lambda de résolution d'une requête GraphQL. Dans la plupart des cas, une AWS AppSync fonction directement connectée à sa source de données cible fournira toutes les fonctionnalités dont vous avez besoin. Dans les situations où vous devez implémenter une logique métier complexe qui n'est

pas prise en charge par l'APPSYNC_JSenvironnement d'exécution, vous pouvez utiliser une source de données Lambda comme proxy pour interagir avec votre source de données cible.

	Intégration directe des sources de données	Source de données Lambda en tant que proxy
Cas d'utilisation	AWS AppSync functions interact directly with API data sources.	AWS AppSync functions call Lambdas that interact with API data sources.
Runtime	*APPSYNC_JS* (JavaScript)	Tout environnement d'exécuti on Lambda pris en charge
Maximum size of code	32 000 caractères par fonction AWS AppSync	50 Mo (compressés, pour le téléchargement direct) par Lambda
External modules	Limité - Fonctionnalités prises en charge par APPSYNC_JS uniquement	Oui
Call any AWS service	Oui - Utilisation d'une source de données AWS AppSync HTTP	Oui - Utilisation du AWS SDK
Access to the request header	Oui	Oui
Network access	Non	Oui
File system access	Non	Oui
Logging and metrics	Oui	Oui
Build and test entirely within AppSync	Oui	Non
Cold start	Non	Non - Avec une simultanéité provisionnée

Auto-scaling	Oui, de manière transparente par AWS AppSync	Oui, tel que configuré dans Lambda
Pricing	Pas de frais supplémentaires	Facturé pour l'utilisation de Lambda

AWS AppSync les fonctions qui s'intègrent directement à leur source de données cible sont idéales pour les cas d'utilisation suivants :

- Interaction avec Amazon DynamoDB, Aurora Serverless et Service OpenSearch
- Interaction avec les API HTTP et transmission des en-têtes entrants
- Interaction avec AWS les services à l'aide de sources de données HTTP (avec signature AWS AppSync automatique des demandes avec le rôle de source de données fourni)
- Mise en œuvre du contrôle d'accès avant d'accéder aux sources de données
- Implémentation du filtrage des données récupérées avant de répondre à une demande
- Implémentation d'une orchestration simple avec exécution séquentielle de AWS AppSync fonctions dans un pipeline de résolution
- Contrôle des connexions de mise en cache et d'abonnement dans les requêtes et les mutations.

AWS AppSync les fonctions qui utilisent une source de données Lambda comme proxy sont idéales pour les cas d'utilisation suivants :

- Utilisation d'un langage autre que JavaScript le langage VTL (Velocity Template Language)
- Réglage et contrôle du processeur ou de la mémoire pour optimiser les performances
- Importation de bibliothèques tierces ou demande de fonctionnalités non prises en charge dans `APPSYNC_JS`
- Effectuer plusieurs requêtes réseau et/ou obtenir l'accès au système de fichiers pour répondre à une requête
- Requêtes par lots à l'aide de la configuration par lots.

Référence de l'objet contextuel du résolveur

AWS AppSyncdéfinit un ensemble de variables et de fonctions permettant de travailler avec les gestionnaires de demandes et de réponses. Cela facilite les opérations logiques sur les données avec GraphQL. Ce document décrit ces fonctions et fournit des exemples.

Accès à la variable **`context`**

Le`context`l'argument d'un gestionnaire de demandes et de réponses est un objet qui contient toutes les informations contextuelles pour l'invocation de votre résolveur. Elle présente la structure suivante :

```
type Context = {
  arguments: any;
  args: any;
  identity: Identity;
  source: any;
  error?: {
    message: string;
    type: string;
  };
  stash: any;
  result: any;
  prev: any;
  request: Request;
  info: Info;
};
```

> **Note**
>
> Vous constaterez souvent que`context`l'objet est appelé`ctx`.

Chaque champ du`context`l'objet est défini comme suit :

Champs de **`context`**

`arguments`

Carte qui contient tous les arguments GraphQL pour ce champ.

`identity`

Objet qui contient des informations sur l'appelant. Pour en savoir plus sur la structure de ce champ, consultez Identité.

`source`

Carte contenant la résolution du champ parent.

`stash`

Le stash est un objet mis à disposition dans chaque résolveur et gestionnaire de fonctions. Le même objet de réserve vit après une seule exécution du résolveur. Cela signifie que vous pouvez utiliser le stash pour transmettre des données arbitraires entre les gestionnaires de requêtes et de réponses et entre les fonctions d'un résolveur de pipeline.

> **Note**
>
> Vous ne pouvez pas supprimer ou remplacer l'intégralité de la réserve, mais vous pouvez ajouter, mettre à jour, supprimer et lire les propriétés de la réserve.

Vous pouvez ajouter des objets à la réserve en modifiant l'un des exemples de code ci-dessous :

```
//Example 1
ctx.stash.newItem = { key: "something" }

//Example 2
Object.assign(ctx.stash, {key1: value1, key2: value})
```

Vous pouvez supprimer des objets de la réserve en modifiant le code ci-dessous :

```
delete ctx.stash.key
```

`result`

Un conteneur pour les résultats de ce résolveur. Ce champ n'est disponible que pour les gestionnaires de réponses.

Par exemple, si vous résolvez le`author`champ de la requête suivante :

```
query {
```

```
    getPost(id: 1234) {
        postId
        title
        content
        author {
            id
            name
        }
    }
}
```

Ensuite, le completcontextune variable est disponible lorsqu'un gestionnaire de réponses est évalué :

```
{
  "arguments" : {
    id: "1234"
  },
  "source": {},
  "result" : {
      "postId": "1234",
      "title": "Some title",
      "content": "Some content",
      "author": {
        "id": "5678",
        "name": "Author Name"
      }
  },
  "identity" : {
      "sourceIp" : ["x.x.x.x"],
      "userArn" : "arn:aws:iam::123456789012:user/appsync",
      "accountId" : "666666666666",
      "user" : "AIDAAAAAAAAAAAAAAAAA"
  }
}
```

prev.result

Le résultat de toute opération précédente exécutée dans un résolveur de pipeline.

Si l'opération précédente était le gestionnaire de requêtes du résolveur de pipeline, alorsctx.prev.resultreprésente ce résultat d'évaluation et est mis à la disposition de la première fonction du pipeline.

Si l'opération précédente était la première fonction, alors`ctx.prev.result`représente le résultat de l'évaluation du premier gestionnaire de réponse de fonction et est mis à la disposition de la deuxième fonction du pipeline.

Si l'opération précédente était la dernière fonction, alors`ctx.prev.result`représente le résultat de l'évaluation de la dernière fonction et est mis à la disposition du gestionnaire de réponses du résolveur de pipeline.

`info`

Objet qui contient des informations sur la demande GraphQL. Pour obtenir la structure de ce champ, veuillez consulter Infos.

Identity

La section `identity` contient les informations sur l'appelant. La forme de cette section dépend du type d'autorisation de votre API AWS AppSync.

Pour plus d'informations surAWS AppSyncoptions de sécurité, voirAutorisation et authentification.

Autorisation **`API_KEY`**

`Leidentity`le champ n'est pas renseigné.

Autorisation **`AWS_LAMBDA`**

`Leidentity`a la forme suivante :

```
type AppSyncIdentityLambda = {
  resolverContext: any;
};
```

`Leidentity`contient le`resolverContext`clé, contenant la même`resolverContext`contenu renvoyé par la fonction Lambda autorisant la demande.

Autorisation **`AWS_IAM`**

`Leidentity`a la forme suivante :

```
type AppSyncIdentityIAM = {
  accountId: string;
  cognitoIdentityPoolId: string;
```

```
  cognitoIdentityId: string;
  sourceIp: string[];
  username: string;
  userArn: string;
  cognitoIdentityAuthType: string;
  cognitoIdentityAuthProvider: string;
};
```

Autorisation **AMAZON_COGNITO_USER_POOLS**

Le`identity`a la forme suivante :

```
type AppSyncIdentityCognito = {
  sourceIp: string[];
  username: string;
  groups: string[] | null;
  sub: string;
  issuer: string;
  claims: any;
  defaultAuthStrategy: string;
};
```

Chaque champ est défini comme suit :

`accountId`

LeAWSnuméro de compte de l'appelant.

`claims`

Demandes de l'utilisateur.

`cognitoIdentityAuthType`

Authentifié ou non authentifié en fonction du type d'identité.

`cognitoIdentityAuthProvider`

Liste séparée par des virgules des informations du fournisseur d'identité externe utilisées pour obtenir les informations d'identification utilisées pour signer la demande.

`cognitoIdentityId`

L'identifiant d'identité Amazon Cognito de l'appelant.

`cognitoIdentityPoolId`

L'ID du pool d'identités Amazon Cognito associé à l'appelant.

`defaultAuthStrategy`

Stratégie d'autorisation par défaut pour cet appelant (`ALLOW` ou `DENY`).

`issuer`

Émetteur du jeton.

`sourceIp`

L'adresse IP source de l'appelant quiAWS AppSyncreçoit. Si la demande n'inclut pas`x-forwarded-for`en-tête, la valeur IP source ne contient qu'une seule adresse IP provenant de la connexion TCP. Si la demande inclut un en-tête `x-forwarded-for`, l'adresse IP source est une liste d'adresses IP de l'en-tête `x-forwarded-for` en plus de l'adresse IP de la connexion TCP.

`sub`

UUID de l'utilisateur authentifié.

`user`

Utilisateur IAM.

`userArn`

Le nom de ressource Amazon (ARN) de l'utilisateur IAM.

`username`

Nom de l'utilisateur authentifié. En cas d'autorisation `AMAZON_COGNITO_USER_POOLS`, la valeur de username est la valeur de l'attribut cognito:username. Dans le cas de`AWS_IAM`autorisation, la valeur denom d'utilisateurest la valeur de`AWS`utilisateur principal. Si vous utilisez l'autorisation IAM avec des informations d'identification provenant de pools d'identités Amazon Cognito, nous vous recommandons d'utiliser`cognitoIdentityId`.

En-têtes de demande d'accès

AWS AppSyncpermet de transmettre des en-têtes personnalisés depuis les clients et d'y accéder dans vos résolveurs GraphQL en utilisant`ctx.request.headers`. Vous pouvez ensuite utiliser les valeurs d'en-tête pour des actions telles que l'insertion de données dans une source de données

ou les contrôles d'autorisation. Vous pouvez utiliser un ou plusieurs en-têtes de demande en utilisant`$curl`avec une clé d'API depuis la ligne de commande, comme illustré dans les exemples suivants :

Exemple d'en-tête unique

Supposons que vous définissiez un en-tête `custom` avec la valeur `nadia`, comme suit :

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:nadia" -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}"}' https://<ENDPOINT>/graphql
```

Il est alors possible d'y accéder avec `ctx.request.headers.custom`. Par exemple, cela peut se trouver dans le code suivant pour DynamoDB :

```
"custom": util.dynamodb.toDynamoDB(ctx.request.headers.custom)
```

Exemple d'en-tête multiple

Vous pouvez également transmettre plusieurs en-têtes dans une seule demande et y accéder dans le gestionnaire de résolution. Par exemple, si`custom`l'en-tête est défini avec deux valeurs :

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:bailey" -H "custom:nadia"
 -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo
\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}"}'
 https://<ENDPOINT>/graphql
```

Vous pouvez ensuite y accéder comme s'il s'agissait d'un tableau : par exemple, `ctx.request.headers.custom[1]`.

> **Note**
>
> AWS AppSyncn'expose pas l'en-tête du cookie dans`ctx.request.headers`.

Accédez au nom de domaine personnalisé de la demande

AWS AppSyncprend en charge la configuration d'un domaine personnalisé que vous pouvez utiliser pour accéder à votre GraphQL et aux points de terminaison en temps réel de vos API. Lorsque vous

faites une demande avec un nom de domaine personnalisé, vous pouvez obtenir le nom de domaine en utilisant`ctx.request.domainName`.

Lorsque vous utilisez le nom de domaine du point de terminaison GraphQL par défaut, la valeur est`null`.

Infos

La section `info` contient des informations sur la demande GraphQL. Cette section se présente sous la forme suivante :

```
type Info = {
  fieldName: string;
  parentTypeName: string;
  variables: any;
  selectionSetList: string[];
  selectionSetGraphQL: string;
};
```

Chaque champ est défini comme suit :

`fieldName`

Nom du champ en cours de résolution.

`parentTypeName`

Nom du type parent du champ en cours de résolution.

`variables`

Carte qui contient toutes les variables transmises dans la demande GraphQL.

`selectionSetList`

Représentation sous forme de liste des champs du jeu de sélection GraphQL. Les champs dotés d'un alias sont référencés uniquement par le nom de l'alias, et non par le nom du champ. L'exemple suivant détaille cela.

`selectionSetGraphQL`

Représentation sous forme de chaîne du jeu de sélection, formatée en langage SDL GraphQL. Bien que les fragments ne soient pas fusionnés dans le jeu de sélection, les fragments intégrés sont conservés, comme le montre l'exemple suivant.

> **Note**
>
> `JSON.stringify`n'inclura pas`selectionSetGraphQL`et`selectionSetList`dans la sérialisation des chaînes. Vous devez référencer ces propriétés directement.

Par exemple, si vous résolvez le champ `getPost` de la requête suivante :

```
query {
  getPost(id: $postId) {
    postId
    title
    secondTitle: title
    content
    author(id: $authorId) {
      authorId
      name
    }
    secondAuthor(id: "789") {
      authorId
    }
    ... on Post {
      inlineFrag: comments: {
        id
      }
    }
    ... postFrag
  }
}

fragment postFrag on Post {
  postFrag: comments: {
    id
  }
}
```

Ensuite, le complet`ctx.info`la variable disponible lors du traitement d'un gestionnaire peut être :

```
{
  "fieldName": "getPost",
  "parentTypeName": "Query",
  "variables": {
```

```
    "postId": "123",
    "authorId": "456"
  },
  "selectionSetList": [
    "postId",
    "title",
    "secondTitle"
    "content",
    "author",
    "author/authorId",
    "author/name",
    "secondAuthor",
    "secondAuthor/authorId",
    "inlineFragComments",
    "inlineFragComments/id",
    "postFragComments",
    "postFragComments/id"
  ],
  "selectionSetGraphQL": "{\n  getPost(id: $postId) {\n    postId\n    title\n
 secondTitle: title\n    content\n    author(id: $authorId) {\n      authorId\n
 name\n    }\n    secondAuthor(id: \"789\") {\n      authorId\n    }\n    ... on Post
 {\n      inlineFrag: comments {\n        id\n      }\n    }\n    ... postFrag\n  }\n}"
}
```

`selectionSetList`n'expose que les champs appartenant au type actuel. Si le type actuel est une interface ou une union, seuls les champs sélectionnés appartenant à l'interface sont exposés. Par exemple, selon le schéma suivant :

```
type Query {
    node(id: ID!): Node
}

interface Node {
    id: ID
}

type Post implements Node {
    id: ID
    title: String
    author: String
}

type Blog implements Node {
```

```
    id: ID
    title: String
    category: String
}
```

Et la requête suivante :

```
query {
    node(id: "post1") {
        id
        ... on Post {
            title
        }

        ... on Blog {
            title
        }
    }
}
```

Lorsque vous appelez`ctx.info.selectionSetListau`Query`.node`résolution de champ, uniquement`id`est exposé :

```
"selectionSetList": [
    "id"
]
```

JavaScript fonctionnalités d'exécution pour les résolveurs et les fonctions

L'environnement `APPSYNC_JS` d'exécution fournit des fonctionnalités similaires à celles de la version 6.0 d'ECMAScript (ES). Il prend en charge un sous-ensemble de ses fonctionnalités et fournit des méthodes supplémentaires (utilitaires) qui ne font pas partie des spécifications ES. Les rubriques suivantes répertorient toutes les fonctions de langages prises en charge.

> **Note**
>
> Actuellement, cette référence ne s'applique qu'à la version d'exécution 1.0.0.

Rubriques

Fonctionnalités d'exécution prises en charge

Les sections ci-dessous décrivent l'ensemble des fonctionnalités prises en charge par le moteur d'exécution APPSYNC_JS.

Fonctions de base

Les fonctionnalités principales suivantes sont prises en charge.

Types

Les types suivants sont pris en charge :

- nombres
- chaînes
- booléens
- objects
- tableaux
- functions

Opérateurs

Les opérateurs sont pris en charge, notamment :

- opérateurs mathématiques standard (`+`, `-`/,`%`,`*`,, etc.)
- opérateur de coalescence nul () `??`
- Chainage optionnel () `?.`
- opérateurs bit à bit
- `void`et `typeof` opérateurs

Les opérateurs suivants ne sont pas pris en charge :

- opérateurs unaires (`++`,`--`, et~)
- Opérateur `in`

> **Note**
>
> Utilisez l'`Object.has0wn`opérateur pour vérifier si la propriété spécifiée se trouve dans l'objet spécifié.

Déclarations

Les déclarations suivantes sont prises en charge :

- `const`
- `let`
- `var`
- `break`
- `else`
- `for-in`
- `for-of`
- `if`
- `return`
- `switch`
- syntaxe de propagation

Les éléments suivants ne sont pas pris en charge :

- `catch`
- `continue`
- `do-while`
- `finally`
- `for(initialization; condition; afterthought)`

> **Note**
>
> Les exceptions sont les `for-of` expressions `for-in` et les expressions, qui sont prises en charge.

- `throw`
- `try`
- `while`
- déclarations étiquetées

Littéraux

Les littéraux de modèle ES 6 suivants sont pris en charge :

- Cordes multilignes
- Interpolation d'expressions
- Modèles d'imbrication

Fonctions

La syntaxe de fonction suivante est prise en charge :

- Les déclarations de fonctions sont prises en charge.
- Les fonctions de flèche ES 6 sont prises en charge.
- La syntaxe des paramètres de repos ES 6 est prise en charge.

Mode strict

Les fonctions opèrent en mode strict par défaut. Vous n'avez donc pas besoin d'ajouter une instruction `use_strict` dans votre code de fonction. Elles ne peuvent pas être modifiées.

Objets primitifs

Les objets primitifs suivants d'ES et leurs fonctions sont pris en charge.

Objet

Les objets suivants sont pris en charge :

- `Object.assign()`
- `Object.entries()`
- `Object.hasOwn()`
- `Object.keys()`
- `Object.values()`
- `delete`

String

Les chaînes suivantes sont prises en charge :

- `String.prototype.length()`
- `String.prototype.charAt()`
- `String.prototype.concat()`
- `String.prototype.endsWith()`
- `String.prototype.indexOf()`
- `String.prototype.lastIndexOf()`
- `String.raw()`
- `String.prototype.replace()`

> **Note**
>
> Les expressions régulières ne sont pas prises en charge.

- `String.prototype.replaceAll()`

> **Note**
>
> Les expressions régulières ne sont pas prises en charge.

- `String.prototype.slice()`
- `String.prototype.split()`
- `String.prototype.startsWith()`
- `String.prototype.toLowerCase()`
- `String.prototype.toUpperCase()`
- `String.prototype.trim()`
- `String.prototype.trimEnd()`
- `String.prototype.trimStart()`

Nombre

Les numéros suivants sont pris en charge :

- `Number.isFinite`
- `Number.isNaN`

Objets et fonctions intégrés

Les fonctions et objets suivants sont pris en charge.

Math

Les fonctions mathématiques suivantes sont prises en charge :

- `Math.random()`
- `Math.min()`
- `Math.max()`
- `Math.round()`
- `Math.floor()`
- `Math.ceil()`

Tableau

Les méthodes matricielles suivantes sont prises en charge :

- `Array.prototype.length`
- `Array.prototype.concat()`
- `Array.prototype.fill()`
- `Array.prototype.flat()`
- `Array.prototype.indexOf()`
- `Array.prototype.join()`
- `Array.prototype.lastIndexOf()`
- `Array.prototype.pop()`
- `Array.prototype.push()`
- `Array.prototype.reverse()`
- `Array.prototype.shift()`
- `Array.prototype.slice()`
- `Array.prototype.sort()`

> **Note**
>
> `Array.prototype.sort()`ne supporte pas les arguments.

- `Array.prototype.splice()`
- `Array.prototype.unshift()`
- `Array.prototype.forEach()`
- `Array.prototype.map()`
- `Array.prototype.flatMap()`
- `Array.prototype.filter()`
- `Array.prototype.reduce()`
- `Array.prototype.reduceRight()`
- `Array.prototype.find()`
- `Array.prototype.some()`

- `Array.prototype.every()`
- `Array.prototype.findIndex()`
- `Array.prototype.findLast()`
- `Array.prototype.findLastIndex()`
- `delete`

Console

L'objet console est disponible pour le débogage. Lors de l'exécution d'une requête en direct, les instructions de journalisation ou d'erreur de la console sont envoyées à Amazon CloudWatch Logs (si la journalisation est activée). Lors de l'évaluation du code avec`evaluateCode`, les instructions de journal sont renvoyées dans la réponse à la commande.

- `console.error()`
- `console.log()`

JSON

Les méthodes JSON suivantes sont prises en charge :

- `JSON.parse()`

> **Note**
>
> Renvoie une chaîne vide si la chaîne analysée n'est pas un JSON valide.

- `JSON.stringify()`

Fonction

- Les `call` méthodes `applybind`, et ne sont pas prises en charge.
- Les constructeurs de fonctions ne sont pas pris en charge.
- La transmission d'une fonction en tant qu'argument n'est pas prise en charge.
- Les appels de fonction récursifs ne sont pas pris en charge.

Promesses

Les processus asynchrones ne sont pas pris en charge et les promesses ne sont pas prises en charge.

> **Note**
>
> L'accès au réseau et au système de fichiers n'est pas pris en charge au cours de l'`APPSYNC_JS`exécution dansAWS AppSync. AWS AppSyncgère toutes les opérations d'E/S en fonction des demandes faites par le AWS AppSync résolveur ou AWS AppSync la fonction.

Globals

Les constantes globales suivantes sont prises en charge :

- `NaN`
- `Infinity`
- `undefined`
- `util`
- `extensions`
- `runtime`

Types d'erreurs

Le renvoi d'erreurs avec n'`throw`est pas pris en charge. Vous pouvez renvoyer une erreur en utilisant `util.error()` la fonction. Vous pouvez inclure une erreur dans votre réponse GraphQL en utilisant la `util.appendError` fonction.

Pour plus d'informations, consultez la section Utils d'erreur.

Utilitaires intégrés

La `util` variable contient des méthodes utilitaires générales pour vous aider à travailler avec les données. Sauf indication contraire, tous les utilitaires emploient le jeu de caractères UTF-8.

Utilitaires d'encodage

Liste des utilitaires d'encodage

`util.urlEncode(String)`

Renvoie la chaîne en entrée sous la forme d'une chaîne codée `application/x-www-form-urlencoded`.

`util.urlDecode(String)`

Décode une chaîne codée `application/x-www-form-urlencoded` sous sa forme initiale non codée.

`util.base64Encode(string) : string`

Code les données d'entrée en une chaîne codée en base64.

`util.base64Decode(string) : string`

Décode les données d'une chaîne encodée en base64.

Utilitaires de génération d'identifiants

Liste des utilitaires de génération d'identifiants

`util.autoId()`

Renvoie un UUID 128 bits généré de façon aléatoire.

`util.autoUlid()`

Renvoie un ULID (identifiant lexicographiquement sortable universel unique) de 128 bits généré aléatoirement.

`util.autoKsuid()`

Renvoie un KSUID (K-Sortable Unique Identifier) de 128 bits généré aléatoirement en base62 et codé sous forme de chaîne d'une longueur de 27.

Utils d'erreur

Liste des utilitaires d'erreur

`util.error(String, String?, Object?, Object?)`

Lève une erreur personnalisée. Peut être utilisé dans les modèles de mappage de demande ou de réponse si le modèle détecte une erreur associée à la demande ou au résultat de l'appel. En outre, un `errorType` champ, un `data` champ et un `errorInfo` champ peuvent être spécifiés. La valeur `data` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL.

> **Note**
>
> `data`sera filtré en fonction de l'ensemble de sélection de requêtes. La valeur `errorInfo` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL.
> `errorInfo`ne sera pas filtré en fonction de l'ensemble de sélection de requêtes.

`util.appendError(String, String?, Object?, Object?)`

Ajoute une erreur personnalisée. Peut être utilisé dans les modèles de mappage de demande ou de réponse si le modèle détecte une erreur associée à la demande ou au résultat de l'appel. En outre, un `errorType` champ, un `data` champ et un `errorInfo` champ peuvent être spécifiés. Contrairement à `util.error(String, String?, Object?, Object?)`, l'évaluation du modèle n'est pas interrompue et, par conséquent, les données peuvent être retournées à l'appelant. La valeur `data` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL.

> **Note**
>
> `data`sera filtré en fonction de l'ensemble de sélection de requêtes. La valeur `errorInfo` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL.
> `errorInfo`ne sera pas filtré en fonction de l'ensemble de sélection de requêtes.

Utilitaires de correspondance de types et de modèles

Liste d'utilitaires correspondant au type et au modèle

`util.matches(String, String) : Boolean`

Renvoie la valeur true si le modèle spécifié dans le premier argument correspond aux données fournies dans le deuxième argument. Le modèle doit être une expression régulière, telle que `util.matches("a*b", "aaaaab")`. La fonctionnalité est basée sur Pattern, que vous pouvez référencer à titre de documentation ultérieure.

`util.authType()`

Renvoie une chaîne décrivant le type d'authentification multiple utilisé par une demande, renvoyant soit « IAM Authorization », « User Pool Authorization », « Open ID Connect Authorization », soit « API Key Authorization ».

Utilitaires de comportement des valeurs renvoyées

Liste des utilitaires relatifs au comportement des valeurs renvoyées

`util.escapeJavaScript(String)`

Renvoie la chaîne d'entrée sous forme de chaîne JavaScript échappée.

Utilitaires d'autorisation du résolveur

Liste des utilitaires d'autorisation du résolveur

`util.unauthorized()`

Lève `Unauthorized` pour le champ en cours de résolution. Utilisez-le dans les modèles de mappage de demandes ou de réponses pour déterminer s'il convient d'autoriser l'appelant à résoudre le champ.

Modules intégrés

Les modules font partie de l'`APPSYNC_JS`environnement d'exécution et fournissent des utilitaires pour aider à écrire des JavaScript résolveurs et des fonctions.

Fonctions du module DynamoDB

Les fonctions du module DynamoDB offrent une expérience améliorée lors de l'interaction avec les sources de données DynamoDB. Vous pouvez envoyer des requêtes à vos sources de données DynamoDB à l'aide des fonctions et sans ajouter de mappage de type.

Les modules sont importés à l'aide de `@aws-appsync/utils/dynamodb` :

```
// Modules are imported using @aws-appsync/utils/dynamodb
import * as ddb from '@aws-appsync/utils/dynamodb';
```

Fonctions

Liste des fonctions

`get<T>(payload: GetInput): DynamoDBGetItemRequest`

> **Tip**
>
> Voir the section called “Inputs” pour plus d'informations sur GetInput.

Génère un `DynamoDBGetItemRequest` objet pour envoyer une GetItemrequête à DynamoDB.

```
import { get } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 return get({ key: { id: ctx.args.id } });
}
```

`put<T>(payload): DynamoDBPutItemRequest`

Génère un `DynamoDBPutItemRequest` objet pour envoyer une PutItemrequête à DynamoDB.

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 return ddb.put({ key: { id: util.autoId() }, item: ctx.args });
}
```

`remove<T>(payload): DynamoDBDeleteItemRequest`

Génère un `DynamoDBDeleteItemRequest` objet pour envoyer une DeleteItemrequête à DynamoDB.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 return ddb.remove({ key: { id: ctx.args.id } });
}
```

`scan<T>(payload): DynamoDBScanRequest`

Génère un `DynamoDBScanRequest` pour envoyer une demande de scan à DynamoDB.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const { limit = 10, nextToken } = ctx.args;
 return ddb.scan({ limit, nextToken });
}
```

`sync<T>(payload): DynamoDBSyncRequest`

Génère un `DynamoDBSyncRequest` objet pour effectuer une demande de synchronisation. La demande ne reçoit que les données modifiées depuis la dernière requête (mises à jour delta). Les demandes peuvent uniquement être adressées à des sources de données DynamoDB versionnées.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const { limit = 10, nextToken, lastSync } = ctx.args;
 return ddb.sync({ limit, nextToken, lastSync });
}
```

`update<T>(payload): DynamoDBUpdateItemRequest`

Génère un `DynamoDBUpdateItemRequest` objet pour envoyer une UpdateItemrequête à DynamoDB.

Opérations

Les aides aux opérations vous permettent d'effectuer des actions spécifiques sur certaines parties de vos données lors des mises à jour. Pour commencer, importez `operations` depuis `@aws-appsync/utils/dynamodb` :

```
// Modules are imported using operations
import {operations} from '@aws-appsync/utils/dynamodb';
```

Liste des opérations

`add<T>(payload)`

Fonction d'assistance qui ajoute un nouvel élément d'attribut lors de la mise à jour de DynamoDB.

Exemple

Pour ajouter une adresse (rue, ville et code postal) à un élément DynamoDB existant à l'aide de la valeur ID :

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const updateObj = {
  address: operations.add({
   street1: '123 Main St',
   city: 'New York',
   zip: '10001',
  }),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

`append <T>(payload)`

Fonction d'assistance qui ajoute une charge utile à la liste existante dans DynamoDB.

Exemple

Pour ajouter les nouveaux identifiants d'amis (`newFriendIds`) à une liste d'amis existante (`friendsIds`) lors d'une mise à jour :

```
import { update, operations } from '@aws-appsync/utils/dynamodb';
```

```
export function request(ctx) {
 const newFriendIds = [101, 104, 111];
 const updateObj = {
  friendsIds: operations.append(newFriendIds),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

`decrement (by?)`

Fonction d'assistance qui décrémente la valeur d'attribut existante dans l'élément lors de la mise à jour de DynamoDB.

Exemple

Pour réduire le compteur (`friendsCount`) d'un ami de 10 :

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const updateObj = {
  friendsCount: operations.decrement(10),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

`increment (by?)`

Fonction d'assistance qui incrémente la valeur d'attribut existante dans l'élément lors de la mise à jour de DynamoDB.

Exemple

Pour augmenter de 10 le compteur d'amis (`friendsCount`) :

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const updateObj = {
  friendsCount: operations.increment(10),
 };
 return update({ key: { id: 1 }, update: updateObj });
```

```
}
```

`prepend <T>(payload)`

Fonction d'assistance qui s'ajoute à la liste existante dans DynamoDB.

Exemple

Pour ajouter les nouveaux identifiants d'amis (`newFriendIds`) à une liste d'amis existante (`friendsIds`) lors d'une mise à jour :

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const newFriendIds = [101, 104, 111];
 const updateObj = {
  friendsIds: operations.prepend(newFriendIds),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

`replace <T>(payload)`

Fonction d'assistance qui remplace un attribut existant lors de la mise à jour d'un élément dans DynamoDB. Cela est utile lorsque vous souhaitez mettre à jour l'intégralité de l'objet ou du sous-objet de l'attribut et pas uniquement les clés de la charge utile.

Exemple

Pour remplacer une adresse (rue, ville et code postal) dans un `info` objet :

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const updateObj = {
  info: {
   address: operations.replace({
    street1: '123 Main St',
    city: 'New York',
    zip: '10001',
   }),
  },
 };
```

```
 return update({ key: { id: 1 }, update: updateObj });
}
```

`updateListItem <T>(payload, index)`

Fonction d'assistance qui remplace un élément d'une liste.

Exemple

Dans le cadre de la mise à jour (`newFriendIds`), cet exemple a `updateListItem` été utilisé pour mettre à jour les valeurs d'ID du deuxième élément (index :1, new ID :102) et du troisième élément (index :2, new ID :112) dans une liste (`friendsIds`).

```
import { update, operations as ops } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const newFriendIds = [
  ops.updateListItem('102', 1), ops.updateListItem('112', 2)
 ];
 const updateObj = { friendsIds: newFriendIds };
 return update({ key: { id: 1 }, update: updateObj });
}
```

Inputs

Liste des entrées

`Type GetInput<T>`

```
GetInput<T>: {
    consistentRead?: boolean;
    key: DynamoDBKey<T>;
}
```

Déclaration de type

- `consistentRead?: boolean` (facultatif)

  Un booléen facultatif qui indique si vous souhaitez effectuer une lecture très cohérente avec DynamoDB.

- `key: DynamoDBKey<T>` (obligatoire)

Paramètre obligatoire qui spécifie la clé de l'élément dans DynamoDB. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou des clés de hachage et de tri.

`Type PutInput<T>`

```
PutInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T> | null;
    customPartitionKey?: string;
    item: Partial<T>;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
}
```

Déclaration de type

- `_version?: number` (facultatif)
- `condition?: DynamoDBFilterObject<T> | null` (facultatif)

  Lorsque vous placez un objet dans une table DynamoDB, vous pouvez éventuellement spécifier une expression conditionnelle qui détermine si la demande doit aboutir ou non en fonction de l'état de l'objet déjà présent dans DynamoDB avant l'exécution de l'opération.

- `customPartitionKey?: string` (facultatif)

  Lorsqu'elle est activée, cette valeur de chaîne modifie le format des `ds_pk` enregistrements `ds_sk` et utilisés par la table de synchronisation delta lorsque le versionnement est activé. Lorsque cette option est activée, le traitement de l'`populateIndexFields`entrée est également activé.

- `item: Partial<T>` (obligatoire)

  Le reste des attributs de l'élément à placer dans DynamoDB.

- `key: DynamoDBKey<T>` (obligatoire)

  Paramètre obligatoire qui spécifie la clé de l'élément dans DynamoDB sur lequel le placement sera effectué. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou des clés de hachage et de tri.

- `populateIndexFields?: boolean` (facultatif)

Valeur booléenne qui, lorsqu'elle est activée en même temps que le`customPartitionKey`, crée de nouvelles entrées pour chaque enregistrement de la table de synchronisation delta, en particulier dans les colonnes `gsi_ds_pk` et`gsi_ds_sk`. Pour plus d'informations, consultez la section Détection et synchronisation des conflits dans le manuel du AWS AppSync développeur.

`Type QueryInput<T>`

```
QueryInput<T>: ScanInput<T> & {
    query: DynamoDBKeyCondition<Required<T>>;
}
```

Déclaration de type

- `query: DynamoDBKeyCondition<Required<T>>` (obligatoire)

  Spécifie une condition clé qui décrit les éléments à interroger. Pour un index donné, la condition d'une clé de partition doit être une égalité et la clé de tri une comparaison ou un `beginsWith` (lorsqu'il s'agit d'une chaîne). Seuls les types de nombres et de chaînes sont pris en charge pour les clés de partition et de tri.

  Exemple

  Prenez le `User` type ci-dessous :

  ```
  type User = {
    id: string;
    name: string;
    age: number;
    isVerified: boolean;
    friendsIds: string[]
  }
  ```

  La requête ne peut inclure que les champs suivants : `id`name, et `age` :

  ```
  const query: QueryInput<User> = {
      name: { eq: 'John' },
      age: { gt: 20 },
  }
  ```

Type RemoveInput<T>

```
RemoveInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T>;
    customPartitionKey?: string;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
}
```

Déclaration de type

- `_version?: number` (facultatif)
- `condition?: DynamoDBFilterObject<T>` (facultatif)

  Lorsque vous supprimez un objet dans DynamoDB, vous pouvez éventuellement spécifier une expression conditionnelle qui détermine si la demande doit aboutir ou non en fonction de l'état de l'objet déjà présent dans DynamoDB avant l'exécution de l'opération.

  Exemple

  L'exemple suivant est une `DeleteItem` expression contenant une condition qui permet à l'opération de réussir uniquement si le propriétaire du document correspond à l'utilisateur qui fait la demande.

```
type Task = {
  id: string;
  title: string;
  description: string;
  owner: string;
  isComplete: boolean;
}
const condition: DynamoDBFilterObject<Task> = {
  owner: { eq: 'XXXXXXXXXXXXXXXX' },
}

remove<Task>({
   key: {
     id: 'XXXXXXXXXXXXXXXX',
  },
  condition,
});
```

- `customPartitionKey?: string` (facultatif)

  Lorsqu'elle est activée, la `customPartitionKey` valeur modifie le format des `ds_pk` enregistrements `ds_sk` et utilisés par la table de synchronisation delta lorsque le versionnement est activé. Lorsque cette option est activée, le traitement de l'`populateIndexFields`entrée est également activé.

- `key: DynamoDBKey<T>` (obligatoire)

  Paramètre obligatoire qui spécifie la clé de l'élément en cours de suppression dans DynamoDB. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou des clés de hachage et de tri.

  Exemple

  Si a `User` uniquement la clé de hachage d'un utilisateur`id`, la clé ressemblera à ceci :

  ```
  type User = {
   id: number
   name: string
   age: number
   isVerified: boolean
  }
  const key: DynamoDBKey<User> = {
   id: 1,
  }
  ```

  Si l'utilisateur de la table possède une clé de hachage (`id`) et une clé de tri (`name`), la clé ressemblera à ceci :

  ```
  type User = {
   id: number
   name: string
   age: number
   isVerified: boolean
   friendsIds: string[]
  }

  const key: DynamoDBKey<User> = {
   id: 1,
   name: 'XXXXXXXXXX',
  ```

```
}
```

- `populateIndexFields?: boolean` (facultatif)

  Valeur booléenne qui, lorsqu'elle est activée en même temps que le`customPartitionKey`, crée de nouvelles entrées pour chaque enregistrement de la table de synchronisation delta, en particulier dans les colonnes `gsi_ds_pk` et`gsi_ds_sk`.

`Type ScanInput<T>`

```
ScanInput<T>: {
    consistentRead?: boolean | null;
    filter?: DynamoDBFilterObject<T> | null;
    index?: string | null;
    limit?: number | null;
    nextToken?: string | null;
    scanIndexForward?: boolean | null;
    segment?: number;
    select?: DynamoDBSelectAttributes;
    totalSegments?: number;
}
```

Déclaration de type

- `consistentRead?: boolean | null` (facultatif)

  Un booléen facultatif pour indiquer des lectures cohérentes lors de l'interrogation de DynamoDB. La valeur par défaut est `false`.

- `filter?: DynamoDBFilterObject<T> | null` (facultatif)

  Filtre facultatif à appliquer aux résultats après les avoir extraits du tableau.

- `index?: string | null` (facultatif)

  Nom facultatif de l'index à analyser.

- `limit?: number | null` (facultatif)

  Nombre maximal facultatif de résultats à renvoyer.

- `nextToken?: string | null` (facultatif)

  Un jeton de pagination facultatif pour poursuivre une requête précédente. Il a été obtenu à partir d'une requête précédente.

- `scanIndexForward?: boolean | null` (facultatif)

  Un booléen facultatif indiquant si la requête est exécutée par ordre croissant ou décroissant. Par défaut, cette valeur indique `true`.

- `segment?: number` (facultatif)
- `select?: DynamoDBSelectAttributes` (facultatif)

  Attributs à renvoyer depuis DynamoDB. Par défaut, le résolveur AWS AppSync DynamoDB renvoie uniquement les attributs projetés dans l'index. Les valeurs prises en charge sont :

  - `ALL_ATTRIBUTES`

    Renvoie tous les attributs d'élément de la table ou de l'index spécifié. Si vous interrogez un index secondaire local, DynamoDB extrait l'élément entier de la table parent pour chaque élément correspondant de l'index. Si l'index est configuré de façon à projeter tous les attributs de l'élément, toutes les données peuvent être obtenues à partir de l'index secondaire local, et aucune extraction n'est nécessaire.

  - `ALL_PROJECTED_ATTRIBUTES`

    Renvoie tous les attributs qui ont été projetés dans l'index. Si l'index est configuré de façon à projeter tous les attributs, la valeur renvoyée revient à spécifier `ALL_ATTRIBUTES`.

  - `SPECIFIC_ATTRIBUTES`

    Renvoie uniquement les attributs répertoriés dans`ProjectionExpression`. Cette valeur de retour revient à spécifier `ProjectionExpression` sans spécifier de valeur pour`AttributesToGet`.

- `totalSegments?: number` (facultatif)

`Type DynamoDBSyncInput<T>`

```
DynamoDBSyncInput<T>: {
    basePartitionKey?: string;
    deltaIndexName?: string;
    filter?: DynamoDBFilterObject<T> | null;
    lastSync?: number;
    limit?: number | null;
    nextToken?: string | null;
}
```

Déclaration de type

- `basePartitionKey?: string` (facultatif)

  La clé de partition de la table de base à utiliser lors d'une opération de synchronisation. Ce champ permet d'effectuer une opération de synchronisation lorsque la table utilise une clé de partition personnalisée.

- `deltaIndexName?: string` (facultatif)

  Index utilisé pour l'opération de synchronisation. Cet index est nécessaire pour activer une opération de synchronisation sur l'ensemble de la table delta store lorsque la table utilise une clé de partition personnalisée. L'opération de synchronisation sera effectuée sur le GSI (créé sur `gsi_ds_pk` et`gsi_ds_sk`).

- `filter?: DynamoDBFilterObject<T> | null` (facultatif)

  Filtre facultatif à appliquer aux résultats après les avoir extraits du tableau.

- `lastSync?: number` (facultatif)

  Moment, en millisecondes d'époque, auquel la dernière opération de synchronisation réussie a commencé. Si spécifié, seuls les éléments qui ont changé après `lastSync` sont retournés. Ce champ ne doit être rempli qu'après avoir récupéré toutes les pages d'une opération de synchronisation initiale. En cas d'omission, les résultats de la table de base seront renvoyés. Dans le cas contraire, les résultats de la table delta seront renvoyés.

- `limit?: number | null` (facultatif)

  Nombre maximal facultatif d'éléments à évaluer simultanément. Si cette option est omise, la limite par défaut sera définie sur `100` éléments. La valeur maximale de ce champ est `1000` éléments.

- `nextToken?: string | null` (facultatif)

`Type DynamoDBUpdateInput<T>`

```
DynamoDBUpdateInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T>;
    customPartitionKey?: string;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
    update: DynamoDBUpdateObject<T>;
}
```

Déclaration de type

- `_version?: number` (facultatif)
- `condition?: DynamoDBFilterObject<T>` (facultatif)

  Lorsque vous mettez à jour un objet dans DynamoDB, vous pouvez éventuellement spécifier une expression conditionnelle qui détermine si la demande doit aboutir ou non en fonction de l'état de l'objet déjà présent dans DynamoDB avant l'exécution de l'opération.

- `customPartitionKey?: string` (facultatif)

  Lorsqu'elle est activée, la `customPartitionKey` valeur modifie le format des `ds_pk` enregistrements `ds_sk` et utilisés par la table de synchronisation delta lorsque le versionnement est activé. Lorsque cette option est activée, le traitement de l'`populateIndexFields`entrée est également activé.

- `key: DynamoDBKey<T>` (obligatoire)

  Paramètre obligatoire qui spécifie la clé de l'élément en cours de mise à jour dans DynamoDB. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou des clés de hachage et de tri.

- `populateIndexFields?: boolean` (facultatif)

  Valeur booléenne qui, lorsqu'elle est activée en même temps que le`customPartitionKey`, crée de nouvelles entrées pour chaque enregistrement de la table de synchronisation delta, en particulier dans les colonnes `gsi_ds_pk` et`gsi_ds_sk`.

- `update: DynamoDBUpdateObject<T>`

  Objet qui spécifie les attributs à mettre à jour ainsi que leurs nouvelles valeurs. L'objet de mise à jour peut être utilisé avec `add` `removereplace`,`increment`,,`decrement`,`append`,`prepend`,`updateListItem`.

Fonctions du module Amazon RDS

Les fonctions du module Amazon RDS offrent une expérience améliorée lors de l'interaction avec les bases de données configurées avec l'API Amazon RDS Data. Le module est importé à l'aide de `@aws-appsync/utils/rds` :

```
import * as rds from '@aws-appsync/utils/rds';
```

Les fonctions peuvent également être importées individuellement. Par exemple, l'importation ci-dessous utilise `sql` :

```
import { sql } from '@aws-appsync/utils/rds';
```

Fonctions

Vous pouvez utiliser les aides utilitaires du module AWS AppSync RDS pour interagir avec votre base de données.

Select

L'`select`utilitaire crée une `SELECT` instruction pour interroger votre base de données relationnelle.

Usage de base

Dans sa forme de base, vous pouvez spécifier la table que vous souhaitez interroger :

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // "SELECT * FROM "persons"
    return createPgStatement(select({table: 'persons'}));
}
```

Notez que vous pouvez également spécifier le schéma dans l'identifiant de votre table :

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // SELECT * FROM "private"."persons"
    return createPgStatement(select({table: 'private.persons'}));
}
```

Spécification des colonnes

Vous pouvez définir des colonnes à l'aide de `columns` cette propriété. Si aucune valeur n'est définie, la valeur par défaut est : *

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name']
    }));
}
```

Vous pouvez également spécifier le tableau d'une colonne :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "persons"."name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'persons.name']
    }));
}
```

Limites et compensations

Vous pouvez appliquer `limit` et répondre `offset` à la requête :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // LIMIT :limit
    // OFFSET :offset
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        limit: 10,
        offset: 40
    }));
}
```

Commander par

Vous pouvez trier vos résultats à l'aide de `orderBy` cette propriété. Fournissez un tableau d'objets spécifiant la colonne et une `dir` propriété facultative :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name" FROM "persons"
    // ORDER BY "name", "id" DESC
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        orderBy: [{column: 'name'}, {column: 'id', dir: 'DESC'}]
    }));
}
```

Filtres

Vous pouvez créer des filtres à l'aide de l'objet de condition spéciale :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}}
    }));
}
```

Vous pouvez également combiner des filtres :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME and "id" > :ID
```

```
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}, id: {gt: 10}}
    }));
}
```

Vous pouvez également créer des `OR` déclarations :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME OR "id" > :ID
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: { or: [
            { name: { eq: 'Stephane'} },
            { id: { gt: 10 } }
        ]}
    }));
}
```

Vous pouvez également annuler une condition avec `not` :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE NOT ("name" = :NAME AND "id" > :ID)
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: { not: [
            { name: { eq: 'Stephane'} },
            { id: { gt: 10 } }
        ]}
    }));
}
```

Vous pouvez également utiliser les opérateurs suivants pour comparer des valeurs :

Opérateur	Description	Types de valeurs possibles
eq	Equal	number, string, boolean
ne	Not equal	number, string, boolean
le	Less than or equal	number, string
lt	Less than	number, string
ge	Greater than or equal	number, string
gt	Greater than	number, string
contains	Like	string
notContains	Not like	string
beginsWith	Starts with prefix	string
between	Between two values	number, string
attributeExists	The attribute is not null	number, string, boolean
size	checks the length of the element	string

Insert

L'`insert`utilitaire fournit un moyen simple d'insérer des éléments d'une seule ligne dans votre base de données avec l'`INSERT`opération.

Insertions d'un seul article

Pour insérer un élément, spécifiez le tableau, puis transmettez votre objet de valeurs. Les clés d'objet sont mappées aux colonnes de votre tableau. Les noms des colonnes sont automatiquement ignorés et les valeurs sont envoyées à la base de données à l'aide de la variable map :

```
import { insert, createMySQLStatement } from '@aws-appsync/utils/rds';
```

```
export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({ table: 'persons', values });

    // Generates statement:
    // INSERT INTO `persons`(`name`)
    // VALUES(:NAME)
    return createMySQLStatement(insertStatement)
}
```

Cas d'utilisation de MySQL

Vous pouvez combiner un `insert` suivi d'un `select` pour récupérer la ligne que vous avez insérée :

```
import { insert, select, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({  table: 'persons', values });
    const selectStatement = select({
        table: 'persons',
        columns: '*',
        where: { id: { eq: values.id } },
        limit: 1,
    });

    // Generates statement:
    // INSERT INTO `persons`(`name`)
    // VALUES(:NAME)
    // and
    // SELECT *
    // FROM `persons`
    // WHERE `id` = :ID
    return createMySQLStatement(insertStatement, selectStatement)
}
```

Cas d'utilisation de Postgres

Avec Postgres, vous pouvez l'utiliser `returning`pour obtenir des données à partir de la ligne que vous avez insérée. Il accepte * un tableau de noms de colonnes :

```
import { insert, createPgStatement } from '@aws-appsync/utils/rds';
```

```
export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({
        table: 'persons',
        values,
        returning: '*'
    });

    // Generates statement:
    // INSERT INTO "persons"("name")
    // VALUES(:NAME)
    // RETURNING *
    return createPgStatement(insertStatement)
}
```

Mettre à jour

L'`update`utilitaire vous permet de mettre à jour les lignes existantes. Vous pouvez utiliser l'objet condition pour appliquer des modifications aux colonnes spécifiées dans toutes les lignes qui répondent à la condition. Supposons, par exemple, que nous ayons un schéma qui nous permet de réaliser cette mutation. Nous voulons mettre à jour les `name` de `Person` avec la `id` valeur de`3`, mais uniquement si nous les connaissons (`known_since`) depuis l'année `2000` :

```
mutation Update {
    updatePerson(
        input: {id: 3, name: "Jon"},
        condition: {known_since: {ge: "2000"}}
    ) {
    id
    name
  }
}
```

Notre résolveur de mises à jour ressemble à ceci :

```
import { update, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: { id, ...values }, condition } = ctx.args;
    const where = {
        ...condition,
```

```
        id: { eq: id },
    };
    const updateStatement = update({
        table: 'persons',
        values,
        where,
        returning: ['id', 'name'],
    });

    // Generates statement:
    // UPDATE "persons"
    // SET "name" = :NAME, "birthday" = :BDAY, "country" = :COUNTRY
    // WHERE "id" = :ID
    // RETURNING "id", "name"
    return createPgStatement(updateStatement)
}
```

Nous pouvons ajouter une vérification à notre condition pour nous assurer que seule la ligne dont la clé primaire est `id` égale à `3` est mise à jour. De même, pour Postgres`inserts`, vous pouvez utiliser `returning` pour renvoyer les données modifiées.

Remove (suppression)

L'`remove`utilitaire vous permet de supprimer des lignes existantes. Vous pouvez utiliser l'objet de condition sur toutes les lignes qui répondent à la condition. Notez qu'il `delete` s'agit d'un mot clé réservé dans JavaScript. `remove`doit être utilisé à la place :

```
import { remove, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: { id }, condition } = ctx.args;
    const where = { ...condition, id: { eq: id } };
    const deleteStatement = remove({
        table: 'persons',
        where,
        returning: ['id', 'name'],
    });

    // Generates statement:
    // DELETE "persons"
    // WHERE "id" = :ID
    // RETURNING "id", "name"
    return createPgStatement(updateStatement)
```

```
}
```

Forçage de type

Dans certains cas, vous souhaiterez peut-être plus de précisions quant au type d'objet approprié à utiliser dans votre déclaration. Vous pouvez utiliser les indications de type fournies pour spécifier le type de vos paramètres. AWS AppSyncprend en charge les mêmes indications de type que l'API Data. Vous pouvez convertir vos paramètres en utilisant les `typeHint` fonctions du AWS AppSync `rds` module.

L'exemple suivant vous permet d'envoyer un tableau sous forme de valeur qui est convertie en objet JSON. Nous utilisons l'`->`opérateur pour récupérer l'élément situé `index` 2 dans le tableau JSON :

```
import { sql, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/
rds';

export function request(ctx) {
    const arr = ctx.args.list_of_ids
    const statement = sql`select ${typeHint.JSON(arr)}->2 as value`
    return createPgStatement(statement)
}

export function response(ctx) {
    return toJsonObject(ctx.result)[0][0].value
}
```

Le casting est également utile lors de la manipulation et de `DATE` la comparaison`TIME`, et `TIMESTAMP` :

```
import { select, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const when = ctx.args.when
    const statement = select({
        table: 'persons',
        where: { createdAt : { gt: typeHint.DATETIME(when) } }
    })
    return createPgStatement(statement)
}
```

Voici un autre exemple montrant comment vous pouvez envoyer la date et l'heure actuelles :

```
import { sql, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const now = util.time.nowFormatted('YYYY-MM-dd HH:mm:ss')
    return createPgStatement(sql`select ${typeHint.TIMESTAMP(now)}`)
}
```

Indications de type disponibles

- `typeHint.DATE`- Le paramètre correspondant est envoyé sous forme d'objet de `DATE` ce type à la base de données. Le format accepté est `YYYY-MM-DD`.
- `typeHint.DECIMAL`- Le paramètre correspondant est envoyé sous forme d'objet de `DECIMAL` ce type à la base de données.
- `typeHint.JSON`- Le paramètre correspondant est envoyé sous forme d'objet de `JSON` ce type à la base de données.
- `typeHint.TIME`- La valeur de paramètre de chaîne correspondante est envoyée sous forme d'objet de `TIME` ce type à la base de données. Le format accepté est `HH:MM:SS[.FFF]`.
- `typeHint.TIMESTAMP`- La valeur de paramètre de chaîne correspondante est envoyée sous forme d'objet de `TIMESTAMP` ce type à la base de données. Le format accepté est `YYYY-MM-DD HH:MM:SS[.FFF]`.
- `typeHint.UUID`- La valeur de paramètre de chaîne correspondante est envoyée sous forme d'objet de `UUID` ce type à la base de données.

utilitaires d'exécution

La `runtime` bibliothèque fournit des utilitaires permettant de contrôler ou de modifier les propriétés d'exécution de vos résolveurs et fonctions.

Liste des utilitaires d'exécution

`runtime.earlyReturn(obj?: unknown): never`

L'invocation de cette fonction interrompt l'exécution de la AWS AppSync fonction ou du résolveur en cours (Unit ou Pipeline Resolver) en fonction du contexte actuel. L'objet spécifié est renvoyé en tant que résultat.

- Lorsqu'il est appelé dans un gestionnaire de demande de AWS AppSync fonction, la source de données et le gestionnaire de réponse sont ignorés, et le gestionnaire de demande de fonction

suivant (ou le gestionnaire de réponse du résolveur de pipeline s'il s'agit de la dernière fonction) est appelé. AWS AppSync

- Lorsqu'il est appelé dans un gestionnaire de demandes de résolution de AWS AppSync pipeline, l'exécution du pipeline est ignorée et le gestionnaire de réponse du résolveur de pipeline est appelé immédiatement.

Exemple

```
import { runtime } from '@aws-appsync/utils'

export function request(ctx) {
  runtime.earlyReturn({ hello: 'world' })
  // code below is not executed
  return ctx.args
}

// never called because request returned early
export function response(ctx) {
  return ctx.result
}
```

Des aides temporelles dans util.time

La variable `util.time` contient les méthodes datetime pour aider à générer les horodatages, à convertir d'un format datetime à l'autre et à analyser les chaînes datetime. La syntaxe des formats datetime est basée sur DateTimeFormatterlaquelle vous pouvez vous référer pour obtenir de la documentation supplémentaire. Nous fournissons quelques exemples ci-dessous, ainsi qu'une liste des méthodes et des descriptions disponibles.

Temps et utilités

Liste des heures et des outils

`util.time.nowISO8601()`

Renvoie une représentation String (chaîne) d'UTC au format ISO8601.

`util.time.nowEpochSeconds()`

Renvoie le nombre de secondes entre l'époque Unix 1970-01-01T00:00:00Z et maintenant.

`util.time.nowEpochMilliSeconds()`

Renvoie le nombre de millisecondes entre l'époque Unix 1970-01-01T00:00:00Z et maintenant.

`util.time.nowFormatted(String)`

Renvoie une chaîne de l'horodatage actuel (UTC) à l'aide du format spécifié à partir d'un type d'entrée String (chaîne).

`util.time.nowFormatted(String, String)`

Renvoie une chaîne de l'horodatage actuel pour un fuseau horaire à l'aide du format et du fuseau spécifiés à partir de types d'entrée String (chaîne).

`util.time.parseFormattedToEpochMilliSeconds(String, String)`

Analyse un horodatage transmis sous forme de chaîne avec un format, puis renvoie l'horodatage en millisecondes depuis l'époque.

`util.time.parseFormattedToEpochMilliSeconds(String, String, String)`

Analyse un horodatage transmis sous forme de chaîne avec un format et un fuseau horaire, puis renvoie l'horodatage en millisecondes depuis l'époque.

`util.time.parseISO8601ToEpochMilliSeconds(String)`

Analyse un horodatage ISO8601 transmis sous forme de chaîne, puis renvoie l'horodatage en millisecondes depuis l'époque.

`util.time.epochMilliSecondsToSeconds(long)`

Convertit une époque Unix en millisecondes en une époque Unix en secondes.

`util.time.epochMilliSecondsToISO8601(long)`

Convertit un horodatage en millisecondes d'époque en un horodatage ISO8601.

`util.time.epochMilliSecondsToFormatted(long, String)`

Convertit l'horodatage d'une époque en millisecondes, transmis sous sa forme la plus longue, en un horodatage formaté selon le format fourni en UTC.

`util.time.epochMilliSecondsToFormatted(long, String, String)`

Convertit un horodatage en millisecondes d'époque, transmis sous forme de long, en un horodatage formaté selon le format fourni dans le fuseau horaire fourni.

Assistants DynamoDB dans util.dynamodb

`util.dynamodb`contient des méthodes d'assistance qui facilitent l'écriture et la lecture de données dans Amazon DynamoDB, telles que le mappage automatique des types et le formatage.

vers DynamoDB

Liste des utilitaires de ToDynamoDB

`util.dynamodb.toDynamoDB(Object)`

Outil général de conversion d'objets pour DynamoDB qui convertit les objets d'entrée en une représentation DynamoDB appropriée. La façon dont il représente certains types est clairement arrêtée : par exemple, il utilise les listes (« L ») plutôt que les ensembles (« SS », « NS », « BS »). Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

Exemple de chaîne

```
Input:      util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

Exemple de numéro

```
Input:      util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

Exemple booléen

```
Input:      util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

Exemple de liste

```
Input:      util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
```

```
                    }
                }
            ]
        }
```

Exemple de carte

```
Input:      util.dynamodb.toDynamoDB({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
                           { "S" : "boop" }
                       ]
                   }
               }
           }
```

Utilitaires ToString

Liste des utilitaires ToString

`util.dynamodb.toString(String)`

Convertit une chaîne d'entrée au format de chaîne DynamoDB. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

`util.dynamodb.toStringSet(List<String>)`

Convertit une liste contenant des chaînes au format de jeu de chaînes DynamoDB. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

Utils ToNumber

Liste des utilitaires ToNumber

`util.dynamodb.toNumber(Number)`

Convertit un nombre au format numérique DynamoDB. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

`util.dynamodb.toNumberSet(List<Number>)`

Convertit une liste de nombres au format d'ensemble de numéros DynamoDB. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

Utilitaires ToBinary

Liste des utilitaires ToBinary

`util.dynamodb.toBinary(String)`

Convertit les données binaires codées sous forme de chaîne base64 au format binaire DynamoDB. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

`util.dynamodb.toBinarySet(List<String>)`

Convertit une liste de données binaires codées sous forme de chaînes base64 au format d'ensemble binaire DynamoDB. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

Utilitaires ToBoolean

Liste des utilitaires ToBoolean

`util.dynamodb.toBoolean(Boolean)`

Convertit un booléen au format booléen DynamoDB approprié. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

Utilitaires ToNull

Liste des utilitaires ToNull

`util.dynamodb.toNull()`

Renvoie une valeur nulle au format DynamoDB nul. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      util.dynamodb.toNull()
Output:     { "NULL" : null }
```

Utilitaires ToList

Liste des utilitaires ToList

`util.dynamodb.toList(List)`

Convertit une liste d'objets au format de liste DynamoDB. Chaque élément de la liste est également converti au format DynamoDB approprié. La façon dont il représente certains objets imbriqués est clairement arrêtée : par exemple, il utilise les listes (« L ») plutôt que les ensembles (« SS », « NS », « BS »). Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
```

```
                { "N" : 123 },
                {
                    "M" : {
                        "bar" : { "S" : "baz" }
                    }
                }
            ]
        }
```

Utilitaires TomaP

Liste des utilitaires de TomaP

`util.dynamodb.toMap(Map)`

Convertit une carte au format de carte DynamoDB. Chaque valeur de la carte est également convertie dans le format DynamoDB approprié. La façon dont il représente certains objets imbriqués est clairement arrêtée : par exemple, il utilise les listes (« L ») plutôt que les ensembles (« SS », « NS », « BS »). Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      util.dynamodb.toMap({ "foo": "bar", "baz" : 1234, "beep": [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
                           { "S" : "boop" }
                       ]
                   }
               }
           }
```

`util.dynamodb.toMapValues(Map)`

Crée une copie de la carte où chaque valeur a été convertie au format DynamoDB approprié. La façon dont il représente certains objets imbriqués est clairement arrêtée : par exemple, il utilise les listes (« L ») plutôt que les ensembles (« SS », « NS », « BS »).

```
Input:      util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
```

```
Output:     {
               "foo"  : { "S" : "bar" },
               "baz"  : { "N" : 1234 },
               "beep" : {
                   "L" : [
                       { "S" : "boop" }
                   ]
               }
           }
```

> **Note**
>
> Cela est légèrement différent `util.dynamodb.toMap(Map)` car il renvoie uniquement le contenu de la valeur d'attribut DynamoDB, mais pas la valeur d'attribut complète elle-même. Par exemple, les instructions suivantes sont exactement identiques :
>
> ```
> util.dynamodb.toMapValues(<map>)
> util.dynamodb.toMap(<map>)("M")
> ```

Utilitaires de l'objet S3

Liste des utilitaires de S3Object

`util.dynamodb.toS3Object(String key, String bucket, String region)`

Convertit la clé, le compartiment et la région en représentation de l'objet DynamoDB S3. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      util.dynamodb.toS3Object("foo", "bar", region = "baz")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo", \"bucket\" : \"bar", \"region
\" : \"baz" } }" }
```

`util.dynamodb.toS3Object(String key, String bucket, String region, String version)`

Convertit la clé, le compartiment, la région et la version facultative en représentation de l'objet DynamoDB S3. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      util.dynamodb.toS3Object("foo", "bar", "baz", "beep")
```

```
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region
\" : \"baz\", \"version\" = \"beep\" } }" }
```

`util.dynamodb.fromS3ObjectJson(String)`

Accepte la valeur de chaîne d'un objet DynamoDB S3 et renvoie une carte contenant la clé, le compartiment, la région et la version facultative.

```
Input:      util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
 \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :
 "beep" }
```

Assistants HTTP dans util.http

L'`util.http`utilitaire fournit des méthodes d'assistance que vous pouvez utiliser pour gérer les paramètres des requêtes HTTP et pour ajouter des en-têtes de réponse.

liste d'utilitaires util.http

`util.http.copyHeaders(headers)`

Copie l'en-tête de la carte sans l'ensemble restreint d'en-têtes HTTP. Vous pouvez l'utiliser pour transférer les en-têtes de requête vers votre point de terminaison HTTP en aval.

`util.http.addResponseHeader(String, Object)`

Ajoute un seul en-tête personnalisé avec le nom (`String`) et la valeur (`Object`) de la réponse. Les limites suivantes s'appliquent :

- Les noms d'en-tête ne peuvent correspondre à aucun des AWS AppSync en-têtes existants AWS ou restreints.
- Les noms d'en-tête ne peuvent pas commencer par des préfixes restreints, tels que `x-amzn-` ou `x-amz-`.
- La taille des en-têtes de réponse personnalisés ne peut pas dépasser 4 Ko. Cela inclut les noms et les valeurs des en-têtes.
- Vous devez définir chaque en-tête de réponse une fois par opération GraphQL. Toutefois, si vous définissez plusieurs fois un en-tête personnalisé portant le même nom, la définition la plus récente apparaît dans la réponse. Tous les en-têtes sont pris en compte dans la limite de taille d'en-tête, quel que soit leur nom.

`util.http.addResponseHeaders(Map)`

Ajoute plusieurs en-têtes de réponse à la réponse à partir de la carte de noms (`String`) et de valeurs (`Object`) spécifiée. Les mêmes limites répertoriées pour la `addResponseHeader(String, Object)` méthode s'appliquent également à cette méthode.

Aides à la transformation dans util.transform

`util.transform`contient des méthodes d'assistance qui facilitent l'exécution d'opérations complexes sur des sources de données.

Liste des outils d'aide à la transformation

`util.transform.toDynamoDBFilterExpression(filterObject: DynamoDBFilterObject) : string`

Convertit une chaîne d'entrée en une expression de filtre à utiliser avec DynamoDB. Nous vous recommandons de l'utiliser `toDynamoDBFilterExpression` avec les fonctions du module intégré.

`util.transform.toElasticsearchQueryDSL(object: OpenSearchQueryObject) : string`

Convertit l'entrée donnée en son expression OpenSearch Query DSL équivalente, en la renvoyant sous forme de chaîne JSON.

Exemple de saisie :

```
util.transform.toElasticsearchQueryDSL({
    "upvotes":{
        "ne":15,
        "range":[
            10,
            20
        ]
    },
    "title":{
        "eq":"hihihi",
        "wildcard":"h*i"
    }
  })
```

Exemple de sortie :

```
{
    "bool":{
      "must":[
          {
            "bool":{
              "must":[
                  {
                    "bool":{
                      "must_not":{
                        "term":{
                          "upvotes":15
                        }
                      }
                    }
                  },
                  {
                    "range":{
                      "upvotes":{
                        "gte":10,
                        "lte":20
                      }
                    }
                  }
              ]
            }
          },
          {
            "bool":{
              "must":[
                  {
                    "term":{
                      "title":"hihihi"
                    }
                  },
                  {
                  "wildcard":{
                      "title":"h*i"
                    }
                  }
              ]
            }
          }
```

```
      ]
    }
}
```

> **Note**
>
> L'opérateur par défaut est supposé être AND.

`util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter`

Convertit un objet `Map` d'entrée en objet d'`SubscriptionFilter`expression. La `util.transform.toSubscriptionFilter` méthode est utilisée comme entrée dans l'`extensions.setSubscriptionFilter()`extension. Pour plus d'informations, consultez la section Extensions.

> **Note**
>
> Les paramètres et l'instruction de retour sont répertoriés ci-dessous :
> Paramètres
>
> - `objFilter`: `SubscriptionFilterObject`
>
>   Un objet `Map` d'entrée converti en objet d'`SubscriptionFilter`expression.
> - `ignoredFields`: `SubscriptionFilterExcludeKeysType` (facultatif)
>
>   A `List` des noms de champs du premier objet qui seront ignorés.
> - `rules`: `SubscriptionFilterRuleObject` (facultatif)
>
>   Un objet `Map` d'entrée avec des règles strictes qui est inclus lors de la construction de l'objet `SubscriptionFilter` d'expression. Ces règles strictes seront incluses dans l'objet `SubscriptionFilter` d'expression afin qu'au moins l'une des règles soit satisfaite pour passer le filtre d'abonnement.
>
> Réponse
> Retourne un `SubscriptionFilter`.

`util.transform.toSubscriptionFilter(Map, List)`

Convertit un objet `Map` d'entrée en objet d'`SubscriptionFilter`expression. La `util.transform.toSubscriptionFilter` méthode est utilisée comme entrée dans l'`extensions.setSubscriptionFilter()`extension. Pour plus d'informations, consultez la section Extensions.

Le premier argument est l'objet `Map` d'entrée converti en objet d'`SubscriptionFilter`expression. Le deuxième argument est un nom `List` de champ qui est ignoré dans le premier objet `Map` d'entrée lors de la construction de l'objet `SubscriptionFilter` d'expression.

`util.transform.toSubscriptionFilter(Map, List, Map)`

Convertit un objet `Map` d'entrée en objet d'`SubscriptionFilter`expression. La `util.transform.toSubscriptionFilter` méthode est utilisée comme entrée dans l'`extensions.setSubscriptionFilter()`extension. Pour plus d'informations, consultez la section Extensions.

`util.transform.toDynamoDBConditionExpression(conditionObject)`

Crée une expression de condition DynamoDB.

Arguments du filtre d'abonnement

Le tableau suivant explique comment les arguments des utilitaires suivants sont définis :

- `Util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter`

Argument 1: Map

L'argument 1 est un `Map` objet dont les valeurs clés sont les suivantes :

- noms de champs
- « et »
- « ou »

Pour les noms de champs sous forme de clés, les conditions relatives aux entrées de ces champs sont sous la forme de`"operator" : "value"`.

L'exemple suivant montre comment des entrées peuvent être ajoutées au `Map` :

```
"field_name" : {
                    "operator1" : value
               }

## We can have multiple conditions for the same field_name:

"field_name" : {
                    "operator1" : value
                    "operator2" : value
                    .
                    .
                    .
               }
```

Lorsqu'un champ comporte au moins deux conditions, toutes ces conditions sont considérées comme utilisant l'opération OR.

L'entrée `Map` peut également comporter des touches « et » et « ou », ce qui implique que toutes les entrées qu'elles contiennent doivent être jointes en utilisant la logique AND ou OR en fonction de la clé. Les valeurs clés « et » et « ou » supposent un ensemble de conditions.

```
"and" : [

            {
                "field_name1" : {
                    "operator1" : value
                }
             },

             {
                "field_name2" : {
                    "operator1" : value
                }
             },
             .
             .
        ].
```

Notez que vous pouvez imbriquer « et » et « ou ». C'est-à-dire que vous pouvez avoir imbriqué « et » /"ou » dans un autre bloc « et » /"ou ». Toutefois, cela ne fonctionne pas pour les champs simples.

```
"and" : [

            {
                "field_name1" : {
                    "operator" : value
                }
             },

             {
                "or" : [
                            {
                                "field_name2" : {
                                    "operator" : value
                                }
                            },

                            {
                                "field_name3" : {
                                    "operator" : value
                                }
                            }

                        ].
```

L'exemple suivant montre une entrée de l'argument 1 utilisant`util.transform.toSubscriptionFilter(Map) : Map`.

Entrée (s)

Argument 1 : Carte :

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
```

```
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Sortie

Le résultat est un `Map` objet :

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
```

```
          "value": 2000
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Admin"
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "gte",
          "value": 20
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
```

```
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Admin"
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "gte",
          "value": 20
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 2: List

L'argument 2 contient des noms `List` de champs qui ne doivent pas être pris en compte dans l'entrée `Map` (argument 1) lors de la construction de l'objet `SubscriptionFilter` d'expression. Ils `List` peuvent également être vides.

L'exemple suivant montre les entrées de l'argument 1 et de l'argument 2 en utilisant`util.transform.toSubscriptionFilter(Map, List) : Map`.

Entrée (s)

Argument 1 : Carte :

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argument 2 : Liste :

```
["percentageUp", "author"]
```

Sortie

Le résultat est un `Map` objet :

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 3: Map

L'argument 3 est un `Map` objet dont les noms de champs sont des valeurs clés (il ne peut pas y avoir « et » ou « ou »). Pour les noms de champs sous forme de clés, les conditions relatives à ces champs sont des entrées sous la forme de`"operator" : "value"`. Contrairement à l'argument 1, l'argument 3 ne peut pas avoir plusieurs conditions dans la même clé. De plus, l'argument 3 ne contient pas de clause « et » ou « ou », il n'y a donc aucune imbrication non plus.

L'argument 3 représente une liste de règles strictes, qui sont ajoutées à l'objet `SubscriptionFilter` d'expression afin qu'au moins l'une de ces conditions soit remplie pour passer le filtre.

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.
```

L'exemple suivant montre les entrées de l'argument 1, de l'argument 2 et de l'argument 3 en utilisant`util.transform.toSubscriptionFilter(Map, List, Map) : Map`.

Entrée (s)

Argument 1 : Carte :

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
```

```
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argument 2 : Liste :

```
["percentageUp", "author"]
```

Argument 3 : Carte :

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
}
```

Sortie

Le résultat est un `Map` objet :

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
```

```
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        },
        {
          "fieldName": "upvotes",
          "operator": "gte",
          "value": 250
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Person1"
        }
      ]
    }
  ]
}
```

assistants de chaîne dans util.str

`util.str`contient des méthodes pour faciliter les opérations courantes sur les chaînes de caractères.

liste des utilitaires de util.str

`util.str.normalize(String, String)`

Normalise une chaîne en utilisant l'une des quatre formes de normalisation Unicode : NFC, NFD, NFKC ou NFKD. Le premier argument est la chaîne à normaliser. Le deuxième argument est « nfc », « nfd », « nfkc » ou « nfkd », spécifiant le type de normalisation à utiliser pour le processus de normalisation.

Extensions

`extensions`contient un ensemble de méthodes permettant d'effectuer des actions supplémentaires dans vos résolveurs.

Extensions de mise en cache

`extensions.evictFromApiCache(typeName: string, fieldName: string, keyValuePair: Record<string, string>) : Object`

Élimine un élément du cache AWS AppSync côté serveur. Le premier argument est le nom du type. Le deuxième argument est le nom du champ. Le troisième argument est un objet contenant des éléments de paire clé-valeur qui spécifient la valeur de la clé de mise en cache. Vous devez placer les éléments dans l'objet dans le même ordre que les clés de mise en cache dans le résolveur mis en cache. `cachingKey` Pour plus d'informations sur la mise en cache, consultez la section Comportement de mise en cache.

Exemple 1 :

Cet exemple évacue les éléments mis en cache pour un résolveur appelé `Query.allClasses` sur lequel une clé de mise en cache appelée a été utilisée. `context.arguments.semester` Lorsque la mutation est appelée et que le résolveur s'exécute, si une entrée est correctement effacée, la réponse contient une `apiCacheEntriesDeleted` valeur dans l'objet extensions qui indique le nombre d'entrées supprimées.

```
import { util, extensions } from '@aws-appsync/utils';
```

```
export const request = (ctx) => ({ payload: null });

export function response(ctx) {
 extensions.evictFromApiCache('Query', 'allClasses', {
  'context.arguments.semester': ctx.args.semester,
 });
 return null;
}
```

> **Note**
>
> Cette fonction ne fonctionne que pour les mutations, pas pour les requêtes.

Extensions d'abonnement

`extensions.setSubscriptionFilter(filterJsonObject)`

Définit des filtres d'abonnement améliorés. Chaque événement de notification d'abonnement est évalué par rapport aux filtres d'abonnement fournis et envoie des notifications aux clients si tous les filtres répondent aux critèrestrue. L'argument est `filterJsonObject` (Vous trouverez plus d'informations sur cet argument ci-dessous dans la filterJsonObject section Argument :). Voir Filtrage amélioré des abonnements.

> **Note**
>
> Vous pouvez utiliser cette fonction d'extension uniquement dans le gestionnaire de réponses d'un résolveur d'abonnement. Nous vous recommandons également de l'utiliser `util.transform.toSubscriptionFilter` pour créer votre filtre.

`extensions.setSubscriptionInvalidationFilter(filterJsonObject)`

Définit les filtres d'invalidation des abonnements. Les filtres d'abonnement sont évalués par rapport à la charge utile d'invalidation, puis invalident un abonnement donné s'ils sont évalués à. `true` L'argument est `filterJsonObject` (Vous trouverez plus d'informations sur cet argument ci-dessous dans la filterJsonObject section Argument :). Voir Filtrage amélioré des abonnements.

> **Note**
>
> Vous pouvez utiliser cette fonction d'extension uniquement dans le gestionnaire de réponses d'un résolveur d'abonnement. Nous vous recommandons également de l'utiliser `util.transform.toSubscriptionFilter` pour créer votre filtre.

`extensions.invalidateSubscriptions(invalidationJsonObject)`

Utilisé pour initier l'invalidation d'un abonnement suite à une mutation. L'argument est `invalidationJsonObject` (Vous trouverez plus d'informations sur cet argument ci-dessous dans la invalidationJsonObject section Argument :).

> **Note**
>
> Cette extension ne peut être utilisée que dans les modèles de mappage des réponses des résolveurs de mutations.
> Vous ne pouvez utiliser qu'au maximum cinq appels de `extensions.invalidateSubscriptions()` méthode uniques dans une seule demande. Si vous dépassez cette limite, vous recevrez une erreur GraphQL.

Argument : filterJsonObject

L'objet JSON définit des filtres d'abonnement ou d'invalidation. Il s'agit d'un ensemble de filtres dans un`filterGroup`. Chaque filtre est un ensemble de filtres individuels.

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
```

```
            "filters" : [
                 {
                     "fieldName" : "group",
                     "operator" : "in",
                     "value" : ["Admin", "Developer"]
                 }
            ]

        }
    ]
}
```

Chaque filtre possède trois attributs :

- `fieldName`— Le champ du schéma GraphQL.
- `operator`— Le type d'opérateur.
- `value`— Les valeurs à comparer à la `fieldName` valeur de la notification d'abonnement.

Voici un exemple d'attribution de ces attributs :

```
{
 "fieldName" : "severity",
 "operator" : "le",
 "value" : context.result.severity
}
```

Argument : invalidationJsonObject

`invalidationJsonObject`définit les éléments suivants :

- `subscriptionField`— L'abonnement au schéma GraphQL à invalider. Un seul abonnement, défini comme une chaîne dans le`subscriptionField`, est considéré comme invalide.
- `payload`— Une liste de paires clé-valeur utilisée comme entrée pour invalider les abonnements si le filtre d'invalidation est évalué par rapport à leurs valeurs. `true`

  L'exemple suivant invalide les clients abonnés et connectés utilisant l'on`UserDelete`abonnement lorsque le filtre d'invalidation défini dans le résolveur d'abonnement est évalué par rapport à la valeur. `true` `payload`

  ```
  export const request = (ctx) => ({ payload: null });
  ```

```
export function response(ctx) {
 extensions.invalidateSubscriptions({
  subscriptionField: 'onUserDelete',
  payload: { group: 'Developer', type: 'Full-Time' },
 });
 return ctx.result;
}
```

Assistants XML dans le fichier util.xml

`util.xml`contient des méthodes pour faciliter la conversion de chaînes XML.

Liste des utilitaires du fichier util.xml

`util.xml.toMap(String) : Object`

Convertit une chaîne XML en dictionnaire.

Exemple 1 :

```
Input:

<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
    <id>1</id>
    <title>Getting started with GraphQL</title>
</post>
</posts>

Output (object):

{
    "posts":{
      "post":{
        "id":1,
        "title":"Getting started with GraphQL"
      }
    }
}
```

Exemple 2 :

```
Input:

<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
<post>
  <id>2</id>
  <title>Getting started with AppSync</title>
</post>
</posts>

Output (JavaScript object):

{
    "posts":{
    "post":[
        {
            "id":1,
            "title":"Getting started with GraphQL"
        },
        {
            "id":2,
            "title":"Getting started with AppSync"
        }
    ]
    }
}
```

`util.xml.toJsonString(String, Boolean?) : String`

Convertit une chaîne XML en chaîne JSON. Ceci est similaire à`toMap`, sauf que la sortie est une chaîne. Cela est utile si vous souhaitez convertir et renvoyer directement la réponse XML à partir d'un objet HTTP vers JSON. Vous pouvez définir un paramètre booléen facultatif pour déterminer si vous souhaitez coder le JSON en chaîne.

JavaScriptréférence de fonction de résolution pour DynamoDB

LeAWS AppSyncLa fonction DynamoDB vous permet d'utiliserGraphQLpour stocker et récupérer des données dans les tables Amazon DynamoDB existantes de votre compte. Ce résolveur fonctionne en vous permettant de mapper une requête GraphQL entrante en un appel DynamoDB, puis de mapper la réponse DynamoDB à GraphQL. Cette section décrit les gestionnaires de demandes et de réponses pour les opérations DynamoDB prises en charge.

GetItem

Le`GetItem`la demande vous permet de direAWS AppSyncFonction DynamoDB pour créer un`GetItem`demande à DynamoDB, et vous permet de spécifier :

- La clé de l'élément dans DynamoDB
- S'il convient d'utiliser une lecture cohérente ou non

Le`GetItem`La demande a la structure suivante :

```
type DynamoDBGetItem = {
  operation: 'GetItem';
  key: { [key: string]: any };
  consistentRead?: ConsistentRead;
  projection?: {
    expression: string;
    expressionNames?: { [key: string]: string };
  };
};
```

Les champs sont définis comme suit :

Champs de GetItem

GetItemliste des champs

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `GetItem`, ce champ doit être défini sur `GetItem`. Cette valeur est obligatoire.

`key`

La clé de l'élément dans DynamoDB. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Cette valeur est obligatoire.

`consistentRead`

S'il faut ou non effectuer une lecture très cohérente avec DynamoDB. Ce champ est facultatif et `false` est la valeur définie par défaut.

`projection`

Projection utilisée pour spécifier les attributs à renvoyer par l'opération DynamoDB. Pour plus d'informations sur les projections, voirProjections. Ce champ est facultatif.

L'élément renvoyé par DynamoDB est automatiquement converti en types primitifs GraphQL et JSON, et est disponible dans le résultat du contexte (`context.result`).

Pour plus d'informations sur la conversion de type DynamoDB, voirSystème de types (mappage des réponses).

Pour plus d'informations surJavaScriptrésolveurs, voirJavaScriptvue d'ensemble des résolveurs.

Exemple

L'exemple suivant est un gestionnaire de demande de fonction pour une requête GraphQL`getThing(foo: String!, bar: String!)`:

```
export function request(ctx) {
  const {foo, bar} = ctx.args
  return {
    operation : "GetItem",
    key : util.dynamodb.toMapValues({foo, bar}),
    consistentRead : true
  }
}
```

Pour de plus amples informations sur l'API de DynamoDB `GetItem`, veuillez consulter la Documentation sur les API de DynamoDB.

PutItem

Le`PutItem`le document de mappage des demandes vous permet d'indiquerAWS AppSyncFonction DynamoDB pour créer un`PutItem`demande à DynamoDB, et vous permet de spécifier les éléments suivants :

- La clé de l'élément dans DynamoDB
- Contenu complet de l'élément (composé de `key` et de `attributeValues`)
- Conditions de réussite de l'opération

Le`PutItem`La demande a la structure suivante :

```
type DynamoDBPutItemRequest = {
  operation: 'PutItem';
  key: { [key: string]: any };
  attributeValues: { [key: string]: any};
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

Les champs sont définis comme suit :

Champs de PutItem

PutItemliste des champs

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `PutItem`, ce champ doit être défini sur `PutItem`. Cette valeur est obligatoire.

`key`

La clé de l'élément dans DynamoDB. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Cette valeur est obligatoire.

`attributeValues`

Le reste des attributs de l'élément doit être placé dans DynamoDB. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Ce champ est facultatif.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Si aucune condition n'est spécifiée, la requête `PutItem` remplace toute entrée existante pour cet élément. Pour plus d'informations sur les conditions, voirExpressions de conditions. Cette valeur est facultative.

`_version`

Valeur numérique représentant la dernière version connue d'un élément. Cette valeur est facultative. Ce champ est utilisé pour la détection de conflits et n'est pris en charge que sur les sources de données versionnées.

`customPartitionKey`

Lorsqu'elle est activée, cette valeur de chaîne modifie le format du`ds_sk`et`ds_pk`enregistrements utilisés par la table de synchronisation delta lorsque le versionnement a été activé (pour plus d'informations, voirDétection et synchronisation des conflitsdans leAWS AppSyncGuide du développeur). Lorsque cette option est activée, le traitement du`populateIndexFields`la saisie est également activée. Ce champ est facultatif.

`populateIndexFields`

Une valeur booléenne qui, lorsqu'elle est activéeainsi que le**`customPartitionKey`**, crée de nouvelles entrées pour chaque enregistrement de la table de synchronisation delta, en particulier dans`gsi_ds_pk`et`gsi_ds_sk`colonnes. Pour plus d'informations, voirDétection et synchronisation des conflitsdans leAWS AppSyncGuide du développeur. Ce champ est facultatif.

L'élément écrit dans DynamoDB est automatiquement converti en types primitifs GraphQL et JSON et est disponible dans le résultat du contexte (`context.result`).

L'élément écrit dans DynamoDB est automatiquement converti en types primitifs GraphQL et JSON et est disponible dans le résultat du contexte (`context.result`).

Pour plus d'informations sur la conversion de type DynamoDB, voirSystème de types (mappage des réponses).

Pour plus d'informations surJavaScriptrésolveurs, voirJavaScriptvue d'ensemble des résolveurs.

Exemple 1

L'exemple suivant est un gestionnaire de demande de fonction pour une mutation GraphQLupdateThing(`foo: String!, bar: String!, name: String!, version: Int!`).

S'il n'existe aucun élément avec la clé spécifiée, il est créé. S'il existe déjà un élément avec la clé spécifiée, il est remplacé.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, ...values} = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}
```

Exemple 2

L'exemple suivant est un gestionnaire de demande de fonction pour une mutation GraphQLupdateThing(`foo: String!, bar: String!, name: String!, expectedVersion: Int!`).

Cet exemple vérifie que l'élément actuellement dans DynamoDB possède le`version`champ défini sur`expectedVersion`.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, name, expectedVersion } = ctx.args;
  const values = { name, version: expectedVersion + 1 };
  let condition = util.transform.toDynamoDBConditionExpression({
    version: { eq: expectedVersion },
  });

  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ foo, bar }),
    attributeValues: util.dynamodb.toMapValues(values),
    condition,
```

```
  };
}
```

Pour de plus amples informations sur l'API de DynamoDB `PutItem`, veuillez consulter la Documentation sur les API de DynamoDB.

UpdateItem

Le`UpdateItem`cette demande vous permet de dire àAWS AppSyncFonction DynamoDB pour créer un`UpdateItem`demande à DynamoDB et vous permet de spécifier les éléments suivants :

- La clé de l'élément dans DynamoDB
- Expression de mise à jour décrivant comment mettre à jour l'élément dans DynamoDB
- Conditions de réussite de l'opération

Le`UpdateItem`La demande a la structure suivante :

```
type DynamoDBUpdateItemRequest = {
  operation: 'UpdateItem';
  key: { [key: string]: any };
  update: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

Les champs sont définis comme suit :

Champs de UpdateItem

UpdateItemliste des champs

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `UpdateItem`, ce champ doit être défini sur `UpdateItem`. Cette valeur est obligatoire.

`key`

La clé de l'élément dans DynamoDB. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la spécification d'une « valeur saisie », voirSystème de types (mappage des demandes). Cette valeur est obligatoire.

`update`

Leupdatecette section vous permet de spécifier une expression de mise à jour qui décrit comment mettre à jour l'élément dans DynamoDB. Pour plus d'informations sur la façon d'écrire des expressions de mise à jour, consultez leDynamoDBUpdateExpressionsdocumentation. Cette section est obligatoire.

La section `update` possède trois composants :

`expression`

Expression de mise à jour. Cette valeur est obligatoire.

`expressionNames`

Substituts des espaces réservés de nom des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé au nom utilisé dans`expression`, et la valeur doit être une chaîne correspondant au nom d'attribut de l'élément dans DynamoDB. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de nom des attributs de l'expression utilisés dans l'`expression`.

`expressionValues`

Substituts des espaces réservés de valeur des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé de valeur utilisé dans l'`expression`, et la valeur doit être typée. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Cela doit être spécifié. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de valeur des attributs de l'expression utilisés dans l'`expression`.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Si aucune condition n'est spécifiée, la requête `UpdateItem` met à jour l'entrée existante quel que soit son état actuel. Pour plus d'informations sur les conditions, voirExpressions de conditions. Cette valeur est facultative.

`_version`

Valeur numérique représentant la dernière version connue d'un élément. Cette valeur est facultative. Ce champ est utilisé pour la détection de conflits et n'est pris en charge que sur les sources de données versionnées.

`customPartitionKey`

Lorsqu'elle est activée, cette valeur de chaîne modifie le format du`ds_sk`et`ds_pk`enregistrements utilisés par la table de synchronisation delta lorsque le versionnement a été activé (pour plus d'informations, voirDétection et synchronisation des conflitsdans leAWS AppSyncGuide du développeur). Lorsque cette option est activée, le traitement du`populateIndexFields`la saisie est également activée. Ce champ est facultatif.

`populateIndexFields`

Une valeur booléenne qui, lorsqu'elle est activéeainsi que le**`customPartitionKey`**, crée de nouvelles entrées pour chaque enregistrement de la table de synchronisation delta, en particulier dans`gsi_ds_pk`et`gsi_ds_sk`colonnes. Pour plus d'informations, voirDétection et synchronisation des conflitsdans leAWS AppSyncGuide du développeur. Ce champ est facultatif.

L'élément mis à jour dans DynamoDB est automatiquement converti en types primitifs GraphQL et JSON et est disponible dans le résultat du contexte (`context.result`).

Pour plus d'informations sur la conversion de type DynamoDB, voirSystème de types (mappage des réponses).

Pour plus d'informations surJavaScriptrésolveurs, voirJavaScriptvue d'ensemble des résolveurs.

Exemple 1

L'exemple suivant est un gestionnaire de demande de fonction pour la mutation GraphQL`upvote(id: ID!)`.

Dans cet exemple, un élément de DynamoDB possède son`upvotes`et`version`champs incrémentés de 1.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { id } = ctx.args;
  return {
    operation: 'UpdateItem',
```

```
    key: util.dynamodb.toMapValues({ id }),
    update: {
      expression: 'ADD #votefield :plusOne, version :plusOne',
      expressionNames: { '#votefield': 'upvotes' },
      expressionValues: { ':plusOne': { N: 1 } },
    },
  };
}
```

Exemple 2

L'exemple suivant est un gestionnaire de demande de fonction pour une mutation GraphQL`updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)`.

Il s'agit d'un exemple complexe qui inspecte les arguments et génère dynamiquement l'expression de mise à jour incluant uniquement les arguments qui ont été fournis par le client. Par exemple, si `title` et `author` sont omis, ils ne sont pas mis à jour. Si un argument est spécifié mais que sa valeur est`null`, puis ce champ est supprimé de l'objet dans DynamoDB. Enfin, l'opération comporte une condition, qui vérifie si l'élément actuellement dans DynamoDB possède`version`champ défini sur`expectedVersion`:

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { args: { input: { id, ...values } } } = ctx;

  const condition = {
    id: { attributeExists: true },
    version: { eq: values.expectedVersion },
  };
  values.expectedVersion += 1;
  return dynamodbUpdateRequest({ keys: { id }, values, condition });
}


/**
 * Helper function to update an item
 * @returns an UpdateItem request
 */
function dynamodbUpdateRequest(params) {
  const { keys, values, condition: inCondObj } = params;
```

```
  const sets = [];
  const removes = [];
  const expressionNames = {};
  const expValues = {};

  // Iterate through the keys of the values
  for (const [key, value] of Object.entries(values)) {
    expressionNames[`#${key}`] = key;
    if (value) {
      sets.push(`#${key} = :${key}`);
      expValues[`:${key}`] = value;
    } else {
      removes.push(`#${key}`);
    }
  }

  let expression = sets.length ? `SET ${sets.join(', ')}` : '';
  expression += removes.length ? ` REMOVE ${removes.join(', ')}` : '';

  const condition = JSON.parse(
    util.transform.toDynamoDBConditionExpression(inCondObj)
  );

  return {
    operation: 'UpdateItem',
    key: util.dynamodb.toMapValues(keys),
    condition,
    update: {
      expression,
      expressionNames,
      expressionValues: util.dynamodb.toMapValues(expValues),
    },
  };
}
```

Pour de plus amples informations sur l'API de DynamoDB `UpdateItem`, veuillez consulter la Documentation sur les API de DynamoDB.

DeleteItem

Le`DeleteItem`la demande vous permet de direAWS AppSyncFonction DynamoDB pour créer un`DeleteItem`demande à DynamoDB, et vous permet de spécifier les éléments suivants :

- La clé de l'élément dans DynamoDB
- Conditions de réussite de l'opération

LeDeleteItemLa demande a la structure suivante :

```
type DynamoDBDeleteItemRequest = {
  operation: 'DeleteItem';
  key: { [key: string]: any };
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

Les champs sont définis comme suit :

Champs de DeleteItem

DeleteItemliste des champs

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `DeleteItem`, ce champ doit être défini sur `DeleteItem`. Cette valeur est obligatoire.

`key`

La clé de l'élément dans DynamoDB. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la spécification d'une « valeur saisie », voirSystème de types (mappage des demandes). Cette valeur est obligatoire.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Si aucune condition n'est spécifiée, la requête `DeleteItem` supprime un élément existant quel que soit son état actuel. Pour plus d'informations sur les conditions, voirExpressions de conditions. Cette valeur est facultative.

`_version`

Valeur numérique représentant la dernière version connue d'un élément. Cette valeur est facultative. Ce champ est utilisé pour la détection de conflits et n'est pris en charge que sur les sources de données versionnées.

`customPartitionKey`

Lorsqu'elle est activée, cette valeur de chaîne modifie le format du`ds_sk`et`ds_pk`enregistrements utilisés par la table de synchronisation delta lorsque le versionnement a été activé (pour plus d'informations, voirDétection et synchronisation des conflitsdans leAWS AppSyncGuide du développeur). Lorsque cette option est activée, le traitement du`populateIndexFields`la saisie est également activée. Ce champ est facultatif.

`populateIndexFields`

Une valeur booléenne qui, lorsqu'elle est activéeainsi que le**`customPartitionKey`**, crée de nouvelles entrées pour chaque enregistrement de la table de synchronisation delta, en particulier dans`gsi_ds_pk`et`gsi_ds_sk`colonnes. Pour plus d'informations, voirDétection et synchronisation des conflitsdans leAWS AppSyncGuide du développeur. Ce champ est facultatif.

L'élément supprimé de DynamoDB est automatiquement converti en types primitifs GraphQL et JSON et est disponible dans le résultat du contexte (`context.result`).

Pour plus d'informations sur la conversion de type DynamoDB, voirSystème de types (mappage des réponses).

Pour plus d'informations surJavaScriptrésolveurs, voirJavaScriptvue d'ensemble des résolveurs.

Exemple 1

L'exemple suivant est un gestionnaire de demande de fonction pour une mutation GraphQL`deleteItem(id: ID!)`. S'il existe déjà un élément avec cet ID, il est supprimé.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'DeleteItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}
```

Exemple 2

L'exemple suivant est un gestionnaire de demande de fonction pour une mutation GraphQL`deleteItem(id: ID!, expectedVersion: Int!)`. S'il existe déjà un élément avec cet ID, il est supprimé, mais uniquement si son champ `version` est défini sur `expectedVersion` :

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { id, expectedVersion } = ctx.args;
  const condition = {
    id: { attributeExists: true },
    version: { eq: expectedVersion },
  };
  return {
    operation: 'DeleteItem',
    key: util.dynamodb.toMapValues({ id }),
    condition: util.transform.toDynamoDBConditionExpression(condition),
  };
}
```

Pour de plus amples informations sur l'API de DynamoDB `DeleteItem`, veuillez consulter la Documentation sur les API de DynamoDB.

Query

Le`Query`l'objet de requête vous permet de direAWS AppSyncRésolveur DynamoDB pour créer un`Query`demande à DynamoDB, et vous permet de spécifier les éléments suivants :

- Expression de la clé
- Quel index utiliser
- Tout filtre supplémentaire
- Combien d'articles renvoyer
- S'il convient d'utiliser une lecture cohérente
- Sens de la requête (vers l'avant ou l'arrière)
- Jeton de pagination

Le`Query`L'objet de requête a la structure suivante :

```
type DynamoDBQueryRequest = {
  operation: 'Query';
  query: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
  index?: string;
  nextToken?: string;
  limit?: number;
  scanIndexForward?: boolean;
  consistentRead?: boolean;
  select?: 'ALL_ATTRIBUTES' | 'ALL_PROJECTED_ATTRIBUTES' | 'SPECIFIC_ATTRIBUTES';
  filter?: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
  projection?: {
    expression: string;
    expressionNames?: { [key: string]: string };
  };
};
```

Les champs sont définis comme suit :

Champs de requête

Liste des champs de requête

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `Query`, ce champ doit être défini sur `Query`. Cette valeur est obligatoire.

`query`

Le`query`cette section vous permet de spécifier une expression de condition clé qui décrit les éléments à récupérer depuis DynamoDB. Pour plus d'informations sur la façon d'écrire des expressions de conditions clés, consultezDynamoDBKeyConditionsdocumentation. Cette section doit être spécifiée.

`expression`

Expression de la requête. Ce champ doit être spécifié.

`expressionNames`

Substituts des espaces réservés de nom des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé au nom utilisé dans`expression`, et la valeur doit être une chaîne correspondant au nom d'attribut de l'élément dans DynamoDB. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de nom des attributs de l'expression utilisés dans l'`expression`.

`expressionValues`

Substituts des espaces réservés de valeur des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé de valeur utilisé dans l'`expression`, et la valeur doit être typée. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Cette valeur est obligatoire. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de valeur des attributs de l'expression utilisés dans l'`expression`.

`filter`

Filtre supplémentaire qui peut être utilisé pour filtrer les résultats de DynamoDB avant qu'ils soient renvoyés. Pour plus d'informations sur les filtres, consultez Filters (Filtres). Ce champ est facultatif.

`index`

Nom de l'index à interroger. L'opération de requête DynamoDB vous permet de rechercher une clé de hachage sur les index secondaires locaux et les index secondaires globaux en plus de l'index de clé primaire. Si cela est spécifié, cela indique à DynamoDB d'interroger l'index spécifié. Si elle ne l'est pas, l'index de clé primaire est interrogé.

`nextToken`

Jeton de pagination pour continuer une requête précédente. Il a été obtenu à partir d'une requête précédente. Ce champ est facultatif.

`limit`

Nombre maximum d'éléments à évaluer (pas nécessairement le nombre d'éléments correspondants). Ce champ est facultatif.

`scanIndexForward`

Valeur booléenne indiquant s'il convient d'interroger vers l'avant ou vers l'arrière. Ce champ est facultatif et contient `true` par défaut.

`consistentRead`

Un booléen indiquant s'il faut utiliser des lectures cohérentes lors de l'interrogation de DynamoDB. Ce champ est facultatif et contient `false` par défaut.

`select`

Par défaut, leAWS AppSyncLe résolveur DynamoDB renvoie uniquement les attributs projetés dans l'index. Si un plus grand nombre d'attributs est requis, vous pouvez définir ce champ. Ce champ est facultatif. Les valeurs prises en charge sont :

`ALL_ATTRIBUTES`

Renvoie tous les attributs de l'élément depuis la table ou l'index spécifié. Si vous interrogez un index secondaire local, DynamoDB extrait l'élément entier de la table parent pour chaque élément correspondant dans l'index. Si l'index est configuré de façon à projeter tous les attributs de l'élément, toutes les données peuvent être obtenues à partir de l'index secondaire local, et aucune extraction n'est nécessaire.

`ALL_PROJECTED_ATTRIBUTES`

Autorisé seulement lorsque vous interrogez un index. Extrait tous les attributs qui ont été projetés dans l'index. Si l'index est configuré de façon à projeter tous les attributs, la valeur renvoyée revient à spécifier `ALL_ATTRIBUTES`.

`SPECIFIC_ATTRIBUTES`

Renvoie uniquement les attributs répertoriés dans le`projection`c'est`expression`. Cette valeur de retour est équivalente à la spécification de`projection`c'est`expression`sans spécifier de valeur pour`Select`.

`projection`

Projection utilisée pour spécifier les attributs à renvoyer par l'opération DynamoDB. Pour plus d'informations sur les projections, voirProjections. Ce champ est facultatif.

Les résultats de DynamoDB sont automatiquement convertis en types primitifs GraphQL et JSON et sont disponibles dans le résultat contextuel (`context.result`).

Pour plus d'informations sur la conversion de type DynamoDB, voirSystème de types (mappage des réponses).

Pour plus d'informations surJavaScriptrésolveurs, voirJavaScriptvue d'ensemble des résolveurs.

Les résultats ont la structure suivante :

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

Les champs sont définis comme suit :

`items`

Liste contenant les éléments renvoyés par la requête DynamoDB.

`nextToken`

S'il peut y avoir plusieurs résultats, `nextToken` contient un jeton de pagination que vous pouvez utiliser dans une autre requête. Notez queAWS AppSyncchiffre et masque le jeton de pagination renvoyé par DynamoDB. Cela empêche que les données provenant de votre table ne soient accidentellement communiquées au mandataire. Notez également que ces jetons de pagination ne peuvent pas être utilisés entre différentes fonctions ou résolveurs.

`scannedCount`

Nombre d'éléments correspondant à l'expression de condition de requête, avant qu'une expression de filtre (si elle est présente) ne soit appliquée.

Exemple

L'exemple suivant est un gestionnaire de demande de fonction pour une requête GraphQL`getPosts(owner: ID!)`.

Dans cet exemple, un index secondaire global sur une table est interrogé afin de renvoyer toutes les publications détenues par l'ID spécifié.

```
import { util } from '@aws-appsync/utils';
```

```
export function request(ctx) {
  const { owner } = ctx.args;
  return {
    operation: 'Query',
    query: {
      expression: 'ownerId = :ownerId',
      expressionValues: util.dynamodb.toMapValues({ ':ownerId': owner }),
    },
    index: 'owner-index',
  };
}
```

Pour de plus amples informations sur l'API de DynamoDB `Query`, veuillez consulter la Documentation sur les API de DynamoDB.

Analyser

Le`Scan`la demande vous permet de direAWS AppSyncFonction DynamoDB pour créer un`Scan`demande à DynamoDB, et vous permet de spécifier les éléments suivants :

- Filtre pour exclure des résultats
- Quel index utiliser
- Combien d'articles renvoyer
- S'il convient d'utiliser une lecture cohérente
- Jeton de pagination
- Analyses parallèles

Le`Scan`L'objet de requête a la structure suivante :

```
type DynamoDBScanRequest = {
  operation: 'Scan';
  index?: string;
  limit?: number;
  consistentRead?: boolean;
  nextToken?: string;
  totalSegments?: number;
  segment?: number;
  filter?: {
    expression: string;
```

```
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
  projection?: {
    expression: string;
    expressionNames?: { [key: string]: string };
  };
};
```

Les champs sont définis comme suit :

Numériser les champs

Numériser la liste des champs

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `Scan`, ce champ doit être défini sur `Scan`. Cette valeur est obligatoire.

`filter`

Filtre qui peut être utilisé pour filtrer les résultats de DynamoDB avant qu'ils ne soient renvoyés. Pour plus d'informations sur les filtres, consultez Filters (Filtres). Ce champ est facultatif.

`index`

Nom de l'index à interroger. L'opération de requête DynamoDB vous permet de rechercher une clé de hachage sur les index secondaires locaux et les index secondaires globaux en plus de l'index de clé primaire. Si cela est spécifié, cela indique à DynamoDB d'interroger l'index spécifié. Si elle ne l'est pas, l'index de clé primaire est interrogé.

`limit`

Nombre maximal d'éléments à évaluer simultanément. Ce champ est facultatif.

`consistentRead`

Un booléen qui indique s'il faut utiliser des lectures cohérentes lors de l'interrogation de DynamoDB. Ce champ est facultatif et contient `false` par défaut.

`nextToken`

Jeton de pagination pour continuer une requête précédente. Il a été obtenu à partir d'une requête précédente. Ce champ est facultatif.

`select`

Par défaut, leAWS AppSyncLa fonction DynamoDB renvoie uniquement les attributs projetés dans l'index. Si un plus grand nombre d'attributs est requis, ce champ peut être défini. Ce champ est facultatif. Les valeurs prises en charge sont :

`ALL_ATTRIBUTES`

Renvoie tous les attributs de l'élément depuis la table ou l'index spécifié. Si vous interrogez un index secondaire local, DynamoDB extrait l'élément entier de la table parent pour chaque élément correspondant dans l'index. Si l'index est configuré de façon à projeter tous les attributs de l'élément, toutes les données peuvent être obtenues à partir de l'index secondaire local, et aucune extraction n'est nécessaire.

`ALL_PROJECTED_ATTRIBUTES`

Autorisé seulement lorsque vous interrogez un index. Extrait tous les attributs qui ont été projetés dans l'index. Si l'index est configuré de façon à projeter tous les attributs, la valeur renvoyée revient à spécifier `ALL_ATTRIBUTES`.

`SPECIFIC_ATTRIBUTES`

Renvoie uniquement les attributs répertoriés dans le`projection`c'est`expression`. Cette valeur de retour est équivalente à la spécification de`projection`c'est`expression`sans spécifier de valeur pour`Select`.

`totalSegments`

Nombre de segments pour partitionner la table lors de l'exécution d'une analyse parallèle. Ce champ est facultatif, mais doit être spécifié si `segment` est spécifié.

`segment`

Segment de table de cette opération lorsque vous effectuez une analyse parallèle. Ce champ est facultatif, mais doit être spécifié si `totalSegments` est spécifié.

`projection`

Projection utilisée pour spécifier les attributs à renvoyer par l'opération DynamoDB. Pour plus d'informations sur les projections, voirProjections. Ce champ est facultatif.

Les résultats renvoyés par le scan DynamoDB sont automatiquement convertis en types primitifs GraphQL et JSON et sont disponibles dans le résultat contextuel (`context.result`).

Pour plus d'informations sur la conversion de type DynamoDB, voirSystème de types (mappage des réponses).

Pour plus d'informations surJavaScriptrésolveurs, voirJavaScriptvue d'ensemble des résolveurs.

Les résultats ont la structure suivante :

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

Les champs sont définis comme suit :

`items`

Liste contenant les éléments renvoyés par le scan DynamoDB.

`nextToken`

S'il devait y avoir d'autres résultats,`nextToken`contient un jeton de pagination que vous pouvez utiliser dans une autre requête.AWS AppSyncchiffre et masque le jeton de pagination renvoyé par DynamoDB. Cela empêche que les données provenant de votre table ne soient accidentellement communiquées au mandataire. De plus, ces jetons de pagination ne peuvent pas être utilisés entre différentes fonctions ou résolveurs.

`scannedCount`

Nombre d'éléments récupérés par DynamoDB avant l'application d'une expression de filtre (le cas échéant).

Exemple 1

L'exemple suivant est un gestionnaire de demande de fonction pour la requête GraphQL :`allPosts`.

Dans cet exemple, toutes les entrées de la table sont renvoyées.

```
export function request(ctx) {
  return { operation: 'Scan' };
}
```

Exemple 2

L'exemple suivant est un gestionnaire de demande de fonction pour la requête GraphQL :`postsMatching(title: String!)`.

Dans cet exemple, toutes les entrées de la table sont renvoyées lorsque le titre commence par l'argument `title`.

```
export function request(ctx) {
  const { title } = ctx.args;
  const filter = { filter: { beginsWith: title } };
  return {
    operation: 'Scan',
    filter: JSON.parse(util.transform.toDynamoDBFilterExpression(filter)),
  };
}
```

Pour de plus amples informations sur l'API de DynamoDB `Scan`, veuillez consulter la Documentation sur les API de DynamoDB.

Sync

Le`Sync`L'objet de requête vous permet de récupérer tous les résultats d'une table DynamoDB, puis de ne recevoir que les données modifiées depuis votre dernière requête (le delta est mis à jour).`Sync`les demandes ne peuvent être adressées qu'à des sources de données DynamoDB versionnées. Vous pouvez spécifier les valeurs suivantes :

- Filtre pour exclure des résultats
- Combien d'articles renvoyer
- Jeton de pagination.
- Lorsque votre dernière opération `Sync` a été lancée

Le`Sync`L'objet de requête a la structure suivante :

```
type DynamoDBSyncRequest = {
  operation: 'Sync';
  basePartitionKey?: string;
  deltaIndexName?: string;
  limit?: number;
```

```
  nextToken?: string;
  lastSync?: number;
  filter?: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
};
```

Les champs sont définis comme suit :

Synchroniser les champs

Liste des champs de synchronisation

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération `Sync`, ce champ doit être défini sur `Sync`. Cette valeur est obligatoire.

`filter`

Filtre qui peut être utilisé pour filtrer les résultats de DynamoDB avant qu'ils ne soient renvoyés. Pour plus d'informations sur les filtres, consultez Filters (Filtres). Ce champ est facultatif.

`limit`

Nombre maximal d'éléments à évaluer simultanément. Ce champ est facultatif. Si cette option est omise, la limite par défaut sera définie sur `100` éléments. La valeur maximale de ce champ est `1000` éléments.

`nextToken`

Jeton de pagination pour continuer une requête précédente. Il a été obtenu à partir d'une requête précédente. Ce champ est facultatif.

`lastSync`

Le moment, en millisecondes Epoch, où la dernière opération `Sync` réussie a commencé. Si spécifié, seuls les éléments qui ont changé après `lastSync` sont retournés. Ce champ est facultatif et ne doit être renseigné qu'après avoir récupéré toutes les pages d'une opération `Sync` initiale. Si cette option est omise, les résultats de la table Base seront retournés, sinon les résultats de la table Delta seront retournés.

basePartitionKey

La clé de partition duBasetable utilisée lors de l'exécution d'unSyncopération. Ce champ permet deSyncopération à effectuer lorsque la table utilise une clé de partition personnalisée. Il s'agit d'un champ facultatif.

deltaIndexName

L'indice utilisé pour leSyncopération. Cet index est nécessaire pour activer unSyncopération sur l'ensemble de la table delta store lorsque la table utilise une clé de partition personnalisée. LeSyncl'opération sera effectuée sur le GSI (créé legsi_ds_pketgsi_ds_sk). Ce champ est facultatif.

Les résultats renvoyés par la synchronisation DynamoDB sont automatiquement convertis en types primitifs GraphQL et JSON et sont disponibles dans le résultat contextuel (`context.result`).

Pour plus d'informations sur la conversion de type DynamoDB, voirSystème de types (mappage des réponses).

Pour plus d'informations surJavaScriptrésolveurs, voirJavaScriptvue d'ensemble des résolveurs.

Les résultats ont la structure suivante :

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10,
    startedAt = 1550000000000
}
```

Les champs sont définis comme suit :

items

Liste contenant les éléments renvoyés par la synchronisation.

nextToken

S'il devait y avoir d'autres résultats,`nextToken`contient un jeton de pagination que vous pouvez utiliser dans une autre requête.AWS AppSyncchiffre et masque le jeton de pagination renvoyé par DynamoDB. Cela empêche que les données provenant de votre table ne soient accidentellement

communiquées au mandataire. De plus, ces jetons de pagination ne peuvent pas être utilisés entre différentes fonctions ou résolveurs.

`scannedCount`

Nombre d'éléments récupérés par DynamoDB avant l'application d'une expression de filtre (le cas échéant).

`startedAt`

Le moment, en millisecondes Epoch, où l'opération de synchronisation a commencé, que vous pouvez stocker localement et utiliser dans une autre requête comme argument `lastSync`. Si un jeton de pagination a été inclus dans la requête, cette valeur sera la même que celle renvoyée par la requête pour la première page de résultats.

Exemple 1

L'exemple suivant est un gestionnaire de demande de fonction pour la requête GraphQL :`syncPosts(nextToken: String, lastSync: AWSTimestamp)`.

Dans cet exemple, si `lastSync` est omis, toutes les entrées de la table de base sont renvoyées. Si `lastSync` est fourni, seules les entrées de la table de synchronisation delta qui ont changé depuis `lastSync` sont renvoyées.

```
export function request(ctx) {
  const { nextToken, lastSync } = ctx.args;
  return { operation: 'Sync', limit: 100, nextToken, lastSync };
}
```

BatchGetItem

Le`BatchGetItem`l'objet de requête vous permet de direAWS AppSyncFonction DynamoDB pour créer un`BatchGetItem`demande à DynamoDB pour récupérer plusieurs éléments, potentiellement sur plusieurs tables. Pour cet objet de demande, vous devez spécifier les éléments suivants :

- Les noms de tables à partir desquels récupérer les éléments
- Les clés des éléments à récupérer dans chaque table

Les limites `BatchGetItem` DynamoDB s'appliquent et aucune expression de condition ne peut être fournie.

LeBatchGetItemL'objet de requête a la structure suivante :

```
type DynamoDBBatchGetItemRequest = {
  operation: 'BatchGetItem';
  tables: {
    [tableName: string]: {
      keys: { [key: string]: any }[];
      consistentRead?: boolean;
      projection?: {
        expression: string;
        expressionNames?: { [key: string]: string };
      };
    };
  };
};
```

Les champs sont définis comme suit :

Champs de BatchGetItem

BatchGetItemliste des champs

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `BatchGetItem`, ce champ doit être défini sur `BatchGetItem`. Cette valeur est obligatoire.

`tables`

Les tables DynamoDB à partir desquelles récupérer les éléments. La valeur est une carte où les noms de table sont spécifiés en tant que clés de la carte. Vous devez fournir au moins une table. Cette valeur `tables` est obligatoire.

`keys`

Liste des clés DynamoDB représentant la clé primaire des éléments à récupérer. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes).

`consistentRead`

S'il faut utiliser une lecture cohérente lors de l'exécution d'unGetItemopération. Cette valeur est facultative et est définie comme faux par défaut.

`projection`

Projection utilisée pour spécifier les attributs à renvoyer par l'opération DynamoDB. Pour plus d'informations sur les projections, voirProjections. Ce champ est facultatif.

Objets à mémoriser :

- Si un élément n'a pas été récupéré à partir de la table, un élément null s'affiche dans le bloc de données pour cette table.
- Les résultats d'invocation sont triés par table, en fonction de l'ordre dans lequel ils ont été fournis dans l'objet de demande.
- ChaqueGetcommande dans unBatchGetItemest atomique, cependant, un lot peut être partiellement traité. Si un lot est traité partiellement en raison d'une erreur, les clés non traitées sont renvoyées dans le cadre du résultat de l'appel dans le bloc unprocessedKeys.
- `BatchGetItem` est limité à 100 clés.

Pour l'exemple de gestionnaire de demandes de fonction suivant :

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'BatchGetItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId })],
      posts: [util.dynamodb.toMapValues({ authorId, postId })],
    },
  };
}
```

Le résultat de l'appel disponible dans `ctx.result` est le suivant :

```
{
   "data": {
     "authors": [null],
     "posts": [
        // Was retrieved
        {
```

```
          "authorId": "a1",
          "postId": "p2",
          "postTitle": "title",
          "postDescription": "description",
        }
      ]
   },
   "unprocessedKeys": {
     "authors": [
        // This item was not processed due to an error
        {
          "authorId": "a1"
        }
      ],
     "posts": []
   }
}
```

`ctx.error` contient des détails sur l'erreur. Les clésdonnées,Clés non traitées, et chaque clé de table fournie dans le résultat de l'objet de demande de fonction est garantie d'être présente dans le résultat de l'invocation. Les éléments ayant été supprimés apparaissent dans le bloc de données. Les éléments qui n'ont pas été traités sont marqués comme null dans le bloc de données et sont placés dans le bloc unprocessedKeys.

BatchDeleteItem

Le`BatchDeleteItem`l'objet de requête vous permet de direAWS AppSyncFonction DynamoDB pour créer un`BatchWriteItem`demande à DynamoDB de supprimer plusieurs éléments, éventuellement sur plusieurs tables. Pour cet objet de demande, vous devez spécifier les éléments suivants :

- Les noms de tables à partir desquels supprimer les éléments
- Les clés des éléments à supprimer dans chaque table

Les limites `BatchWriteItem` DynamoDB s'appliquent et aucune expression de condition ne peut être fournie.

Le`BatchDeleteItem`L'objet de requête a la structure suivante :

```
type DynamoDBBatchDeleteItemRequest = {
  operation: 'BatchDeleteItem';
  tables: {
```

```
    [tableName: string]: { [key: string]: any }[];
  };
};
```

Les champs sont définis comme suit :

Champs de BatchDeleteItem

BatchDeleteItemliste des champs

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `BatchDeleteItem`, ce champ doit être défini sur `BatchDeleteItem`. Cette valeur est obligatoire.

`tables`

Les tables DynamoDB dont les éléments doivent être supprimés. Chaque table est une liste de clés DynamoDB représentant la clé primaire des éléments à supprimer. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Vous devez fournir au moins une table. L`etables`valeur est obligatoire.

Objets à mémoriser :

- Contrairement à l'opération `DeleteItem`, l'élément complètement supprimé n'est pas renvoyé dans la réponse. Seule la clé passée est renvoyée.
- Si un élément n'a pas été supprimé à partir de la table, un élément null s'affiche dans le bloc de données pour cette table.
- Les résultats d'invocation sont triés par table, en fonction de l'ordre dans lequel ils ont été fournis dans l'objet de demande.
- Chaque`Delete`commande dans un`BatchDeleteItem`est atomique. Cependant, un lot peut être partiellement traité. Si un lot est traité partiellement en raison d'une erreur, les clés non traitées sont renvoyées dans le cadre du résultat de l'appel dans le bloc unprocessedKeys.
- `BatchDeleteItem` est limité à 25 clés.

Pour l'exemple de gestionnaire de demandes de fonction suivant :

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'BatchDeleteItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId })],
      posts: [util.dynamodb.toMapValues({ authorId, postId })],
    },
  };
}
```

Le résultat de l'appel disponible dans `ctx.result` est le suivant :

```
{
   "data": {
     "authors": [null],
     "posts": [
        // Was deleted
        {
          "authorId": "a1",
          "postId": "p2"
        }
     ]
   },
   "unprocessedKeys": {
     "authors": [
        // This key was not processed due to an error
        {
          "authorId": "a1"
        }
      ],
     "posts": []
   }
}
```

`ctx.error` contient des détails sur l'erreur. Les clésdonnées,Clés non traitées, et la présence de chaque clé de table fournie dans l'objet de demande de fonction est garantie dans le résultat de l'appel. Les éléments ayant été supprimés sont présents dans le bloc de données. Les éléments qui n'ont pas été traités sont marqués comme null dans le bloc de données et sont placés dans le bloc unprocessedKeys.

BatchPutItem

Le`BatchPutItem`l'objet de requête vous permet de direAWS AppSyncFonction DynamoDB pour créer un`BatchWriteItem`demande à DynamoDB de placer plusieurs éléments, potentiellement sur plusieurs tables. Pour cet objet de demande, vous devez spécifier les éléments suivants :

- Les noms de tables dans lesquels placer les éléments
- Les éléments complets à placer dans chaque table

Les limites `BatchWriteItem` DynamoDB s'appliquent et aucune expression de condition ne peut être fournie.

Le`BatchPutItem`L'objet de requête a la structure suivante :

```
type DynamoDBBatchPutItemRequest = {
  operation: 'BatchPutItem';
  tables: {
    [tableName: string]: { [key: string]: any}[];
  };
};
```

Les champs sont définis comme suit :

Champs de BatchPutItem

BatchPutItemliste des champs

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `BatchPutItem`, ce champ doit être défini sur `BatchPutItem`. Cette valeur est obligatoire.

`tables`

Les tables DynamoDB dans lesquelles placer les éléments. Chaque entrée de table représente une liste d'éléments DynamoDB à insérer pour cette table spécifique. Vous devez fournir au moins une table. Cette valeur est obligatoire.

Objets à mémoriser :

- Les éléments entièrement insérés sont renvoyés dans la réponse, en cas de succès.
- Si un élément n'a pas été inséré dans la table, un élément null s'affiche dans le bloc de données pour cette table.
- Les éléments insérés sont triés par table, en fonction de l'ordre dans lequel ils ont été fournis dans l'objet de demande.
- ChaquePutcommande dans unBatchPutItemest atomique, cependant, un lot peut être partiellement traité. Si un lot est traité partiellement en raison d'une erreur, les clés non traitées sont renvoyées dans le cadre du résultat de l'appel dans le bloc unprocessedKeys.
- `BatchPutItem` est limité à 25 éléments.

Pour l'exemple de gestionnaire de demandes de fonction suivant :

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId, name, title } = ctx.args;
  return {
    operation: 'BatchPutItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId, name })],
      posts: [util.dynamodb.toMapValues({ authorId, postId, title })],
    },
  };
}
```

Le résultat de l'appel disponible dans `ctx.result` est le suivant :

```
{
   "data": {
     "authors": [
         null
     ],
     "posts": [
        // Was inserted
        {
          "authorId": "a1",
          "postId": "p2",
          "title": "title"
        }
```

```
      ]
    },
    "unprocessedItems": {
      "authors": [
         // This item was not processed due to an error
         {
           "authorId": "a1",
           "name": "a1_name"
         }
       ],
      "posts": []
    }
}
```

`ctx.error` contient des détails sur l'erreur. Les clésdonnées,Articles non traités, et la présence de chaque clé de table fournie dans l'objet de demande est garantie dans le résultat de l'appel. Les éléments ayant été insérés sont dans le bloc de données. Les éléments qui n'ont pas été traités sont marqués comme null dans le bloc de données et sont placés dans le bloc unprocessedItems.

TransactGetItems

Le`TransactGetItems`l'objet de requête vous permet de direAWS AppSyncFonction DynamoDB pour créer un`TransactGetItems`demande à DynamoDB pour récupérer plusieurs éléments, potentiellement sur plusieurs tables. Pour cet objet de demande, vous devez spécifier les éléments suivants :

- Nom de la table de chaque élément de requête dans lequel extraire l'élément
- La clé de chaque élément de requête à récupérer à partir de chaque table

Les limites `TransactGetItems` DynamoDB s'appliquent et aucune expression de condition ne peut être fournie.

Le`TransactGetItems`L'objet de requête a la structure suivante :

```
type DynamoDBTransactGetItemsRequest = {
  operation: 'TransactGetItems';
  transactItems: { table: string; key: { [key: string]: any }; projection?:
 { expression: string; expressionNames?: { [key: string]: string }; }[];
  };
};
```

Les champs sont définis comme suit :

Champs de TransactGetItems

TransactGetItemsliste des champs

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `TransactGetItems`, ce champ doit être défini sur `TransactGetItems`. Cette valeur est obligatoire.

`transactItems`

Les éléments de requête à inclure. La valeur est un tableau d'éléments de requête. Au moins un élément de requête doit être fourni. Cette valeur `transactItems` est obligatoire.

`table`

La table DynamoDB à partir de laquelle récupérer l'élément. La valeur est une chaîne du nom de la table. Cette valeur `table` est obligatoire.

`key`

La clé DynamoDB représentant la clé primaire de l'élément à récupérer. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes).

`projection`

Projection utilisée pour spécifier les attributs à renvoyer par l'opération DynamoDB. Pour plus d'informations sur les projections, voirProjections. Ce champ est facultatif.

Objets à mémoriser :

- Si une transaction réussit, l'ordre des éléments récupérés dans le bloc `items` sera le même que celui des éléments de la requête.
- Les transactions sont effectuées dans unall-or-nothingfaçon. Si un élément de requête provoque une erreur, la transaction entière ne sera pas effectuée et les détails de l'erreur seront retournés.
- Un élément de requête qui ne peut pas être récupéré n'est pas une erreur. Au lieu de cela, un élément null apparaît dans le bloc éléments dans la position correspondante.

- Si l'erreur d'une transaction estTransactionCanceledException, le`cancellationReasons`le bloc sera rempli. L'ordre des motifs d'annulation dans le bloc `cancellationReasons` sera le même que l'ordre des éléments de demande.
- `TransactGetItems` est limité à 25 éléments de demande.

Pour l'exemple de gestionnaire de demandes de fonction suivant :

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'TransactGetItems',
    transactItems: [
      {
        table: 'posts',
        key: util.dynamodb.toMapValues({ postId }),
      },
      {
        table: 'authors',
        key: util.dynamodb.toMapValues({ authorId }),
      },
    ],
  };
}
```

Si la transaction réussit et que seul le premier élément demandé est extrait, le résultat d'appel disponible dans `ctx.result` est le suivant :

```
{
    "items": [
       {
           // Attributes of the first requested item
           "post_id": "p1",
           "post_title": "title",
           "post_description": "description"
       },
       // Could not retrieve the second requested item
       null,
    ],
    "cancellationReasons": null
```

```
}
```

Si la transaction échoue en raison deTransactionCanceledExceptioncausé par le premier élément de demande, le résultat de l'invocation est disponible dans`ctx.result`est le suivant :

```
{
    "items": null,
    "cancellationReasons": [
       {
           "type":"Sample error type",
           "message":"Sample error message"
       },
       {
           "type":"None",
           "message":"None"
       }
    ]
}
```

`ctx.error` contient des détails sur l'erreur. Les éléments clés et cancellationReasons sont nécessairement présents dans `ctx.result`.

TransactWriteItems

LE`TransactWriteItems`l'objet de requête vous permet de direAWS AppSyncFonction DynamoDB pour créer un`TransactWriteItems`demande à DynamoDB d'écrire plusieurs éléments, éventuellement dans plusieurs tables. Pour cet objet de demande, vous devez spécifier les éléments suivants :

- Nom de la table de destination de chaque élément de requête
- L'opération de chaque élément de demande à effectuer. Quatre types d'opérations sont pris en charge :PutItem,UpdateItem,DeleteItem, etConditionCheck
- La clé de chaque élément de demande à écrire

Les limites `TransactWriteItems` DynamoDB s'appliquent.

LE`TransactWriteItems`L'objet de requête a la structure suivante :

```
type DynamoDBTransactWriteItemsRequest = {
  operation: 'TransactWriteItems';
```

```
  transactItems: TransactItem[];
};
type TransactItem =
  | TransactWritePutItem
  | TransactWriteUpdateItem
  | TransactWriteDeleteItem
  | TransactWriteConditionCheckItem;
type TransactWritePutItem = {
  table: string;
  operation: 'PutItem';
  key: { [key: string]: any };
  attributeValues: { [key: string]: string};
  condition?: TransactConditionCheckExpression;
};
type TransactWriteUpdateItem = {
  table: string;
  operation: 'UpdateItem';
  key: { [key: string]: any };
  update: DynamoDBExpression;
  condition?: TransactConditionCheckExpression;
};
type TransactWriteDeleteItem = {
  table: string;
  operation: 'DeleteItem';
  key: { [key: string]: any };
  condition?: TransactConditionCheckExpression;
};
type TransactWriteConditionCheckItem = {
  table: string;
  operation: 'ConditionCheck';
  key: { [key: string]: any };
  condition?: TransactConditionCheckExpression;
};
type TransactConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
  returnValuesOnConditionCheckFailure: boolean;
};
```

Champs de TransactWriteItems

TransactWriteItemsliste des champs

Les champs sont définis comme suit :

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `TransactWriteItems`, ce champ doit être défini sur `TransactWriteItems`. Cette valeur est obligatoire.

`transactItems`

Les éléments de requête à inclure. La valeur est un tableau d'éléments de requête. Au moins un élément de requête doit être fourni. Cette valeur `transactItems` est obligatoire.

Pour `PutItem`, les champs sont définis comme suit :

`table`

La table DynamoDB de destination. La valeur est une chaîne du nom de la table. Cette valeur `table` est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `PutItem`, ce champ doit être défini sur `PutItem`. Cette valeur est obligatoire.

`key`

La clé DynamoDB représentant la clé primaire de l'élément à insérer. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Cette valeur est obligatoire.

`attributeValues`

Le reste des attributs de l'élément doit être placé dans DynamoDB. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Ce champ est facultatif.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Si aucune condition n'est spécifiée, la requête `PutItem` remplace toute entrée existante pour cet élément. Vous pouvez spécifier s'il faut récupérer l'élément existant en cas d'échec de la vérification de l'état. Pour plus d'informations sur les conditions transactionnelles, voirExpressions des conditions de transaction. Cette valeur est facultative.

Pour `UpdateItem`, les champs sont définis comme suit :

`table`

Table DynamoDB à mettre à jour. La valeur est une chaîne du nom de la table. Cette valeur `table` est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `UpdateItem`, ce champ doit être défini sur `UpdateItem`. Cette valeur est obligatoire.

`key`

La clé DynamoDB représentant la clé primaire de l'élément à mettre à jour. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Cette valeur est obligatoire.

`update`

Le`update`cette section vous permet de spécifier une expression de mise à jour qui décrit comment mettre à jour l'élément dans DynamoDB. Pour plus d'informations sur la façon d'écrire des expressions de mise à jour, consultez leDynamoDBUpdateExpressionsdocumentation. Cette section est obligatoire.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Si aucune condition n'est spécifiée, la requête `UpdateItem` met à jour l'entrée existante quel que soit son état actuel. Vous pouvez spécifier s'il faut récupérer l'élément existant en cas d'échec de la vérification de l'état. Pour

plus d'informations sur les conditions transactionnelles, voirExpressions des conditions de transaction. Cette valeur est facultative.

Pour `DeleteItem`, les champs sont définis comme suit :

`table`

Table DynamoDB dans laquelle vous souhaitez supprimer l'élément. La valeur est une chaîne du nom de la table. Cette valeur `table` est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `DeleteItem`, ce champ doit être défini sur `DeleteItem`. Cette valeur est obligatoire.

`key`

La clé DynamoDB représentant la clé primaire de l'élément à supprimer. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Cette valeur est obligatoire.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Si aucune condition n'est spécifiée, la requête `DeleteItem` supprime un élément existant quel que soit son état actuel. Vous pouvez spécifier s'il faut récupérer l'élément existant en cas d'échec de la vérification de l'état. Pour plus d'informations sur les conditions transactionnelles, voirExpressions des conditions de transaction. Cette valeur est facultative.

Pour `ConditionCheck`, les champs sont définis comme suit :

`table`

La table DynamoDB dans laquelle vérifier la condition. La valeur est une chaîne du nom de la table. Cette valeur `table` est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `ConditionCheck`, ce champ doit être défini sur `ConditionCheck`. Cette valeur est obligatoire.

`key`

La clé DynamoDB représentant la clé primaire de l'élément à vérifier. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Cette valeur est obligatoire.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Vous pouvez spécifier s'il faut récupérer l'élément existant en cas d'échec de la vérification de l'état. Pour plus d'informations sur les conditions transactionnelles, voirExpressions des conditions de transaction. Cette valeur est obligatoire.

Objets à mémoriser :

- Seules les clés des éléments de demande sont renvoyées dans la réponse, si elles réussissent. L'ordre des clés sera le même que l'ordre des éléments de demande.
- Les transactions sont effectuées dans unall-or-nothingfaçon. Si un élément de requête provoque une erreur, la transaction entière ne sera pas effectuée et les détails de l'erreur seront retournés.
- Aucun élément de demande ne peut cibler le même élément. Sinon, ils provoquerontTransactionCanceledExceptionerreur.
- Si l'erreur d'une transaction estTransactionCanceledException, le`cancellationReasons`le bloc sera rempli. Si la vérification de l'état d'un élément de demande échoue et que vous n'avez pas spécifié `returnValuesOnConditionCheckFailure` comme étant `false`, l'élément existant dans la table est récupéré et stocké `item` à la position correspondante du bloc `cancellationReasons`.
- `TransactWriteItems` est limité à 25 éléments de demande.

Pour l'exemple de gestionnaire de demandes de fonction suivant :

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId, title, description, oldTitle, authorName } = ctx.args;
```

```
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({ postId }),
        attributeValues: util.dynamodb.toMapValues({ title, description }),
        condition: util.transform.toDynamoDBConditionExpression({
          title: { eq: oldTitle },
        }),
      },
      {
        table: 'authors',
        operation: 'UpdateItem',
        key: util.dynamodb.toMapValues({ authorId }),
        update: {
          expression: 'SET authorName = :name',
          expressionValues: util.dynamodb.toMapValues({ ':name': authorName }),
        },
      },
    ],
  };
}
```

Si la transaction réussit, le résultat d'appel disponible dans `ctx.result` est le suivant :

```
{
    "keys": [
       // Key of the PutItem request
       {
           "post_id": "p1",
       },
       // Key of the UpdateItem request
       {
           "author_id": "a1"
       }
    ],
    "cancellationReasons": null
}
```

Si la transaction échoue en raison d'un échec de la vérification de l'état du`PutItem`demande, le résultat de l'invocation disponible dans`ctx.result`est le suivant :

```
{
    "keys": null,
    "cancellationReasons": [
       {
           "item": {
               "post_id": "p1",
               "post_title": "Actual old title",
               "post_description": "Old description"
           },
           "type": "ConditionCheckFailed",
           "message": "The condition check failed."
       },
       {
           "type": "None",
           "message": "None"
       }
    ]
}
```

`ctx.error` contient des détails sur l'erreur. Les clés clés et cancellationReasons sont nécessairement présentes dans `ctx.result`.

Système de types (mappage des demandes)

Lors de l'utilisation duAWS AppSyncfonction DynamoDB pour appeler vos tables DynamoDB,AWS AppSyncdoit connaître le type de chaque valeur à utiliser dans cet appel. Cela est dû au fait que DynamoDB prend en charge un plus grand nombre de primitives de type que GraphQL ou JSON (telles que les ensembles et les données binaires).AWS AppSynca besoin de quelques conseils lors de la traduction entre GraphQL et DynamoDB, sinon il devra émettre des hypothèses sur la manière dont les données sont structurées dans votre table.

Pour plus d'informations sur les types de données DynamoDB, consultez leDescripteurs de types de donnéesetTypes de donnéesdocumentation.

Une valeur DynamoDB est représentée par un objet JSON contenant une seule paire clé-valeur. La clé indique le type DynamoDB et la valeur indique la valeur elle-même. Dans l'exemple suivant, la clé `S` indique que la valeur est une chaîne, et la valeur `identifier` est la valeur de chaîne elle-même.

```
{ "S" : "identifier" }
```

Notez que l'objet JSON ne peut pas avoir plus d'une paire clé-valeur. Si plusieurs paires clé-valeur sont spécifiées, l'objet de la requête n'est pas analysé.

Une valeur DynamoDB est utilisée n'importe où dans un objet de requête où vous devez spécifier une valeur. Vous devrez notamment procéder ainsi dans les sections suivantes : `key` et `attributeValue`, ainsi que la section `expressionValues` des sections d'expression. Dans l'exemple suivant, la valeur DynamoDB String`identifier`est affecté au`id`champ dans un`key`section (peut-être dans un`GetItem`objet de la demande).

```
"key" : {
   "id" : { "S" : "identifier" }
}
```

Types pris en charge

AWS AppSyncprend en charge les types de scalaires, de documents et d'ensembles DynamoDB suivants :

Type de chaîne **S**

Valeur de chaîne unique. Une valeur de chaîne DynamoDB est indiquée par :

```
{ "S" : "some string" }
```

Voici un exemple d'utilisation :

```
"key" : {
   "id" : { "S" : "some string" }
}
```

Type d'ensemble de chaîne **SS**

Ensemble de valeurs de chaîne. La valeur d'un ensemble de chaînes DynamoDB est indiquée par :

```
{ "SS" : [ "first value", "second value", ... ] }
```

Voici un exemple d'utilisation :

```
"attributeValues" : {
   "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
```

```
}
```

Type de nombre **N**

Valeur numérique unique. La valeur d'un numéro DynamoDB est indiquée par :

```
{ "N" : 1234 }
```

Voici un exemple d'utilisation :

```
"expressionValues" : {
   ":expectedVersion" : { "N" : 1 }
}
```

Type d'ensemble de nombres **NS**

Ensemble de valeurs de nombres. La valeur d'un ensemble de numéros DynamoDB est indiquée par :

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

Voici un exemple d'utilisation :

```
"attributeValues" : {
   "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }
}
```

Type binaire **B**

Valeur binaire. Une valeur binaire DynamoDB est désignée par :

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

Notez que la valeur est en fait une chaîne, où la chaîne est la représentation codée en base64 des données binaires.AWS AppSyncdécode cette chaîne dans sa valeur binaire avant de l'envoyer à DynamoDB.AWS AppSyncutilise le schéma de décodage base64 tel que défini par la RFC 2045 : tout caractère qui n'est pas dans l'alphabet base64 est ignoré.

Voici un exemple d'utilisation :

```
"attributeValues" : {
```

```
   "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

Type d'ensemble binaire **BS**

Ensemble de valeurs binaires. La valeur d'un ensemble binaire DynamoDB est désignée par :

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

Notez que la valeur est en fait une chaîne, où la chaîne est la représentation codée en base64 des données binaires.AWS AppSyncdécode cette chaîne dans sa valeur binaire avant de l'envoyer à DynamoDB.AWS AppSyncutilise le schéma de décodage base64 tel que défini par la RFC 2045 : tout caractère qui n'est pas dans l'alphabet base64 est ignoré.

Voici un exemple d'utilisation :

```
"attributeValues" : {
   "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }
}
```

Type booléen **BOOL**

Valeur booléenne. Une valeur booléenne DynamoDB est désignée par :

```
{ "BOOL" : true }
```

Notez que seules les valeurs `true` et `false` sont valides.

Voici un exemple d'utilisation :

```
"attributeValues" : {
   "orderComplete" : { "BOOL" : false }
}
```

Type de liste **L**

Liste de toutes les autres valeurs DynamoDB prises en charge. Une valeur de liste DynamoDB est désignée par :

```
{ "L" : [ ... ] }
```

Notez que la valeur est une valeur composée, la liste pouvant contenir zéro ou plus de toute valeur DynamoDB prise en charge (y compris les autres listes). La liste peut également contenir une combinaison de différents types.

Voici un exemple d'utilisation :

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

Type de mappage **M**

Représentant une collection non ordonnée de paires clé-valeur d'autres valeurs DynamoDB prises en charge. La valeur d'une carte DynamoDB est indiquée par :

```
{ "M" : { ... } }
```

Notez qu'un mappage peut contenir zéro ou plusieurs paires clé-valeur. La clé doit être une chaîne, et la valeur peut être n'importe quelle valeur DynamoDB prise en charge (y compris d'autres cartes). Le mappage peut également contenir une combinaison de différents types.

Voici un exemple d'utilisation :

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

Type nul **NULL**

Valeur null. Une valeur DynamoDB Null est désignée par :

```
{ "NULL" : null }
```

Voici un exemple d'utilisation :

```
"attributeValues" : {
   "phoneNumbers" : { "NULL" : null }
}
```

Pour plus d'informations sur chaque type, consultez la documentation DynamoDB.

Système de types (mappage des réponses)

Lorsque vous recevez une réponse de DynamoDB,AWS AppSyncle convertit automatiquement en types primitifs GraphQL et JSON. Chaque attribut de DynamoDB est décodé et renvoyé dans le contexte du gestionnaire de réponses.

Par exemple, si DynamoDB renvoie ce qui suit :

```
{
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

Lorsque le résultat est renvoyé par votre résolveur de pipeline,AWS AppSyncle convertit en types GraphQL et JSON sous la forme :

```
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

Cette section explique commentAWS AppSyncconvertit les types de scalaire, de document et d'ensemble DynamoDB suivants :

Type de chaîne **S**

Valeur de chaîne unique. Une valeur de chaîne DynamoDB est renvoyée sous forme de chaîne.

Par exemple, si DynamoDB a renvoyé la valeur de chaîne DynamoDB suivante :

```
{ "S" : "some string" }
```

AWS AppSyncle convertit en chaîne de caractères :

```
"some string"
```

Type d'ensemble de chaîne **SS**

Ensemble de valeurs de chaîne. Une valeur d'ensemble de chaînes DynamoDB est renvoyée sous forme de liste de chaînes.

Par exemple, si DynamoDB a renvoyé la valeur DynamoDB String Set suivante :

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSyncle convertit en une liste de chaînes :

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

Type de nombre **N**

Valeur numérique unique. Une valeur numérique DynamoDB est renvoyée sous forme de nombre.

Par exemple, si DynamoDB a renvoyé la valeur numérique DynamoDB suivante :

```
{ "N" : 1234 }
```

AWS AppSyncle convertit en nombre :

```
1234
```

Type d'ensemble de nombres **NS**

Ensemble de valeurs de nombres. Une valeur d'ensemble de numéros DynamoDB est renvoyée sous forme de liste de nombres.

Par exemple, si DynamoDB a renvoyé la valeur DynamoDB Number Set suivante :

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSyncle convertit en une liste de nombres :

```
[ 67.8, 12.2, 70 ]
```

Type binaire **B**

Valeur binaire. Une valeur binaire DynamoDB est renvoyée sous forme de chaîne contenant la représentation base64 de cette valeur.

Par exemple, si DynamoDB a renvoyé la valeur binaire DynamoDB suivante :

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSyncle convertit en une chaîne contenant la représentation base64 de la valeur :

```
"SGVsbG8sIFdvcmxkIQo="
```

Notez que les données binaires sont codées selon le schéma Base64 tel que spécifié dans RFC 4648 et RFC 2045.

Type d'ensemble binaire **BS**

Ensemble de valeurs binaires. Une valeur d'ensemble binaire DynamoDB est renvoyée sous forme de liste de chaînes contenant la représentation base64 des valeurs.

Par exemple, si DynamoDB a renvoyé la valeur d'ensemble binaire DynamoDB suivante :

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSyncle convertit en une liste de chaînes contenant la représentation en base64 des valeurs :

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

Notez que les données binaires sont codées selon le schéma Base64 tel que spécifié dans RFC 4648 et RFC 2045.

Type booléen **BOOL**

Valeur booléenne. Une valeur booléenne DynamoDB est renvoyée sous forme de booléen.

Par exemple, si DynamoDB a renvoyé la valeur booléenne DynamoDB suivante :

```
{ "BOOL" : true }
```

AWS AppSyncle convertit en booléen :

```
true
```

Type de liste **L**

Liste de toutes les autres valeurs DynamoDB prises en charge. Une valeur de liste DynamoDB est renvoyée sous forme de liste de valeurs, chaque valeur interne étant également convertie.

Par exemple, si DynamoDB a renvoyé la valeur de liste DynamoDB suivante :

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

AWS AppSyncle convertit en une liste de valeurs converties :

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

Type de mappage **M**

Une collection clé/valeur de toute autre valeur DynamoDB prise en charge. Une valeur de carte DynamoDB est renvoyée sous forme d'objet JSON, où chaque clé/valeur est également convertie.

Par exemple, si DynamoDB a renvoyé la valeur de carte DynamoDB suivante :

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

AWS AppSyncle convertit en objet JSON :

```
{
   "someString" : "A string value",
   "someNumber" : 1,
   "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

Type nul **`NULL`**

Valeur null.

Par exemple, si DynamoDB a renvoyé la valeur Null DynamoDB suivante :

```
{ "NULL" : null }
```

AWS AppSyncle convertit en valeur nulle :

```
null
```

Filtres

Lorsque vous interrogez des objets dans DynamoDB à l'aide du`Query`et`Scan`opérations, vous pouvez éventuellement spécifier un`filter`qui évalue les résultats et renvoie uniquement les valeurs souhaitées.

La propriété de filtre d'un`Query`ou`Scan`La demande a la structure suivante :

```
type DynamoDBExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
};
```

Les champs sont définis comme suit :

`expression`

Expression de la requête. Pour plus d'informations sur la façon d'écrire des expressions de filtre, consultez leDynamoDBQueryFilteretDynamoDBScanFilterdocumentation. Ce champ doit être spécifié.

`expressionNames`

Substituts des espaces réservés de nom des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé de nom utilisé dans le `expression`. La valeur doit être une chaîne correspondant au nom d'attribut de l'élément dans DynamoDB. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de nom des attributs de l'expression utilisés dans l'`expression`.

`expressionValues`

Substituts des espaces réservés de valeur des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé de valeur utilisé dans l'`expression`, et la valeur doit être typée. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Cela doit être spécifié. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de valeur des attributs de l'expression utilisés dans l'`expression`.

Exemple

L'exemple suivant est une section de filtre pour une demande, dans laquelle les entrées extraites de DynamoDB ne sont renvoyées que si le titre commence par le`title`argument.

Ici, nous utilisons le`util.transform.toDynamoDBFilterExpression`pour créer automatiquement un filtre à partir d'un objet :

```
const filter = util.transform.toDynamoDBFilterExpression({
  title: { beginsWith: 'far away' },
});

const request = {};
request.filter = JSON.parse(filter);
```

Cela génère le filtre suivant :

```
{
  "filter": {
    "expression": "(begins_with(#title,:title_beginsWith))",
    "expressionNames": { "#title": "title" },
    "expressionValues": {
      ":title_beginsWith": { "S": "far away" }
    }
```

```
  }
}
```

Expressions de condition

Lorsque vous mutez des objets dans DynamoDB à l'aide du`PutItem`,`UpdateItem`, et`DeleteItem`Opérations DynamoDB : vous pouvez éventuellement spécifier une expression de condition qui détermine si la demande doit aboutir ou non, en fonction de l'état de l'objet déjà présent dans DynamoDB avant l'exécution de l'opération.

LeAWS AppSyncLa fonction DynamoDB permet de spécifier une expression de condition dans`PutItem`,`UpdateItem`, et`DeleteItem`objets de demande, ainsi qu'une stratégie à suivre si la condition échoue et que l'objet n'a pas été mis à jour.

Exemple 1

Ce qui suit`PutItem`l'objet de demande n'a pas d'expression de condition. Par conséquent, il place un élément dans DynamoDB même s'il existe déjà un élément portant la même clé, remplaçant ainsi l'élément existant.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, ...values} = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}
```

Exemple 2

Ce qui suit`PutItem`l'objet possède une expression de condition qui permet à l'opération de réussir uniquement si un élément avec la même clépasexistent dans DynamoDB.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, ...values} = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
```

```
    attributeValues: util.dynamodb.toMapValues(values),
    condition: { expression: "attribute_not_exists(id)" }
  };
}
```

Par défaut, en cas d'échec de la vérification de condition,AWS AppSyncLa fonction DynamoDB fournit une erreur dans`ctx.error`. Vous pouvez renvoyer l'erreur associée à la mutation et la valeur actuelle de l'objet dans DynamoDB dans un`data`champ dans le`error`section de la réponse GraphQL.

Cependant, leAWS AppSyncLa fonction DynamoDB offre des fonctionnalités supplémentaires pour aider les développeurs à gérer certains cas extrêmes courants :

- SiAWS AppSyncLes fonctions DynamoDB peuvent déterminer que la valeur actuelle dans DynamoDB correspond au résultat souhaité, elles traitent l'opération comme si elle avait réussi de toute façon.
- Au lieu de renvoyer une erreur, vous pouvez configurer la fonction pour appeler une fonction Lambda personnalisée afin de décider commentAWS AppSyncLa fonction DynamoDB devrait gérer l'échec.

Ils sont décrits plus en détail dans leGestion d'un échec de vérification d'étatsection.

Pour plus d'informations sur les expressions de conditions DynamoDB, consultez leDynamoDBConditionExpressionsdocumentation.

Spécifier une condition

Le`PutItem`,`UpdateItem`, et`DeleteItem`les objets de requête autorisent tous une option`condition`section à préciser. Si cette section est omise, aucune vérification de condition n'est effectuée. Si elle est spécifiée, la condition doit être true pour que l'opération réussisse.

Une section `condition` a la structure suivante :

```
type ConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
  equalsIgnore?: string[];
  consistentRead?: boolean;
  conditionalCheckFailedHandler?: {
```

```
    strategy: 'Custom' | 'Reject';
    lambdaArn?: string;
  };
};
```

Les champs suivants spécifient la condition :

`expression`

Expression de mise à jour elle-même. Pour plus d'informations sur la façon d'écrire des expressions de condition, consultez leDynamoDBConditionExpressionsdocumentation. Ce champ doit être spécifié.

`expressionNames`

Substituts des espaces réservés de nom des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé au nom utilisé dansexpression, et la valeur doit être une chaîne correspondant au nom d'attribut de l'élément dans DynamoDB. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de nom des attributs de l'expression utilisés dans l'expression.

`expressionValues`

Substituts des espaces réservés de valeur des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé de valeur utilisé dans l'expression, et la valeur doit être typée. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Cela doit être spécifié. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de valeur des attributs de l'expression utilisés dans l'expression.

Les champs restants indiquentAWS AppSyncFonction DynamoDB pour gérer un échec de vérification de condition :

`equalsIgnore`

Lorsqu'un contrôle d'état échoue lors de l'utilisation du`PutItem`opération, leAWS AppSyncLa fonction DynamoDB compare l'élément actuellement dans DynamoDB à l'élément qu'il a essayé d'écrire. S'ils sont identiques, il traite l'opération comme si elle avait réussi. Vous pouvez utiliser le`equalsIgnore`champ pour spécifier une liste d'attributs quiAWS AppSyncdoit être ignoré lors de cette comparaison. Par exemple, si la seule différence était`version`attribut, il traite l'opération comme si elle avait réussi. Ce champ est facultatif.

`consistentRead`

Lorsqu'un contrôle de condition échoue,AWS AppSyncobtient la valeur actuelle de l'élément depuis DynamoDB en utilisant une lecture très cohérente. Vous pouvez utiliser ce champ pour indiquerAWS AppSyncFonction DynamoDB pour utiliser une lecture finalement cohérente à la place. Ce champ est facultatif et contient `true` par défaut.

`conditionalCheckFailedHandler`

Cette section vous permet de définir la manière dontAWS AppSyncLa fonction DynamoDB traite un échec de vérification de condition après avoir comparé la valeur actuelle dans DynamoDB au résultat attendu. Cette section est facultative. Si elle n'est pas spécifiée, la valeur par défaut est une stratégie `Reject`.

`strategy`

La stratégie, leAWS AppSyncLa fonction DynamoDB prend une fois qu'elle a comparé la valeur actuelle dans DynamoDB au résultat attendu. Ce champ est obligatoire et les valeurs suivantes sont possibles :

`Reject`

La mutation échoue, et une erreur se produit pour la mutation et la valeur actuelle de l'objet dans DynamoDB dans un`data`champ dans le`error`section de la réponse GraphQL.

`Custom`

LeAWS AppSyncLa fonction DynamoDB invoque une fonction Lambda personnalisée pour décider de la manière de gérer l'échec du contrôle de condition. Lorsque la `strategy` est définie sur `Custom`, le champ `lambdaArn` doit contenir l'ARN de la fonction Lambda à appeler.

`lambdaArn`

L'ARN de la fonction Lambda à invoquer qui détermine commentAWS AppSyncLa fonction DynamoDB doit gérer l'échec de la vérification de condition. Ce champ doit être spécifié uniquement lorsque `strategy` est défini sur `Custom`. Pour plus d'informations sur l'utilisation de cette fonctionnalité, voirGestion d'un échec de vérification d'état.

Gestion d'un échec de vérification d'état

Lorsqu'un contrôle d'état échoue, leAWS AppSyncLa fonction DynamoDB peut transmettre l'erreur liée à la mutation et la valeur actuelle de l'objet en utilisant`util.appendError`utilité. Cela ajoute

ledatachamp dans leerrorsection de la réponse GraphQL. Cependant, leAWS AppSyncLa fonction DynamoDB offre des fonctionnalités supplémentaires pour aider les développeurs à gérer certains cas extrêmes courants :

- SiAWS AppSyncLes fonctions DynamoDB peuvent déterminer que la valeur actuelle dans DynamoDB correspond au résultat souhaité, elles traitent l'opération comme si elle avait réussi de toute façon.
- Au lieu de renvoyer une erreur, vous pouvez configurer la fonction pour appeler une fonction Lambda personnalisée afin de décider commentAWS AppSyncLa fonction DynamoDB devrait gérer l'échec.

Le diagramme de ce processus est le suivant :

Vérification du résultat souhaité

Lorsque le contrôle de l'état échoue,AWS AppSyncLa fonction DynamoDB exécute uneGetItemDemande DynamoDB pour obtenir la valeur actuelle de l'élément auprès de DynamoDB. Par défaut, il utilise une lecture à cohérence forte, mais cela peut être configuré à l'aide du champ `consistentRead` dans le bloc `condition` et comparé aux résultats prévus :

- Pour lePutItemopération, leAWS AppSyncLa fonction DynamoDB compare la valeur actuelle à celle qu'elle a tenté d'écrire, en excluant les attributs répertoriés dansequalsIgnoreà partir de la comparaison. Si les éléments sont identiques, il considère l'opération comme réussie et renvoie l'élément extrait de DynamoDB. Dans le cas contraire, il suit la stratégie configurée.

  Par exemple, si lePutItem'objet de la demande ressemblait à ce qui suit :

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { id, name, version} = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues({ name, version: version+1 }),
    condition: {
      expression: "version = :expectedVersion",
      expressionValues: util.dynamodb.toMapValues({':expectedVersion': version}),
      equalsIgnore: ['version']
    }
```

```
  };
}
```

Et si l'élément actuellement dans DynamoDB ressemble à ce qui suit :

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

LeAWS AppSyncLa fonction DynamoDB comparait l'élément qu'elle a essayé d'écrire à la valeur actuelle, en vérifiant que la seule différence était`version`, mais parce qu'il est configuré pour ignorer le`version`champ, il considère l'opération comme réussie et renvoie l'élément récupéré depuis DynamoDB.

- Pour le`DeleteItem`opération, leAWS AppSyncLa fonction DynamoDB vérifie qu'un élément a été renvoyé par DynamoDB. Si aucun élément n'a été renvoyé, il traite l'opération comme réussie. Dans le cas contraire, il suit la stratégie configurée.
- Pour le`UpdateItem`opération, leAWS AppSyncLa fonction DynamoDB ne dispose pas de suffisamment d'informations pour déterminer si l'élément actuellement dans DynamoDB correspond au résultat attendu et suit donc la stratégie configurée.

Si l'état actuel de l'objet dans DynamoDB est différent du résultat attendu,AWS AppSyncLa fonction DynamoDB suit la stratégie configurée, soit pour rejeter la mutation, soit pour invoquer une fonction Lambda pour déterminer la marche à suivre.

Suivre la stratégie du « rejet »

Lorsque vous suivez le`Reject`stratégie, leAWS AppSyncLa fonction DynamoDB renvoie une erreur pour la mutation, et la valeur actuelle de l'objet dans DynamoDB est également renvoyée dans un`data`champ dans le`error`section de la réponse GraphQL. L'élément renvoyé par DynamoDB est soumis au gestionnaire de réponse aux fonctions pour le traduire dans le format attendu par le client, puis il est filtré par le jeu de sélection.

Par exemple, si nous avons la demande de mutation suivante :

```
mutation {
    updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
        Name
```

```
        theVersion
    }
}
```

Si l'article renvoyé par DynamoDB ressemble à ce qui suit :

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

Et le gestionnaire de réponse aux fonctions ressemble à ce qui suit :

```
import { util } from '@aws-appsync/utils';
export function response(ctx) {
  const { version, ...values } = ctx.result;
  const result = { ...values, theVersion: version };
  if (ctx.error) {
    if (error) {
      return util.appendError(error.message, error.type, result, null);
    }
  }
  return result
}
```

La réponse GraphQL se présente comme suit :

```
{
  "data": null,
  "errors": [
    {
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "data": {
        "Name": "Steve",
        "theVersion": 8
      },
      ...
    }
```

```
  ]
}
```

Notez également que si des champs de l'objet renvoyé sont remplis par d'autres résolveurs et que la mutation a réussi, ils ne seront pas résolus lorsque l'objet sera renvoyé dans la section `error`.

Suivre la stratégie « personnalisée »

Lorsque vous suivez le`Custom`stratégie, leAWS AppSyncLa fonction DynamoDB invoque une fonction Lambda pour décider de la marche à suivre. La fonction Lambda choisit l'une des options suivantes :

- `reject` la mutation. Cela indique queAWS AppSyncFonction DynamoDB pour se comporter comme si la stratégie configurée était`Reject`, renvoyant une erreur concernant la mutation et la valeur actuelle de l'objet dans DynamoDB, comme décrit dans la section précédente.
- `discard` la mutation. Cela indique queAWS AppSyncFonction DynamoDB permettant d'ignorer silencieusement l'échec de la vérification des conditions et de renvoyer la valeur dans DynamoDB.
- `retry` la mutation. Cela indique queAWS AppSyncFonction DynamoDB pour réessayer la mutation avec un nouvel objet de requête.

Requête d'appel Lambda

LeAWS AppSyncLa fonction DynamoDB invoque la fonction Lambda spécifiée dans le`lambdaArn`. Il utilise le même `service-role-arn` que celui configuré sur la source de données. La charge utile de l'appel a la structure suivante :

```
{
    "arguments": { ... },
    "requestMapping": {... },
    "currentValue": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

Les champs sont définis comme suit :

`arguments`

Arguments de la mutation GraphQL. C'est la même chose que les arguments disponibles pour l'objet de requête dans`context.arguments`.

`requestMapping`

L'objet de la demande pour cette opération.

`currentValue`

La valeur actuelle de l'objet dans DynamoDB.

`resolver`

Informations sur leAWS AppSyncrésolveur ou fonction.

`identity`

Informations sur l'appelant. Il s'agit de la même chose que les informations d'identité disponibles pour l'objet de la demande dans`context.identity`.

Exemple complet de la charge utile :

```
{
    "arguments": {
        "id": "1",
        "name": "Steve",
        "expectedVersion": 1
    },
    "requestMapping": {
        "version" : "2017-02-28",
        "operation" : "PutItem",
        "key" : {
           "id" : { "S" : "1" }
        },
        "attributeValues" : {
           "name" : { "S" : "Steve" },
           "version" : { "N" : 2 }
        },
        "condition" : {
           "expression" : "version = :expectedVersion",
           "expressionValues" : {
               ":expectedVersion" : { "N" : 1 }
           },
           "equalsIgnore": [ "version" ]
        }
    },
    "currentValue": {
        "id" : { "S" : "1" },
```

```
        "name" : { "S" : "Steve" },
        "version" : { "N" : 8 }
    },
    "resolver": {
        "tableName": "People",
        "awsRegion": "us-west-2",
        "parentType": "Mutation",
        "field": "updatePerson",
        "outputType": "Person"
    },
    "identity": {
        "accountId": "123456789012",
        "sourceIp": "x.x.x.x",
        "user": "AIDAAAAAAAAAAAAAAAAA",
        "userArn": "arn:aws:iam::123456789012:user/appsync"
    }
}
```

Réponse à l'appel de Lambda

La fonction Lambda peut inspecter la charge utile d'invocation et appliquer n'importe quelle logique métier pour déterminer commentAWS AppSyncLa fonction DynamoDB devrait gérer l'échec. Il existe trois options pour gérer l'échec de vérification de la condition :

- `reject` la mutation. La charge utile de la réponse pour cette option doit avoir cette structure :

  ```
  {
      "action": "reject"
  }
  ```

  Cela indique queAWS AppSyncFonction DynamoDB pour se comporter comme si la stratégie configurée étaitReject, renvoyant une erreur pour la mutation et la valeur actuelle de l'objet dans DynamoDB, comme décrit dans la section ci-dessus.
- `discard` la mutation. La charge utile de la réponse pour cette option doit avoir cette structure :

  ```
  {
      "action": "discard"
  }
  ```

  Cela indique queAWS AppSyncFonction DynamoDB permettant d'ignorer silencieusement l'échec de la vérification des conditions et de renvoyer la valeur dans DynamoDB.

- `retry` la mutation. La charge utile de la réponse pour cette option doit avoir cette structure :

```
{
    "action": "retry",
    "retryMapping": { ... }
}
```

Cela indique queAWS AppSyncFonction DynamoDB pour réessayer la mutation avec un nouvel objet de requête. La structure du`retryMapping`la section dépend de l'opération DynamoDB et constitue un sous-ensemble de l'objet de demande complet pour cette opération.

Pour `PutItem`, la section `retryMapping` a la structure suivante. Pour une description du`attributeValues`terrain, voirPutItem.

```
{
    "attributeValues": { ... },
    "condition": {
        "equalsIgnore" = [ ... ],
        "consistentRead" = true
    }
}
```

Pour `UpdateItem`, la section `retryMapping` a la structure suivante. Pour une description du`update`section, voirUpdateItem.

```
{
    "update" : {
        "expression" : "someExpression"
        "expressionNames" : {
            "#foo" : "foo"
        },
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "condition": {
        "consistentRead" = true
    }
}
```

Pour `DeleteItem`, la section `retryMapping` a la structure suivante.

```
{
    "condition": {
        "consistentRead" = true
    }
}
```

Il n'y a aucun moyen de spécifier une autre opération ou une autre clé sur laquelle travailler. LeAWS AppSyncLa fonction DynamoDB autorise uniquement les tentatives de la même opération sur le même objet. D'autre part, la section `condition` ne permet pas de spécifier un `conditionalCheckFailedHandler`. Si la nouvelle tentative échoue,AWS AppSyncLa fonction DynamoDB suit le`Reject`stratégie.

Voici un exemple de fonction Lambda pour traiter une demande `PutItem` qui a échoué. La logique métier s'adresse à celui qui effectue l'appel. S'il a été fabriqué par`jeffTheAdmin`, il réessaie la demande en mettant à jour le`version`et`expectedVersion`à partir de l'élément actuellement dans DynamoDB. Dans le cas contraire, il rejette la mutation.

```
exports.handler = (event, context, callback) => {
    console.log("Event: "+ JSON.stringify(event));

    // Business logic goes here.

    var response;
    if ( event.identity.user == "jeffTheAdmin" ) {
        response = {
            "action" : "retry",
            "retryMapping" : {
                "attributeValues" : event.requestMapping.attributeValues,
                "condition" : {
                    "expression" : event.requestMapping.condition.expression,
                    "expressionValues" :
 event.requestMapping.condition.expressionValues
                }
            }
        }
        response.retryMapping.attributeValues.version = { "N" :
 event.currentValue.version.N + 1 }
```

```
        response.retryMapping.condition.expressionValues[':expectedVersion'] =
 event.currentValue.version

    } else {
        response = { "action" : "reject" }
    }

    console.log("Response: "+ JSON.stringify(response))
    callback(null, response)
};
```

Expressions des conditions de transaction

Les expressions des conditions de transaction sont disponibles dans les demandes des quatre types d'opérations dans`TransactWriteItems`, à savoir`PutItem`,`DeleteItem`,`UpdateItem`, et`ConditionCheck`.

Pour`PutItem`,`DeleteItem`, et`UpdateItem`, l'expression de condition de transaction est facultative. Pour`ConditionCheck`, l'expression de la condition de transaction est obligatoire.

Exemple 1

Le transactionnel suivant`DeleteItem`le gestionnaire de demandes de fonction n'a pas d'expression de condition. Par conséquent, il supprime l'élément dans DynamoDB.

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { postId } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'DeleteItem',
        key: util.dynamodb.toMapValues({ postId }),
      }
    ],
  };
}
```

Exemple 2

Le transactionnel suivant`DeleteItem`le gestionnaire de demandes de fonction possède une expression de condition de transaction qui permet à l'opération de réussir uniquement si l'auteur de cette publication porte un certain nom.

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { postId, authorName} = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'DeleteItem',
        key: util.dynamodb.toMapValues({ postId }),
        condition: util.transform.toDynamoDBConditionExpression({
          authorName: { eq: authorName },
        }),
      }
    ],
  };
}
```

Si la vérification de condition échoue, elle provoque `TransactionCanceledException` et les détails de l'erreur sont renvoyés dans `ctx.result.cancellationReasons`. Notez que par défaut, l'ancien élément de DynamoDB à l'origine de l'échec de la vérification d'état sera renvoyé dans`ctx.result.cancellationReasons`.

Spécifier une condition

Le`PutItem`,`UpdateItem`, et`DeleteItem`les objets de requête autorisent tous une option`condition`section à préciser. Si cette section est omise, aucune vérification de condition n'est effectuée. Si elle est spécifiée, la condition doit être true pour que l'opération réussisse. Le `ConditionCheck` doit avoir une section `condition` à spécifier. La condition doit être vraie pour que l'ensemble de la transaction réussisse.

Une section `condition` a la structure suivante :

```
type TransactConditionCheckExpression = {
```

```
  expression: string;
  expressionNames?: { [key: string]: string };
  expressionValues?: { [key: string]: string };
  returnValuesOnConditionCheckFailure: boolean;
};
```

Les champs suivants spécifient la condition :

`expression`

Expression de mise à jour elle-même. Pour plus d'informations sur la façon d'écrire des expressions de condition, consultez leDynamoDBConditionExpressionsdocumentation. Ce champ doit être spécifié.

`expressionNames`

Substituts des espaces réservés de nom des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé au nom utilisé dansexpression, et la valeur doit être une chaîne correspondant au nom d'attribut de l'élément dans DynamoDB. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de nom des attributs de l'expression utilisés dans l'expression.

`expressionValues`

Substituts des espaces réservés de valeur des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé de valeur utilisé dans l'expression, et la valeur doit être typée. Pour plus d'informations sur la manière de spécifier une « valeur saisie », voirSystème de types (mappage des demandes). Cela doit être spécifié. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de valeur des attributs de l'expression utilisés dans l'expression.

`returnValuesOnConditionCheckFailure`

Spécifiez s'il faut récupérer l'élément dans DynamoDB en cas d'échec d'une vérification de condition. L'élément récupéré est dans `ctx.result.cancellationReasons[<index>].item`, où `<index>` est l'index de l'élément de demande qui a échoué à la vérification de condition. Cette valeur est définie par défaut sur true.

Projections

Lorsque vous lisez des objets dans DynamoDB à l'aide du`GetItem`,`Scan`,`Query`,`BatchGetItem`, et`TransactGetItems`opérations, vous pouvez éventuellement spécifier une projection qui identifie les attributs souhaités. La structure de la propriété de projection est la suivante, similaire à celle des filtres :

```
type DynamoDBExpression = {
  expression: string;
  expressionNames?: { [key: string]: string}
};
```

Les champs sont définis comme suit :

`expression`

L'expression de projection, qui est une chaîne. Pour récupérer un seul attribut, spécifiez son nom. Pour les attributs multiples, les noms doivent être des valeurs séparées par des virgules. Pour plus d'informations sur l'écriture d'expressions de projection, consultez leExpressions de projection DynamoDBdocumentation. Ce champ est obligatoire.

`expressionNames`

Les substitutions pour l'attribut d'expressionnomdes espaces réservés sous forme de paires clé-valeur. La clé correspond à un espace réservé de nom utilisé dans le `expression`. La valeur doit être une chaîne correspondant au nom d'attribut de l'élément dans DynamoDB. Ce champ est facultatif et ne doit être rempli qu'avec des substitutions pour les espaces réservés aux noms d'attributs d'expression utilisés dans`expression`. Pour plus d'informations sur`expressionNames`, consultez leDocumentation DynamoDB.

Exemple 1

L'exemple suivant est une section de projection pourJavaScriptfonction dans laquelle seuls les attributs`author`et`id`sont renvoyés par DynamoDB :

```
projection : {
    expression : "#author, id",
    expressionNames : {
        "#author" : "author"
```

```
    }
}
```

> **Tip**
>
> Vous pouvez accéder à votre ensemble de sélection de requêtes GraphQL en utilisantselectionSetList. Ce champ vous permet de cadrer votre expression de projection de manière dynamique en fonction de vos besoins.

> **Note**
>
> Lors de l'utilisation d'expressions de projection avecQueryetScanopérations, la valeur deselectdoit êtreSPECIFIC_ATTRIBUTES. Pour plus d'informations, consultez leDocumentation DynamoDB.

JavaScript référence de la fonction de résolution pour OpenSearch

LeAWS AppSync résolveur pour Amazon OpenSearch Service vous permet d'utiliser GraphQL pour stocker et récupérer des données dans les domaines OpenSearch Service existants de votre compte. Ce résolveur fonctionne en vous permettant de mapper une requête GraphQL entrante en une demande de OpenSearch service, puis de mapper la réponse du OpenSearch service à GraphQL. Cette section décrit les gestionnaires de demandes et de réponses de fonctions pour les opérations de OpenSearch service prises en charge.

Requête

La plupart des objets de demande de OpenSearch service ont une structure commune dans laquelle seuls quelques éléments changent. L'exemple suivant exécute une recherche sur un domaine de OpenSearch service, dans lequel les documents sont de typepost et sont indexésid. Les paramètres de recherche sont définis dans la section body, avec un grand nombre de clauses de requête courants définies dans le champ query. Cet exemple recherche des documents contenant "Nadia" ou "Bailey", ou les deux, dans le champ author d'un document :

```
export function request(ctx) {
  return {
```

```
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: {
        from: 0,
        size: 50,
        query: {
          bool: {
            should: [
              { match: { author: 'Nadia' } },
              { match: { author: 'Bailey' } },
            ],
          },
        },
      },
    },
  };
}
```

Réponse

Comme pour les autres sources de données, OpenSearch Service envoie une réponseAWS AppSync qui doit être convertie en GraphQL.

La plupart des requêtes GraphQL recherchent le`_source` champ à partir d'une réponse de OpenSearch service. Comme vous pouvez effectuer des recherches pour renvoyer un document individuel ou une liste de documents, deux modèles de réponse courants sont utilisés dans OpenSearch Service :

Liste des résultats

```
export function response(ctx) {
  const entries = [];
  for (const entry of ctx.result.hits.hits) {
    entries.push(entry['_source']);
  }
  return entries;
}
```

Élément individuel

```
export function response(ctx) {
  return ctx.result['_source']
}
```

operation field

(gestionnaire de requêtes uniquement)

Méthode ou verbe HTTP (GET, POST, PUT, HEAD ou DELETE) quiAWS AppSync envoie au domaine OpenSearch Service. La clé et la valeur doivent être une chaîne.

```
"operation" : "PUT"
```

path field

(gestionnaire de requêtes uniquement)

Le chemin de recherche d'une demande OpenSearch de service émanant deAWS AppSync. Cela forme une URL pour le verbe HTTP de l'opération. La clé et la valeur doivent être des chaînes.

```
"path" : "/indexname/type"

"path" : "/indexname/type/_search"
```

Lorsque le gestionnaire de requêtes est évalué, ce chemin est envoyé dans le cadre de la requête HTTP, y compris le domaine OpenSearch de service. Ainsi, l'exemple précédent pourrait donner :

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

params field

(gestionnaire de requêtes uniquement)

Permet de spécifier l'action exécutée par votre recherche, plus couramment en définissant la valeur de requête à l'intérieur du corps. Toutefois, il existe plusieurs autres capacités qui peuvent être configurées, comme la mise en forme des réponses.

- headers

Informations d'en-tête, sous forme de paires clé-valeur. La clé et la valeur doivent être des chaînes. Par exemple :

```
"headers" : {
    "Content-Type" : "application/json"
}
```

> **Note**
>
> AWS AppSync ne prend actuellement en charge que le JSON en tant que`Content-Type`.

- queryString

  Paires clé-valeur qui spécifient les options courantes, telles que la mise en forme du code pour les réponses JSON. La clé et la valeur doivent être une chaîne. Par exemple, si vous souhaitez obtenir un JSON mis en forme correctement, vous devez utiliser :

```
"queryString" : {
    "pretty" : "true"
}
```

- body

  Il s'agit de la partie principale de votre demande, qui permetAWS AppSync de créer une demande de recherche bien formée pour votre domaine OpenSearch de service. La clé doit être une chaîne composée d'un objet. Voici quelques démonstrations.

Exemple 1

Renvoyer tous les documents avec une ville correspondant à « seattle » :

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: { from: 0, size: 50, query: { match: { city: 'seattle' } } },
```

```
    },
  };
}
```

Exemple 2

Renvoyer tous les documents correspondant à « washington » comme ville ou État :

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: {
        from: 0,
        size: 50,
        query: {
          multi_match: { query: 'washington', fields: ['city', 'state'] },
        },
      },
    },
  };
}
```

Transmission de variables

(gestionnaire de requêtes uniquement)

Vous pouvez également transmettre des variables dans le cadre de l'évaluation dans votre gestionnaire de requêtes. Par exemple, supposons que vous ayez une requête GraphQL similaire à la suivante :

```
query {
    searchForState(state: "washington"){
        ...
    }
}
```

Le gestionnaire de demandes de fonction peut être le suivant :

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: {
        from: 0,
        size: 50,
        query: {
          multi_match: { query: ctx.args.state, fields: ['city', 'state'] },
        },
      },
    },
  };
}
```

JavaScript référence de la fonction de résolution pour Lambda

Vous pouvez utiliser des AWS AppSync fonctions et des résolveurs pour appeler les fonctions Lambda situées dans votre compte. Vous pouvez façonner les charges utiles de vos demandes et la réponse de vos fonctions Lambda avant de les renvoyer à vos clients. Vous pouvez également spécifier le type d'opération à effectuer dans votre objet de demande. Cette section décrit les demandes relatives aux opérations Lambda prises en charge.

Objet Requête

L'objet de requête Lambda gère les champs liés à votre fonction Lambda :

```
export type LambdaRequest = {
  operation: 'Invoke' | 'BatchInvoke';
  invocationType?: 'RequestResponse' | 'Event';
  payload: unknown;
};
```

Voici un exemple qui utilise une `invoke` opération dont les données de charge utile sont le `getPost` champ d'un schéma GraphQL avec ses arguments issus du contexte :

```
export function request(ctx) {
  return {
```

```
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

L'intégralité du document de mappage est transmise en entrée à votre fonction Lambda, de sorte que l'exemple précédent ressemble désormais à ceci :

```
{
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "input": {
        "id": "postId1",
      }
    }
  }
}
```

Opération

La source de données Lambda vous permet de définir deux opérations `operation` sur le terrain : `Invoke` et. `BatchInvoke` L'`Invoke`opération permet de AWS AppSync savoir d'appeler votre fonction Lambda pour chaque résolveur de champs GraphQL. `BatchInvoke`indique de regrouper AWS AppSync les requêtes pour le champ GraphQL actuel. Le champ `operation` est obligatoire.

En effet`Invoke`, la demande résolue correspond à la charge utile d'entrée de la fonction Lambda. Modifions l'exemple ci-dessus :

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

Ceci est résolu et transmis à la fonction Lambda, qui pourrait ressembler à ceci :

```
{
  "operation": "Invoke",
```

```
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

En `BatchInvoke` effet, la demande est appliquée à chaque résolveur de champs du lot. Par souci de concision, AWS AppSync fusionne toutes les `payload` valeurs de demande dans une liste sous un seul objet correspondant à l'objet de la demande. L'exemple de gestionnaire de demandes suivant illustre la fusion :

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: ctx,
  };
}
```

Cette demande est évaluée et résolue dans le document de mappage suivant :

```
{
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

Chaque élément de la `payload` liste correspond à un seul article du lot. La fonction Lambda devrait également renvoyer une réponse sous forme de liste correspondant à l'ordre des éléments envoyés dans la demande :

```
[
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
 1
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
 2
  { "data": {...}, "errorMessage": null, "errorType": null }  // result for batch item
 3
```

```
]
```

Charge utile

Le `payload` champ est un conteneur utilisé pour transmettre des données à la fonction Lambda. Si le `operation` champ est défini sur`BatchInvoke`, AWS AppSync regroupe les `payload` valeurs existantes dans une liste. Le champ `payload` est facultatif.

Type d'invocation

La source de données Lambda vous permet de définir deux types d'invocation : et. `RequestResponse` `Event` Les types d'invocation sont synonymes des types d'invocation définis dans l'API Lambda. Le type `RequestResponse` d'invocation permet d' AWS AppSync appeler votre fonction Lambda de manière synchrone pour attendre une réponse. L'`Event`invocation vous permet d'appeler votre fonction Lambda de manière asynchrone. Pour plus d'informations sur la façon dont Lambda gère les demandes de type `Event` invocation, consultez Invocation asynchrone. Le champ `invocationType` est facultatif. Si ce champ n'est pas inclus dans la demande, le type d'`RequestResponse`appel AWS AppSync sera défini par défaut.

Quel que soit `invocationType` le champ, la demande résolue correspond à la charge utile d'entrée de la fonction Lambda. Modifions l'exemple ci-dessus :

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    invocationType: 'Event',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

Ceci est résolu et transmis à la fonction Lambda, qui pourrait ressembler à ceci :

```
{
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
```

```
}
```

Lorsque l'`BatchInvoke`opération est utilisée conjointement avec le champ de type d'`Event`invocation, elle AWS AppSync fusionne le résolveur de champ de la même manière que celle mentionnée ci-dessus, et la demande est transmise à votre fonction Lambda sous forme d'événement asynchrone sous la forme d'une liste de valeurs. `payload` La réponse d'une demande de type `Event` invocation aboutit à une `null` valeur sans gestionnaire de réponse :

```
{
  "data": {
    "field": null
  }
}
```

Nous vous recommandons de désactiver la mise en cache des résolveurs pour les résolveurs de type `Event` invocation, car ceux-ci ne seraient pas envoyés à Lambda en cas d'accès au cache.

Objet Réponse

Comme pour les autres sources de données, votre fonction Lambda envoie une réponse AWS AppSync qui doit être convertie en un type GraphQL. Le résultat de la fonction Lambda est contenu dans la propriété de `context` résultat ()`context.result`.

Si la forme de la réponse de votre fonction Lambda correspond à la forme du type GraphQL, vous pouvez transférer la réponse à l'aide du gestionnaire de réponse de fonction suivant :

```
export function response(ctx) {
  return ctx.result
}
```

Aucun champ obligatoire ni aucune restriction de forme ne s'appliquent à l'objet de réponse. Cependant, comme GraphQL est fortement typé, la réponse résolue doit correspondre au type GraphQL attendu.

Réponse par lots de la fonction Lambda

Si le `operation` champ est défini sur`BatchInvoke`, AWS AppSync attend une liste d'éléments en retour de la fonction Lambda. Afin de AWS AppSync faire correspondre chaque résultat à l'élément de demande d'origine, la liste de réponses doit correspondre en taille et en ordre. Il est valide d'avoir des `null` éléments dans la liste de réponses ; elle `ctx.result` est définie sur null en conséquence.

JavaScript référence de fonction de résolution pour la source de EventBridge données

La demande et AWS AppSync la réponse de la fonction de résolution utilisées avec la source de EventBridge données vous permettent d'envoyer des événements personnalisés au EventBridge bus Amazon.

Demande

Le gestionnaire de demandes vous permet d'envoyer plusieurs événements personnalisés à un bus d' EventBridge événements :

```
export function request(ctx) {
  return {
    "operation" : "PutEvents",
    "events" : [{}]
  }
}
```

La définition de type d'une EventBridge `PutEvents` demande est la suivante :

```
type PutEventsRequest = {
  operation: 'PutEvents'
  events: {
    source: string
    detail: { [key: string]: any }
    detailType: string
    resources?: string[]
    time?: string // RFC3339 Timestamp format
  }[]
}
```

Réponse

Si l'`PutEvents`opération est réussie, la réponse du formulaire EventBridge est incluse dans le `ctx.result` :

```
export function response(ctx) {
  if(ctx.error)
```

```
    util.error(ctx.error.message, ctx.error.type, ctx.result)
  else
    return ctx.result
}
```

Les erreurs qui se produisent lors de l'exécution d'`PutEvents`opérations telles que `InternalExceptions` ou `Timeouts` apparaîtront dans`ctx.error`. Pour obtenir la liste EventBridge des erreurs courantes, consultez la référence des erreurs EventBridge courantes.

Ils `result` auront la définition de type suivante :

```
type PutEventsResult = {
  Entries: {
    ErrorCode: string
    ErrorMessage: string
    EventId: string
  }[]
  FailedEntryCount: number
}
```

- Entrées

  L'événement ingéré entraîne à la fois un succès et un échec. Si l'ingestion a réussi, l'entrée `EventID` en contient. Sinon, vous pouvez utiliser le `ErrorCode` et `ErrorMessage` pour identifier le problème lié à l'entrée.

  Pour chaque enregistrement, l'index de l'élément de réponse est le même que celui du tableau de requêtes.
- FailedEntryCount

  Le nombre d'entrées ayant échoué. Cette valeur est représentée sous la forme d'un entier.

Pour plus d'informations sur la réponse de`PutEvents`, voir PutEvents.

Exemple de réponse 1

L'exemple suivant est une `PutEvents` opération avec deux événements réussis :

```
{
    "Entries" : [
        {
```

```
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ],
    "FailedEntryCount" : 0
}
```

Exemple de réponse 2

L'exemple suivant est une `PutEvents` opération comportant trois événements, deux réussites et un échec :

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        },
        {
            "ErrorCode" : "SampleErrorCode",
            "ErrorMessage" : "Sample Error Message"
        }
    ],
    "FailedEntryCount" : 1
}
```

`PutEvents` field

- Version

  Commun à tous les modèles de mappage de demandes, le `version` champ définit la version utilisée par le modèle. Ce champ est obligatoire. La valeur `2018-05-29` est la seule version prise en charge pour les modèles de EventBridge mappage.

- Opération

  La seule opération prise en charge est`PutEvents`. Cette opération vous permet d'ajouter des événements personnalisés à votre bus d'événements.

- Evénements

  Un ensemble d'événements qui seront ajoutés au bus d'événements. Ce tableau doit avoir une allocation de 1 à 10 éléments.

  L'objet `Event` comporte les champs suivants :

  - `"source"`: chaîne qui définit la source de l'événement.
  - `"detail"`: objet JSON que vous pouvez utiliser pour joindre des informations sur l'événement. Ce champ peut être une carte vide (`{ }`).
  - `"detailType`: chaîne identifiant le type d'événement.
  - `"resources"`: tableau JSON de chaînes identifiant les ressources impliquées dans l'événement. Ce champ peut être un tableau vide.
  - `"time"`: l'horodatage de l'événement fourni sous forme de chaîne. Cela doit suivre le format d'horodatage RFC3339.

Les extraits ci-dessous sont des exemples d'objets valides `Event` :

Exemple 1

```
{
    "source" : "source1",
    "detail" : {
        "key1" : [1,2,3,4],
        "key2" : "strval"
    },
    "detailType" : "sampleDetailType",
    "resources" : ["Resouce1", "Resource2"],
    "time" : "2022-01-10T05:00:10Z"
}
```

Exemple 2

```
{
    "source" : "source1",
    "detail" : {},
    "detailType" : "sampleDetailType"
}
```

Exemple 3

```
{
    "source" : "source1",
    "detail" : {
        "key1" : 1200
    },
    "detailType" : "sampleDetailType",
    "resources" : []
}
```

JavaScript Référence de la fonction de résolution pour Aucune source de données

La fonction deAWS AppSync résolution Requête et réponse avec la source de données de type None vous permet de mettre en forme les demandes pour les opérationsAWS AppSync locales.

Requête

Le gestionnaire de requêtes peut être simple et vous permet de transmettre autant d'informations contextuelles que possible via le`payload` champ.

```
type NONERequest = {
  payload: any;
};
```

Voici un exemple où les arguments du champ sont transmis à la charge utile :

```
export function request(ctx) {
  return {
    payload: context.args
  };
}
```

La valeur du`payload` champ sera transmise au gestionnaire de réponse de la fonction et est disponible dans`context.result`.

Charge utile

Le`payload` champ est un conteneur qui peut être utilisé pour transmettre toutes les données qui sont ensuite mises à disposition du gestionnaire de réponse de la fonction.

Le champ `payload` est facultatif.

Réponse

En l'absence de source de données, la valeur du`payload` champ sera transmise au gestionnaire de réponse de la fonction et définie sur la`context.result` propriété.

Si la forme de la valeur du`payload` champ correspond exactement à la forme du type GraphQL, vous pouvez transférer la réponse à l'aide du gestionnaire de réponse suivant :

```
export function request(ctx) {
  return ctx.result;
}
```

Aucun champ obligatoire ni aucune restriction de forme ne s'appliquent à la réponse renvoyée. Toutefois, étant donné que GraphQL est fortement typé, la réponse résolue doit correspondre au type GraphQL attendu.

JavaScript référence de fonction de résolution pour HTTP

Les fonctions de résolution AWS AppSync HTTP vous permettent d'envoyer des demandes depuis n'importe quel point AWS AppSync de terminaison HTTP, et des réponses depuis votre point de terminaison HTTP vers AWS AppSync. Avec votre gestionnaire de demandes, vous pouvez fournir des indications AWS AppSync sur la nature de l'opération à invoquer. Cette section décrit les différentes configurations du résolveur HTTP pris en charge.

Demande

```
type HTTPRequest = {
  method: 'PUT' | 'POST' | 'GET' | 'DELETE' | 'PATCH';
  params?: {
    query?: { [key: string]: any };
    headers?: { [key: string]: string };
    body?: any;
  };
  resourcePath: string;
};
```

L'extrait suivant est un exemple de requête HTTP POST, avec un `text/plain` corps :

```
export function request(ctx) {
  return {
    method: 'POST',
    params: {
      headers: { 'Content-Type': 'text/plain' },
      body: 'this is an example of text body',
    },
    resourcePath: '/',
  };
}
```

Méthode

Gestionnaire de demandes uniquement

Méthode ou verbe HTTP (GET, POST, PUT, PATCH ou DELETE) envoyé par AWS AppSync au point de terminaison HTTP.

```
"method": "PUT"
```

ResourcePath

Gestionnaire de demandes uniquement

Chemin de la ressource à laquelle vous souhaitez accéder. Avec le point de terminaison dans la source de données HTTP, le chemin de la ressource constitue l'URL à laquelle le service AWS AppSync adresse une demande.

```
"resourcePath": "/v1/users"
```

Lorsque la demande est évaluée, ce chemin est envoyé dans le cadre de la requête HTTP, y compris le point de terminaison HTTP. Ainsi, l'exemple précédent pourrait donner :

```
PUT <endpoint>/v1/users
```

Champ Params

Gestionnaire de demandes uniquement

Permet de spécifier l'action exécutée par votre recherche, plus couramment en définissant la valeur de query à l'intérieur de la section body. Toutefois, il existe plusieurs autres capacités qui peuvent être configurées, comme la mise en forme des réponses.

headers

Informations d'en-tête, sous forme de paires clé-valeur. La clé et la valeur doivent être des chaînes.

Par exemple :

```
"headers" : {
    "Content-Type" : "application/json"
}
```

Les en-têtes `Content-Type` actuellement pris en charge sont les suivants :

```
text/*
application/xml
application/json
application/soap+xml
application/x-amz-json-1.0
application/x-amz-json-1.1
application/vnd.api+json
application/x-ndjson
```

Vous ne pouvez pas définir les en-têtes HTTP suivants :

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

Paires clé-valeur qui spécifient les options courantes, telles que la mise en forme du code pour les réponses JSON. La clé et la valeur doivent être une chaîne. L'exemple suivant montre comment envoyer une chaîne de requête comme `?type=json` :

```
"query" : {
    "type" : "json"
}
```

body

Le corps contient le corps de la demande HTTP que vous choisissez de définir. Le corps de la demande est toujours une chaîne encodée en UTF-8, sauf si le type de contenu spécifie le jeu de caractères.

```
"body":"body string"
```

Réponse

Consultez un exemple ici.

JavaScript référence de la fonction de résolution pour Amazon RDS

La fonction et le résolveur AWS AppSync RDS permettent aux développeurs d'envoyer des SQL requêtes à une base de données de clusters Amazon Aurora à l'aide de l'API RDS Data et de récupérer le résultat de ces requêtes. Vous pouvez écrire SQL des instructions qui sont envoyées à l'API AWS AppSync de `rds` données en utilisant le modèle `sql` étiqueté par `rds` module du module ou en utilisant les `select` fonctions `insert` d'`remove`assistance du module. `update` AWS AppSync utilise l'`ExecuteStatement`action du service de données RDS pour exécuter des instructions SQL sur la base de données.

Rubriques

- Modèle balisé SQL
- Création de déclarations
- Récupération des données
- Fonctions utilitaires
- SQL Select
- Insérer du code SQL
- Mise à jour SQL
- Supprimer SQL

- Forçage de type

Modèle balisé SQL

AWS AppSync le modèle `sql` balisé vous permet de créer une instruction statique capable de recevoir des valeurs dynamiques lors de l'exécution à l'aide d'expressions de modèle. AWS AppSync crée une carte variable à partir des valeurs d'expression pour créer une `SqlParameterized`requête envoyée à l'API de données sans serveur Amazon Aurora. Avec cette méthode, il n'est pas possible que les valeurs dynamiques transmises lors de l'exécution modifient l'instruction d'origine, ce qui pourrait entraîner une exécution involontaire. Toutes les valeurs dynamiques sont transmises sous forme de paramètres, ne peuvent pas modifier l'instruction d'origine et ne sont pas exécutées par la base de données. Cela rend votre requête moins vulnérable aux attaques SQL par injection.

> **Note**
>
> Dans tous les cas, lorsque vous rédigez SQL des déclarations, vous devez suivre les consignes de sécurité afin de gérer correctement les données que vous recevez en entrée.

> **Note**
>
> Le modèle `sql` balisé prend uniquement en charge le transfert de valeurs variables. Vous ne pouvez pas utiliser d'expression pour spécifier dynamiquement les noms des colonnes ou des tables. Vous pouvez toutefois utiliser des fonctions utilitaires pour créer des instructions dynamiques.

Dans l'exemple suivant, nous créons une requête qui filtre en fonction de la valeur de l'`col`argument défini dynamiquement dans la requête GraphQL au moment de l'exécution. La valeur ne peut être ajoutée à l'instruction qu'à l'aide de l'expression de balise :

```
import { sql, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const query = sql`
SELECT * FROM table
WHERE column = ${ctx.args.col}`
  ;
```

```
  return createMySQLStatement(query);
}
```

En faisant passer toutes les valeurs dynamiques par le biais de la carte des variables, nous nous appuyons sur le moteur de base de données pour gérer et nettoyer les valeurs en toute sécurité.

Création de déclarations

Les fonctions et les résolveurs peuvent interagir avec les bases de données MySQL et PostgreSQL. Utilisez `createMySQLStatement` et `createPgStatement` respectivement pour créer des instructions. Par exemple, `createMySQLStatement` vous pouvez créer une requête MySQL. Ces fonctions acceptent jusqu'à deux instructions, ce qui est utile lorsqu'une demande doit récupérer des résultats immédiatement. Avec MySQL, vous pouvez faire :

```
import { sql, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { id, text } = ctx.args;
    const s1 = sql`insert into Post(id, text) values(${id}, ${text})`;
    const s2 = sql`select * from Post where id = ${id}`;
    return createMySQLStatement(s1, s2);
}
```

> **Note**
>
> `createPgStatement`et `createMySQLStatement` n'échappe pas aux instructions créées à l'aide du modèle `sql` balisé et ne cite pas de citations.

Récupération des données

Le résultat de l'instruction SQL exécutée est disponible dans le gestionnaire de réponses de l'`context.result`objet. Le résultat est une chaîne JSON contenant les éléments de réponse de l'`ExecuteStatement`action. Une fois analysé, le résultat prend la forme suivante :

```
type SQLStatementResults = {
    sqlStatementResults: {
        records: any[];
        columnMetadata: any[];
        numberOfRecordsUpdated: number;
```

```
        generatedFields?: any[]
    }[]
}
```

Vous pouvez utiliser cet `toJsonObject` utilitaire pour transformer le résultat en une liste d'objets JSON représentant les lignes renvoyées. Par exemple :

```
import { toJsonObject } from '@aws-appsync/utils/rds';

export function response(ctx) {
    const { error, result } = ctx;
    if (error) {
        return util.appendError(
            error.message,
            error.type,
            result
        )
    }
    return toJsonObject(result)[1][0]
}
```

Notez que cela `toJsonObject` renvoie un tableau de résultats d'instructions. Si vous avez fourni une seule instruction, la longueur du tableau est1. Si vous avez fourni deux instructions, la longueur du tableau est2. Chaque résultat du tableau contient au `0` moins plusieurs lignes. `toJsonObject`renvoie `null` si la valeur du résultat n'est pas valide ou est inattendue.

Fonctions utilitaires

Vous pouvez utiliser les aides utilitaires du module AWS AppSync RDS pour interagir avec votre base de données.

SQL Select

L'`select`utilitaire crée une `SELECT` instruction pour interroger votre base de données relationnelle.

Usage de base

Dans sa forme de base, vous pouvez spécifier la table que vous souhaitez interroger :

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';
```

```
export function request(ctx) {

    // Generates statement:
    // "SELECT * FROM "persons"
    return createPgStatement(select({table: 'persons'}));
}
```

Notez que vous pouvez également spécifier le schéma dans l'identifiant de votre table :

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // SELECT * FROM "private"."persons"
    return createPgStatement(select({table: 'private.persons'}));
}
```

Spécification des colonnes

Vous pouvez définir des colonnes à l'aide de `columns` cette propriété. S'il n'est pas défini sur une valeur, la valeur par défaut est : *

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name']
    }));
}
```

Vous pouvez également spécifier le tableau d'une colonne :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "persons"."name"
    // FROM "persons"
    return createPgStatement(select({
```

```
        table: 'persons',
        columns: ['id', 'persons.name']
    }));
}
```

Limites et compensations

Vous pouvez appliquer `limit` et répondre `offset` à la requête :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // LIMIT :limit
    // OFFSET :offset
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        limit: 10,
        offset: 40
    }));
}
```

Commander par

Vous pouvez trier vos résultats à l'aide de `orderBy` cette propriété. Fournissez un tableau d'objets spécifiant la colonne et une `dir` propriété facultative :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name" FROM "persons"
    // ORDER BY "name", "id" DESC
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        orderBy: [{column: 'name'}, {column: 'id', dir: 'DESC'}]
    }));
}
```

Filtres

Vous pouvez créer des filtres à l'aide de l'objet de condition spéciale :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}}
    }));
}
```

Vous pouvez également combiner des filtres :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME and "id" > :ID
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}, id: {gt: 10}}
    }));
}
```

Vous pouvez également créer des OR déclarations :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME OR "id" > :ID
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: { or: [
            { name: { eq: 'Stephane'} },
```

```
            { id: { gt: 10 } }
        ]}
    }));
}
```

Vous pouvez également annuler une condition avec `not` :

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE NOT ("name" = :NAME AND "id" > :ID)
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: { not: [
            { name: { eq: 'Stephane'} },
            { id: { gt: 10 } }
        ]}
    }));
}
```

Vous pouvez également utiliser les opérateurs suivants pour comparer des valeurs :

Opérateur	Description	Types de valeurs possibles
eq	Égal à	nombre, chaîne, booléen
ne	Non égal à	nombre, chaîne, booléen
le	Inférieur ou égal à	nombre, chaîne
lt	Inférieur à	nombre, chaîne
gm	Supérieur ou égal à	nombre, chaîne
gt	Supérieure à	nombre, chaîne
contient	Comme	chaîne
NE CONTIENT PAS	Pas comme	chaîne

Commence par	Commence par le préfixe	chaîne
between	Entre deux valeurs	nombre, chaîne
L'attribut existe	L'attribut n'est pas nul	nombre, chaîne, booléen
size	vérifie la longueur de l'élément	chaîne

Insérer du code SQL

L'`insert`utilitaire fournit un moyen simple d'insérer des éléments d'une seule ligne dans votre base de données avec l'`INSERT`opération.

Insertions d'un seul article

Pour insérer un élément, spécifiez le tableau, puis transmettez votre objet de valeurs. Les clés d'objet sont mappées aux colonnes de votre tableau. Les noms des colonnes sont automatiquement ignorés et les valeurs sont envoyées à la base de données à l'aide de la variable map :

```
import { insert, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({ table: 'persons', values });

    // Generates statement:
    // INSERT INTO `persons`(`name`)
    // VALUES(:NAME)
    return createMySQLStatement(insertStatement)
}
```

Cas d'utilisation de MySQL

Vous pouvez combiner un `insert` suivi d'un `select` pour récupérer la ligne que vous avez insérée :

```
import { insert, select, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({  table: 'persons', values });
    const selectStatement = select({
```

```
        table: 'persons',
        columns: '*',
        where: { id: { eq: values.id } },
        limit: 1,
    });

    // Generates statement:
    // INSERT INTO `persons`(`name`)
    // VALUES(:NAME)
    // and
    // SELECT *
    // FROM `persons`
    // WHERE `id` = :ID
    return createMySQLStatement(insertStatement, selectStatement)
}
```

Cas d'utilisation de Postgres

Avec Postgres, vous pouvez l'utiliser `returning`pour obtenir des données à partir de la ligne que vous avez insérée. Il accepte * un tableau de noms de colonnes :

```
import { insert, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({
        table: 'persons',
        values,
        returning: '*'
    });

    // Generates statement:
    // INSERT INTO "persons"("name")
    // VALUES(:NAME)
    // RETURNING *
    return createPgStatement(insertStatement)
}
```

Mise à jour SQL

L'`update`utilitaire vous permet de mettre à jour les lignes existantes. Vous pouvez utiliser l'objet condition pour appliquer des modifications aux colonnes spécifiées dans toutes les lignes qui

répondent à la condition. Supposons, par exemple, que nous ayons un schéma qui nous permet de réaliser cette mutation. Nous voulons mettre à jour les `name` de `Person` avec la `id` valeur de`3`, mais uniquement si nous les connaissons (`known_since`) depuis l'année `2000` :

```
mutation Update {
    updatePerson(
        input: {id: 3, name: "Jon"},
        condition: {known_since: {ge: "2000"}}
    ) {
    id
    name
  }
}
```

Notre résolveur de mises à jour ressemble à ceci :

```
import { update, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: { id, ...values }, condition } = ctx.args;
    const where = {
        ...condition,
        id: { eq: id },
    };
    const updateStatement = update({
        table: 'persons',
        values,
        where,
        returning: ['id', 'name'],
    });

    // Generates statement:
    // UPDATE "persons"
    // SET "name" = :NAME, "birthday" = :BDAY, "country" = :COUNTRY
    // WHERE "id" = :ID
    // RETURNING "id", "name"
    return createPgStatement(updateStatement)
}
```

Nous pouvons ajouter une vérification à notre condition pour nous assurer que seule la ligne dont la clé primaire est `id` égale à `3` est mise à jour. De même, pour Postgres`inserts`, vous pouvez utiliser `returning` pour renvoyer les données modifiées.

Supprimer SQL

L'`remove`utilitaire vous permet de supprimer des lignes existantes. Vous pouvez utiliser l'objet de condition sur toutes les lignes qui répondent à la condition. Notez qu'il `delete` s'agit d'un mot clé réservé dans JavaScript. `remove`doit être utilisé à la place :

```
import { remove, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: { id }, condition } = ctx.args;
    const where = { ...condition, id: { eq: id } };
    const deleteStatement = remove({
        table: 'persons',
        where,
        returning: ['id', 'name'],
    });

    // Generates statement:
    // DELETE "persons"
    // WHERE "id" = :ID
    // RETURNING "id", "name"
    return createPgStatement(updateStatement)
}
```

Forçage de type

Dans certains cas, vous souhaiterez peut-être plus de précisions quant au type d'objet approprié à utiliser dans votre déclaration. Vous pouvez utiliser les indications de type fournies pour spécifier le type de vos paramètres. AWS AppSync prend en charge les mêmes indications de type que l'API Data. Vous pouvez convertir vos paramètres en utilisant les `typeHint` fonctions du AWS AppSync `rds` module.

L'exemple suivant vous permet d'envoyer un tableau sous forme de valeur qui est convertie en objet JSON. Nous utilisons l'`->`opérateur pour récupérer l'élément situé `index` 2 dans le tableau JSON :

```
import { sql, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/
rds';

export function request(ctx) {
    const arr = ctx.args.list_of_ids
    const statement = sql`select ${typeHint.JSON(arr)}->2 as value`
```

```
    return createPgStatement(statement)
}

export function response(ctx) {
    return toJsonObject(ctx.result)[0][0].value
}
```

Le casting est également utile lors de la manipulation et de DATE la comparaisonTIME, et TIMESTAMP :

```
import { select, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const when = ctx.args.when
    const statement = select({
        table: 'persons',
        where: { createdAt : { gt: typeHint.DATETIME(when) } }
    })
    return createPgStatement(statement)
}
```

Voici un autre exemple montrant comment envoyer la date et l'heure actuelles :

```
import { sql, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const now = util.time.nowFormatted('YYYY-MM-dd HH:mm:ss')
    return createPgStatement(sql`select ${typeHint.TIMESTAMP(now)}`)
}
```

Indications de type disponibles

- `typeHint.DATE`- Le paramètre correspondant est envoyé sous forme d'objet de `DATE` ce type à la base de données. Le format accepté est `YYYY-MM-DD`.
- `typeHint.DECIMAL`- Le paramètre correspondant est envoyé sous forme d'objet de `DECIMAL` ce type à la base de données.
- `typeHint.JSON`- Le paramètre correspondant est envoyé sous forme d'objet de `JSON` ce type à la base de données.
- `typeHint.TIME`- La valeur de paramètre de chaîne correspondante est envoyée sous forme d'objet de `TIME` ce type à la base de données. Le format accepté est `HH:MM:SS[.FFF]`.

- `typeHint.TIMESTAMP`- La valeur de paramètre de chaîne correspondante est envoyée sous forme d'objet de `TIMESTAMP` ce type à la base de données. Le format accepté est `YYYY-MM-DD HH:MM:SS[.FFF]`.
- `typeHint.UUID`- La valeur de paramètre de chaîne correspondante est envoyée sous forme d'objet de `UUID` ce type à la base de données.

Référence du modèle de mappage du résolveur (VTL)

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Les sections suivantes décrivent comment les opérations utilitaires peuvent être utilisées dans les modèles de mappage.

Rubriques

Présentation du modèle de mappage Resolver

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

AWS AppSync vous permet de répondre aux requêtes GraphQL en effectuant des opérations sur vos ressources. Pour chaque champ GraphQL sur lequel vous souhaitez exécuter une requête ou une mutation, un résolveur doit être attaché afin de communiquer avec une source de données. La communication s'effectue généralement par le biais de paramètres ou d'opérations propres à la source de données.

Les résolveurs sont les connecteurs entre GraphQL et une source de données. Ils expliquent AWS AppSync comment traduire une requête GraphQL entrante en instructions pour votre source de données principale, et comment traduire la réponse de cette source de données en réponse GraphQL. Ils sont écrits dans le langage de modèle Apache Velocity (VTL), qui prend votre requête en entrée et génère un document JSON contenant les instructions pour le résolveur. Vous pouvez utiliser des modèles de mappage pour des instructions simples, telles que la transmission d'arguments provenant de champs GraphQL, ou pour des instructions plus complexes, telles que le fait de parcourir des arguments en boucle pour créer un élément avant de l'insérer dans DynamoDB.

Il existe deux types de résolveurs AWS AppSync qui exploitent les modèles de mappage de manière légèrement différente :

- Résolveurs d'unités
- Résolveurs de pipelines

Résolveurs d'unités

Les résolveurs d'unités sont des entités autonomes qui incluent uniquement un modèle de demande et de réponse. Utilisez-les pour les opérations simples et uniques, comme lister des éléments à partir d'une seule source de données.

- Modèles de demande : prenez la demande entrante après l'analyse d'une opération GraphQL et convertissez-la en une configuration de demande pour l'opération de source de données sélectionnée.
- Modèles de réponses : interprétez les réponses de votre source de données et mappez-les à la forme du type de sortie du champ GraphQL.

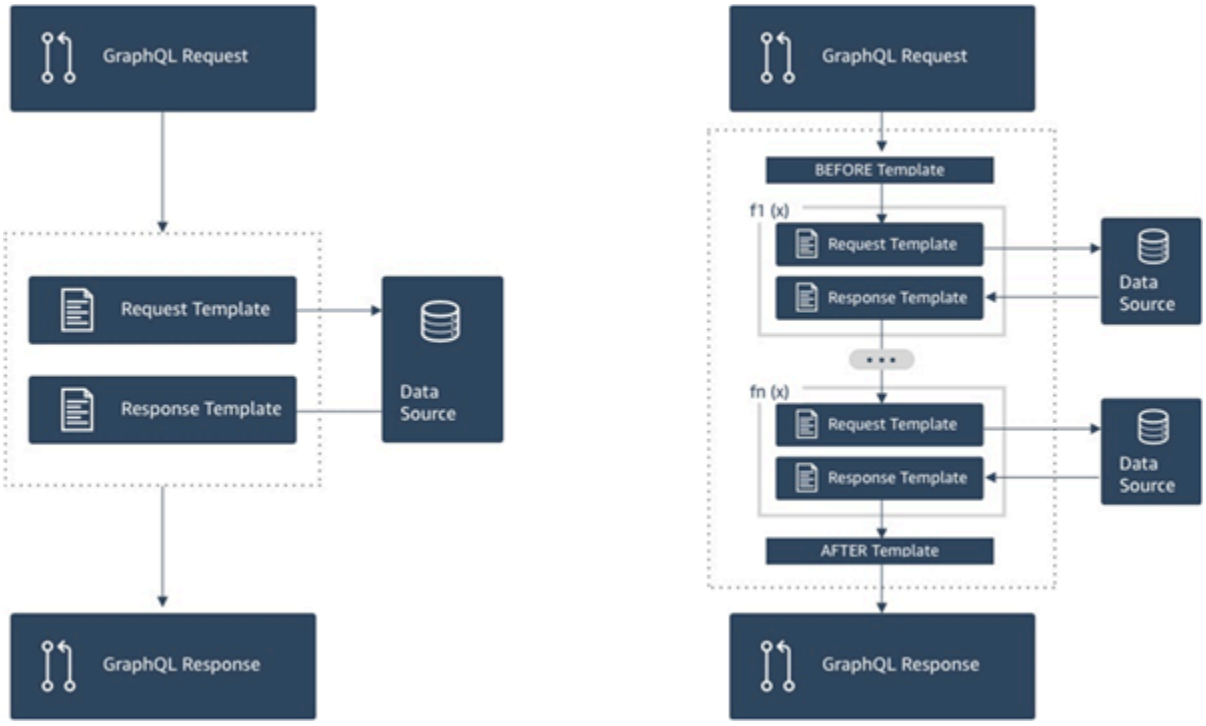
Résolveurs de pipelines

Les résolveurs de pipeline contiennent une ou plusieurs fonctions exécutées dans un ordre séquentiel. Chaque fonction inclut un modèle de demande et un modèle de réponse. Un résolveur

de pipeline possède également un modèle avant et un modèle après qui entourent la séquence de fonctions que le modèle contient. Le modèle After correspond au type de sortie du champ GraphQL. Les résolveurs de pipeline diffèrent des résolveurs d'unités dans la façon dont le modèle de réponse mappe la sortie. Un résolveur de pipeline peut mapper sur n'importe quelle sortie de votre choix, y compris l'entrée d'une autre fonction ou le modèle postérieur du résolveur de pipeline.

Les fonctions de résolution de pipeline vous permettent d'écrire une logique commune que vous pouvez réutiliser dans plusieurs résolveurs de votre schéma. Les fonctions sont associées directement à une source de données et, comme un résolveur d'unités, contiennent le même format de modèle de mappage de demandes et de réponses.

Le schéma suivant montre le flux de processus d'un résolveur d'unités sur la gauche et d'un résolveur de pipeline sur la droite.



Les résolveurs de pipeline contiennent un surensemble des fonctionnalités prises en charge par les résolveurs unitaires, et bien plus encore, au prix d'un peu plus de complexité.

Anatomie d'un résolveur de pipeline

Un résolveur de pipeline est composé d'un modèle avant mappage, d'un modèle après mappage et d'une liste de fonctions. Chaque fonction possède un modèle de mappage de requêtes et de réponses qu'elle exécute par rapport à une source de données. Comme un résolveur de pipeline délègue l'exécution à une liste de fonctions, il n'est lié à aucune source de données. Les résolveurs

d'unité et les fonctions sont des primitifs qui exécutent l'opération sur les sources de données. Consultez la présentation du modèle de mappage Resolver pour plus d'informations.

Avant le modèle de mappage

Le modèle de mappage de demandes d'un résolveur de pipeline, ou l'étape Before, vous permet d'exécuter une certaine logique de préparation avant d'exécuter les fonctions définies.

Liste des fonctions

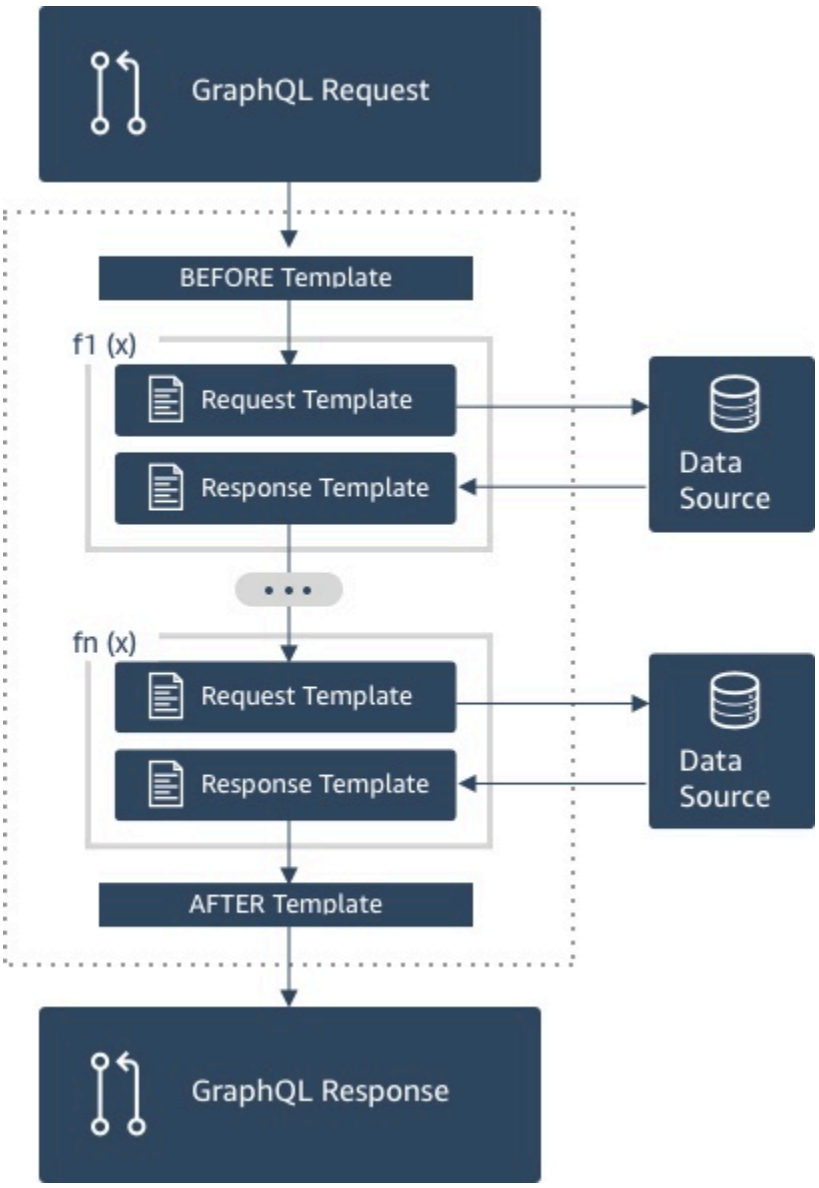
La liste des fonctions d'un résolveur de pipeline est exécutée dans l'ordre. Le résultat évalué du modèle de mappage de demande du résolveur de pipeline est mis à disposition de la première fonction en tant que `$ctx.prev.result`. Chaque sortie d'une fonction est disponible pour la fonction suivante en tant que `$ctx.prev.result`.

Modèle de mappage Après

Le modèle de mappage des réponses d'un résolveur de pipeline, ou l'étape After, vous permet d'exécuter une logique de mappage finale entre la sortie de la dernière fonction et le type de champ GraphQL attendu. La sortie de la dernière fonction de la liste de fonctions est disponible dans le modèle de mappage du résolveur de pipeline en tant que `$ctx.prev.result` ou `$ctx.result`.

Flux d'exécution

Étant donné qu'un résolveur de pipeline est composé de deux fonctions, la liste ci-dessous représente le flux d'exécution lorsque le résolveur est invoqué :



1. Résolveur de pipeline avant le modèle de mappage
2. Fonction 1 : Modèle de mappage de demande de fonction
3. Fonction 1 : Appel de source de données
4. Fonction 1 : Modèle de mappage de réponse de fonction
5. Fonction 2 : Modèle de mappage de demande de fonction
6. Fonction 2 : Appel de source de données
7. Fonction 2 : Modèle de mappage de réponse de fonction
8. Résolveur de pipeline après le modèle de mappage

> **Note**
>
> Le flux d'exécution du résolveur de pipeline est unidirectionnel et défini de manière statique sur le résolveur.

Utilitaires VTL (Apache Velocity Template Language) utiles

À mesure qu'une application devient plus complexe, les utilitaires VTL et les directives permettent de faciliter la productivité de développement. Les utilitaires suivants peuvent vous aider si vous travaillez avec des résolveurs de pipeline.

$ctx.stash

Le stash est disponible dans chaque résolveur et chaque modèle de mappage de fonctions. `Map` La même instance stash perdure pendant une exécution de résolveur. Cela signifie que vous pouvez utiliser le stash pour transmettre des données arbitraires sur des modèles de mappage de demande et de réponse, et sur des fonctions dans un résolveur de pipeline. Le stash expose les mêmes méthodes que la structure de données cartographiques Java.

$ctx.prev.result

Le `$ctx.prev.result` représente le résultat de l'opération précédente exécutée dans le résolveur de pipeline.

Si l'opération précédente était le modèle de mappage Before du résolveur de pipeline, elle `$ctx.prev.result` représente le résultat de l'évaluation du modèle et est mise à la disposition de la première fonction du pipeline. Si l'opération précédente est la première fonction, alors `$ctx.prev.result` représente le résultat de la première fonction et il est disponible pour la seconde fonction du pipeline. Si l'opération précédente était la dernière fonction, elle `$ctx.prev.result` représente la sortie de la dernière fonction et est mise à la disposition du modèle After mapping du résolveur de pipeline.

#return(data: Object)

La directive `#return(data: Object)` s'avère utile si vous avez besoin de revenir en arrière prématurément depuis un modèle de mappage. `#return(data: Object)` est analogue au mot clé return dans les langages de programmation, car il renvoie au bloc de logique à portée le plus proche. Cela signifie qu'utiliser `#return` dans un modèle de mappage de résolveur renvoie des données

à partir du résolveur. L'utilisation de `#return(data: Object)` dans un modèle de mappage de résolveur définit `data` sur le champ GraphQL. De plus, l'utilisation de `#return(data: Object)` à partir d'un modèle de mappage de fonction renvoie des données à partir de la fonction et continue l'exécution vers la prochaine fonction du pipeline ou vers le modèle de mappage de réponse du résolveur.

#return

C'est la même chose que`#return(data: Object)`, mais elle `null` sera renvoyée à la place.

$util.error

L'utilitaire `$util.error` est utile pour envoyer une erreur de champ. L'utilisation de `$util.error` dans un modèle de mappage de fonction envoie immédiatement une erreur de champ qui empêche l'exécution des fonctions suivantes. Pour plus de détails et pour d'autres `$util.error` signatures, consultez la référence de l'utilitaire de modèle de mappage Resolver.

$util.appendError

`$util.appendError` est similaire à `$util.error()`, avec une différence majeure : il n'interrompt pas l'évaluation du modèle de mappage. Au lieu de cela, il signale qu'une erreur s'est produite avec le champ, mais autorise l'évaluation du modèle et renvoie les données. L'utilisation de `$util.appendError` dans une fonction ne perturbera pas le flux d'exécution du pipeline. Pour plus de détails et pour d'autres `$util.error` signatures, consultez la référence de l'utilitaire de modèle de mappage Resolver.

Exemple de modèle

Supposons que vous disposiez d'une source de données DynamoDB et d'un résolveur d'unités dans un champ `getPost(id:ID!)` nommé qui renvoie `Post` un type avec la requête GraphQL suivante :

```
getPost(id:1){
    id
    title
    content
}
```

Votre modèle de résolveur peut ressembler à ce qui suit :

```
    {
```

```
    "version" : "2018-05-29",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

Ce modèle remplace la valeur du paramètre d'entrée `id` par `1` (au lieu de `${ctx.args.id}`) et génère le code JSON suivant :

```
     {
    "version" : "2018-05-29",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "1" }
    }
}
```

AWS AppSync utilise ce modèle pour générer des instructions permettant de communiquer avec DynamoDB et d'obtenir des données (ou d'effectuer d'autres opérations le cas échéant). Une fois les données renvoyées, AWS AppSync les exécute via un modèle de mappage de réponse facultatif que vous pouvez utiliser pour effectuer des opérations de mise en forme ou de logique sur les données. Par exemple, lorsque nous récupérons les résultats de DynamoDB, ils peuvent ressembler à ceci :

```
{
        "id" : 1,
        "theTitle" : "AWS AppSync works offline!",
        "theContent-part1" : "It also has realtime functionality",
        "theContent-part2" : "using GraphQL"
}
```

Vous pouvez choisir de regrouper deux des champs en un seul avec le modèle de mappage de réponse suivant :

```
{
        "id" : $util.toJson($context.data.id),
        "title" : $util.toJson($context.data.theTitle),
        "content" : $util.toJson("${context.data.theContent-part1}
 ${context.data.theContent-part2}")
}
```

Voici comment les données sont formatées une fois que le modèle leur a été appliqué :

```
{
        "id" : 1,
        "title" : "AWS AppSync works offline!",
        "content" : "It also has realtime functionality using GraphQL"
}
```

Ces données sont renvoyées au client sous forme de réponse comme suit :

```
{
        "data": {
                "getPost":      {
                        "id" : 1,
                        "title" : "AWS AppSync works offline!",
                        "content" : "It also has realtime functionality using GraphQL"
                }
        }
}
```

Notez que, dans la plupart des cas, les modèles de mappage de réponse sont une simple transmission de données, lesquels sont surtout différents si vous renvoyez un élément individuel ou une liste d'éléments. Pour un élément individuel, la transmission est :

```
$util.toJson($context.result)
```

Pour une liste, la transmission est généralement :

```
$util.toJson($context.result.items)
```

Pour voir d'autres exemples de résolveurs unitaires et de résolveurs de pipeline, consultez les didacticiels sur les résolveurs.

Règles de désérialisation des modèles de mappage évalués

Les modèles de mappage sont évalués selon une chaîne. Dans AWS AppSync, la chaîne de sortie doit suivre une structure JSON pour être valable.

En outre, les règles de désérialisation suivantes sont appliquées.

Les clés en double ne sont pas autorisées dans les objets JSON

Si la chaîne de modèle de mappage évaluée représente un objet JSON ou contient un objet qui contient des clés en double, le modèle de mappage renvoie le message d'erreur suivant :

`Duplicate field 'aField' detected on Object. Duplicate JSON keys are not allowed.`

Exemple de clé en double dans un modèle de mappage de demande évalué :

```
{
    "version": "2018-05-29",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "1",
        "field": "getPost" ## key 'field' has been redefined
    }
}
```

Pour corriger cette erreur, ne redéfinissez pas les clés dans les objets JSON.

Les caractères de fin ne sont pas autorisés dans les objets JSON

Si la chaîne de modèle de mappage évaluée représente un objet JSON et contient des caractères de fin étrangers, le modèle de mappage renvoie le message d'erreur suivant :

`Trailing characters at the end of the JSON string are not allowed.`

Exemple de caractères de fin dans un modèle de mappage de demande évalué :

```
{
    "version": "2018-05-29",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "1",
    }
}extraneouschars
```

Pour corriger cette erreur, assurez-vous que les modèles évalués sont strictement conformes au JSON.

Guide de programmation du modèle de mappage Resolver

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Il s'agit d'un tutoriel de type livre de recettes sur la programmation en langage Apache VTL (Velocity Template Language) dans AWS AppSync. Si vous connaissez d'autres langages de programmation tels que JavaScript C ou Java, cela devrait être assez simple.

AWS AppSync utilise VTL pour traduire les requêtes GraphQL des clients en demandes adressées à votre source de données. Ensuite, il inverse ce processus pour convertir la réponse de la source de données en une réponse GraphQL. Le VTL est un langage de modèle logique qui vous permet de manipuler à la fois la demande et la réponse dans le flux de demande/réponse standard d'une application Web, en utilisant des techniques telles que :

- Valeurs par défaut pour les nouveaux éléments
- Validation et formatage des informations fournies
- Transformation et mise en forme des données
- Itération sur des listes, des cartes et des tableaux pour manipuler ou modifier les valeurs
- Filtrage/modification des réponses en fonction de l'identité de l'utilisateur
- Vérifications d'autorisation complexes

Par exemple, vous souhaiterez peut-être valider un numéro de téléphone dans le service sur un argument GraphQL ou convertir un paramètre d'entrée en majuscules avant de le stocker dans DynamoDB. Les systèmes clients peuvent également fournir un code, dans le cadre d'un argument GraphQL, d'une demande de jeton JWT ou d'un en-tête HTTP, et ne répondre avec des données que si ce code correspond à une chaîne spécifique dans une liste. Ce sont toutes des vérifications logiques que vous pouvez effectuer avec VTL in AWS AppSync.

VTL permet d'appliquer une logique à l'aide de techniques de programmation qui peuvent vous être familières. Toutefois, son exécution est limitée au flux de demande/réponse standard pour garantir que votre API GraphQL est scalable lorsque votre base d'utilisateurs croît. Comme il est AWS AppSync également compatible en AWS Lambda tant que résolveur, vous pouvez écrire des

fonctions Lambda dans le langage de programmation de votre choix (Node.js, Python, Go, Java, etc.) si vous avez besoin de plus de flexibilité.

Installation

Une technique courante lors de l'apprentissage d'une langue consiste à imprimer les résultats (par exemple, `console.log(variable)` dans JavaScript) pour voir ce qui se passe. Dans ce didacticiel, nous illustrons cela en créant un schéma GraphQL simple et en transmettant une carte de valeurs à une fonction Lambda. La fonction Lambda affiche les valeurs, puis les utilise pour répondre. Ceci vous permet de comprendre le flux de demande/réponse et de voir différentes techniques de programmation.

Commencez par créer le schéma GraphQL suivant :

```
type Query {
    get(id: ID, meta: String): Thing
}

type Thing {
    id: ID!
    title: String!
    meta: String
}

schema {
    query: Query
}
```

À présent, créez la fonction AWS Lambda suivante, en utilisant Node.js comme langage :

```
exports.handler = (event, context, callback) => {
    console.log('VTL details: ', event);
    callback(null, event);
};
```

Dans le volet Sources de données de la console AWS AppSync , ajoutez cette fonction Lambda en tant que nouvelle source de données. Revenez à la page Schema (Schéma) de la console et cliquez sur le bouton ATTACH (ATTACHER) situé à droite, en regard de la requête `get(...):Thing`. Pour le modèle de demande, choisissez le modèle existant dans le menu Invoke and forward arguments (Invoquer et transférer des arguments). Pour le modèle de réponse, choisissez Return Lambda result (Renvoyer le résultat de Lambda).

Ouvrez Amazon CloudWatch Logs pour votre fonction Lambda en un seul endroit, puis dans l'onglet Requêtes de la AWS AppSync console, exécutez la requête GraphQL suivante :

```
query test {
  get(id:123 meta:"testing"){
    id
    meta
  }
}
```

La réponse GraphQL doit contenir `id:123` et `meta:testing`, parce que la fonction Lambda en fait écho. Au bout de quelques secondes, vous devriez également voir un enregistrement dans CloudWatch Logs contenant ces informations.

Variables

VTL utilise des références, que vous pouvez utiliser pour stocker ou manipuler les données. Il existe trois types de références dans VTL : les variables, les propriétés et les méthodes. Les variables sont précédées d'un symbole $ et sont créées avec la directive `#set` :

```
#set($var = "a string")
```

Les variables stockent des types similaires que vous connaissez bien dans d'autres langages, comme des nombres, des chaînes, des tableaux, des listes et des cartes. Vous avez peut-être remarqué qu'une charge utile JSON était envoyée dans le modèle de demande par défaut pour les résolveurs Lambda :

```
"payload": $util.toJson($context.arguments)
```

Deux choses sont à noter ici : tout d'abord, AWS AppSync fournit plusieurs fonctions pratiques pour les opérations courantes. Dans cet exemple, `$util.toJson` convertit une variable au format JSON. Deuxièmement, la variable `$context.arguments` est automatiquement renseignée à partir d'une demande GraphQL comme un objet carte. Vous pouvez créer une nouvelle carte comme suit :

```
#set( $myMap = {
  "id": $context.arguments.id,
  "meta": "stuff",
  "upperMeta" : $context.arguments.meta.toUpperCase()
```

```
} )
```

Vous venez de créer une variable nommée `$myMap`, qui possède des clés de `id`, `meta` et `upperMeta`. Ceci illustre aussi les éléments suivants :

- `id` est renseignée avec une clé issue des arguments GraphQL. Il est courant dans VTL de saisir des arguments à partir des clients.
- L'argument `meta` est codé en dur avec une valeur, exposant les valeurs par défaut.
- `upperMeta` transforme l'argument `meta` à l'aide d'une méthode `.toUpperCase()`.

Placez le code précédent en haut de votre modèle de demande et modifiez `payload` pour utiliser la nouvelle variable `$myMap` :

```
"payload": $util.toJson($myMap)
```

Exécutez votre fonction Lambda et vous pourrez voir le changement de réponse ainsi que ces données dans CloudWatch les journaux. Au fur et à mesure que vous parcourrez le reste de ce didacticiel, nous continuerons à renseigner `$myMap` pour que vous puissiez exécuter des tests similaires.

Vous pouvez également définir des propriétés sur vos variables. Ce peut être de simples chaînes, des tableaux ou des éléments JSON :

```
#set($myMap.myProperty = "ABC")
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])
#set($myMap.jsonProperty = {
    "AppSync" : "Offline and Realtime",
    "Cognito" : "AuthN and AuthZ"
})
```

Références silencieuses

Comme VTL est un langage de création de modèles, par défaut, toute référence que vous lui donnez exécute une méthode `.toString()`. Si la référence n'est pas définie, elle affiche la représentation de référence en tant que chaîne. Par exemple :

```
#set($myValue = 5)
```

```
##Prints '5'
$myValue

##Prints '$somethingelse'
$somethingelse
```

Pour résoudre cela, VTL possède une syntaxe de type référence calme ou référence silencieuse, qui indique au moteur de modèle de supprimer ce comportement. La syntaxe correspondante est `$!{}`. Par exemple, si nous modifions légèrement le code précédent pour utiliser `$!{somethingelse}`, l'affichage est supprimé :

```
#set($myValue = 5)
##Prints '5'
$myValue

##Nothing prints out
$!{somethingelse}
```

Appel de méthodes

Dans l'exemple précédent, nous vous a montré comment créer une variable et les valeurs définies simultanément. Vous pouvez également procéder en deux étapes en ajoutant des données à votre carte :

```
#set ($myMap = {})
#set ($myList = [])

##Nothing prints out
$!{myMap.put("id", "first value")}
##Prints "first value"
$!{myMap.put("id", "another value")}
##Prints true
$!{myList.add("something")}
```

TOUTEFOIS, vous devez connaître une information importante sur ce comportement. Même si la notation de référence silencieuse `$!{}` vous permet d'appeler des méthodes, comme ci-dessus, elle NE supprime PAS la valeur renvoyée de la méthode exécutée. C'est pourquoi nous avons noté `##Prints "first value"` et `##Prints true` ci-dessus. Cela peut entraîner des erreurs lors d'une itération sur des cartes ou des listes, comme l'insertion d'une valeur là où une clé existe déjà, car la sortie ajoute des chaînes inattendues au modèle lors de l'évaluation.

Une solution de contournement consiste parfois à appeler les méthodes à l'aide d'une directive `#set` et d'ignorer la variable. Par exemple :

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

Vous pouvez utiliser cette technique dans vos modèles, car elle empêche l'impression de chaînes inattendues dans le modèle. AWS AppSync fournit une fonction pratique alternative qui offre le même comportement dans une notation plus succincte. Cela vous évite d'avoir à réfléchir à ces spécificités de mise en œuvre. Vous pouvez accéder à cette fonction sous `$util.quiet()` ou son alias `$util.qr()`. Par exemple :

```
#set ($myMap = {})
#set ($myList = [])

##Nothing prints out
$util.quiet($myMap.put("id", "first value"))
##Nothing prints out
$util.qr($myList.add("something"))
```

Chaînes

Comme avec de nombreux langages de programmation, il peut être difficile de gérer les chaînes, notamment lorsque vous voulez les générer à partir de variables. Des éléments communs se manifestent dans le cadre de VTL.

Supposons que vous insérez des données sous forme de chaîne dans une source de données telle que DynamoDB, mais qu'elles soient renseignées à partir d'une variable, comme un argument GraphQL. Une chaîne a des guillemets doubles et vous avez simplement besoin de `"${}"` pour référencer la variable dans une chaîne (donc d'aucun `!` comme dans la notation de référence silencieuse). Ceci est similaire à un modèle littéral dans JavaScript : https://developer.mozilla.org/en-US/docs/Web/ JavaScript /reference/Template_Literals

```
#set($firstname = "Jeff")
$!{myMap.put("Firstname", "${firstname}")}
```

Vous pouvez le constater dans les modèles de requête DynamoDB, par `"author": { "S" : "${context.arguments.author}"}` exemple lorsque vous utilisez des arguments provenant de clients GraphQL ou pour la génération automatique d'identifiants, par exemple. `"id" : { "S" :`

`"$util.autoId()"}` Cela signifie que vous pouvez référencer une variable ou le résultat d'une méthode à l'intérieur d'une chaîne pour renseigner les données.

Vous pouvez également utiliser les méthodes publiques de la classe String Java, telles que l'extraction d'une sous-chaîne :

```
#set($bigstring = "This is a long string, I want to pull out everything after the
 comma")
#set ($comma = $bigstring.indexOf(','))
#set ($comma = $comma +2)
#set ($substring = $bigstring.substring($comma))

$util.qr($myMap.put("substring", "${substring}"))
```

La concaténation de chaînes est aussi une tâche très courante. Vous pouvez procéder avec des références de variables seules ou avec des valeurs statiques :

```
#set($s1 = "Hello")
#set($s2 = " World")

$util.qr($myMap.put("concat","$s1$s2"))
$util.qr($myMap.put("concat2","Second $s1 World"))
```

Boucles

Maintenant que vous avez créé des variables et appelé des méthodes, vous pouvez ajouter une logique à votre code. Contrairement à d'autres langages, VTL autorise uniquement des boucles avec un nombre d'itérations prédéterminé. Il n'existe pas de `do..while` dans Velocity. Cette conception garantit que le processus d'évaluation arrivera à son terme et elle fournit des limites pour la scalabilité lorsque vos opérations GraphQL s'exécutent.

L'instruction `#foreach` permet de créer des boucles et nécessite que vous fournissiez une variable de boucle et un objet itérable, tel qu'un tableau, une liste, une carte ou une collection. Un exemple de programmation standard avec une boucle `#foreach` consiste à itérer sur les éléments d'une collection et à les afficher, pour les prélever et les ajouter à la carte :

```
#set($start = 0)
#set($end = 5)
#set($range = [$start..$end])
```

```
#foreach($i in $range)
   ##$util.qr($myMap.put($i, "abc"))
   ##$util.qr($myMap.put($i, $i.toString()+"foo")) ##Concat variable with string
   $util.qr($myMap.put($i, "${i}foo"))     ##Reference a variable in a string with
 "${varname}"
#end
```

Cet exemple illustre quelques éléments. Le premier est l'utilisation des variables avec l'opérateur `[..]` de plage pour créer un objet itérable. Ensuite, chaque élément est référencé par une variable `$i` que vous pouvez utiliser. Dans l'exemple précédent, vous voyez aussi des commentaires qui sont signalés par deux dièses ##. Ceci illustre également l'utilisation de la variable de boucle dans les clés et les valeurs, ainsi que différentes méthodes de concaténation utilisant les chaînes.

Notez que `$i` est un entier, si bien que vous pouvez appeler une méthode `.toString()`. Pour les types GraphQL de INT, cela peut être utile.

Vous pouvez également utiliser un opérateur de plage directement, par exemple :

```
#foreach($item in [1..5])
    ...
#end
```

Arrays (tableaux)

Vous avez manipulé une carte jusqu'à ce stade, mais les tableaux sont également courants dans VTL. Avec les tableaux, vous avez également accès à certaines méthodes sous-jacentes telles que `.isEmpty()`, `.size()`, `.set()`, `.get()` et `.add()`, comme illustré ci-dessous :

```
#set($array = [])
#set($idx = 0)

##adding elements
$util.qr($array.add("element in array"))
$util.qr($myMap.put("array", $array[$idx]))

##initialize array vals on create
#set($arr2 = [42, "a string", 21, "test"])

$util.qr($myMap.put("arr2", $arr2[$idx]))
$util.qr($myMap.put("isEmpty", $array.isEmpty()))  ##isEmpty == false
$util.qr($myMap.put("size", $array.size()))
```

```
##Get and set items in an array
$util.qr($myMap.put("set", $array.set(0, 'changing array value')))
$util.qr($myMap.put("get", $array.get(0)))
```

L'exemple précédent utilisait la notation d'index de tableau pour récupérer un élément avec`arr2[$idx]`. Vous pouvez rechercher par nom à partir d'une carte/d'un dictionnaire d'une manière similaire :

```
#set($result = {
    "Author" : "Nadia",
    "Topic" : "GraphQL"
})

$util.qr($myMap.put("Author", $result["Author"]))
```

Cela est très courant lors du filtrage des résultats provenant des sources de données dans les modèles de réponse lors de l'utilisation de conditions.

Vérifications conditionnelles

La section antérieure avec `#foreach` illustrait des exemples de l'utilisation d'une logique pour transformer les données avec VTL. Vous pouvez également appliquer des contrôles conditionnels pour évaluer les données lors de l'exécution :

```
#if(!$array.isEmpty())
    $util.qr($myMap.put("ifCheck", "Array not empty"))
#else
    $util.qr($myMap.put("ifCheck", "Your array is empty"))
#end
```

La vérification `#if()` ci-dessus d'une expression booléenne est correcte, mais vous pouvez également utiliser des opérateurs et `#elseif()` pour une ramification :

```
#if ($arr2.size() == 0)
    $util.qr($myMap.put("elseIfCheck", "You forgot to put anything into this array!"))
#elseif ($arr2.size() == 1)
    $util.qr($myMap.put("elseIfCheck", "Good start but please add more stuff"))
#else
    $util.qr($myMap.put("elseIfCheck", "Good job!"))
```

```
#end
```

Ces deux exemples montrent la négation (!) et l'égalité (==). Nous pouvons également utiliser ||, &&, >, <, >=, <=, et !=

```
#set($T = true)
#set($F = false)

#if ($T || $F)
  $util.qr($myMap.put("OR", "TRUE"))
#end

#if ($T && $F)
  $util.qr($myMap.put("AND", "TRUE"))
#end
```

Remarque : Seuls `Boolean.FALSE` et `null` sont considérés comme false dans les conditions. Zéro (0) et les chaînes vides ("") ne sont pas équivalents à false.

Opérateurs

Aucun langage de programmation ne serait complet sans certains opérateurs pour effectuer certaines opérations mathématiques. Voici quelques exemples pour commencer :

```
#set($x = 5)
#set($y = 7)
#set($z = $x + $y)
#set($x-y = $x - $y)
#set($xy = $x * $y)
#set($xDIVy = $x / $y)
#set($xMODy = $x % $y)

$util.qr($myMap.put("z", $z))
$util.qr($myMap.put("x-y", $x-y))
$util.qr($myMap.put("x*y", $xy))
$util.qr($myMap.put("x/y", $xDIVy))
$util.qr($myMap.put("x|y", $xMODy))
```

Association de boucles et de conditions

Il est très courant lors de la transformation de données dans VTL, par exemple avant d'écrire ou de lire dans une source de données, d'itérer sur des objets puis d'effectuer des vérifications

avant d'effectuer une action. L'association de certains outils des sections précédentes vous offre un grand nombre de fonctionnalités. Un outil pratique est de savoir que `#foreach` vous fournit automatiquement un nombre (`.count`) sur chaque élément :

```
#foreach ($item in $arr2)
  #set($idx = "item" + $foreach.count)
  $util.qr($myMap.put($idx, $item))
#end
```

Par exemple, vous voulez prélever des valeurs d'une carte seulement si sa taille est inférieure à une valeur donnée. L'utilisation du nombre avec les conditions et l'instruction `#break` vous permet d'écrire :

```
#set($hashmap = {
  "DynamoDB" : "https://aws.amazon.com/dynamodb/",
  "Amplify" : "https://github.com/aws/aws-amplify",
  "DynamoDB2" : "https://aws.amazon.com/dynamodb/",
  "Amplify2" : "https://github.com/aws/aws-amplify"
})

#foreach ($key in $hashmap.keySet())
    #if($foreach.count > 2)
    #break
  #end
    $util.qr($myMap.put($key, $hashmap.get($key)))
#end
```

Le bloc `#foreach` précédent est itéré avec `.keySet()`, que vous pouvez utiliser sur les cartes. Cela vous donne l'accès nécessaire pour obtenir `$key` et référencer la valeur avec un `.get($key)`. Les arguments GraphQL provenant des clients dans AWS AppSync sont stockés en tant que carte. Ils peuvent également être itérés avec `.entrySet()` et vous pouvez ensuite accéder à la fois aux clés et aux valeurs en tant qu'ensemble, et renseigner les autres variables ou effectuer des vérifications conditionnelles complexes, telles que la validation ou la transformation de l'entrée :

```
#foreach( $entry in $context.arguments.entrySet() )
#if ($entry.key == "XYZ" && $entry.value == "BAD")
    #set($myvar = "...")
  #else
    #break
  #end
```

```
#end
```

D'autres exemples courants sont le remplissage automatique des informations par défaut, comme les versions initiales des objets lors de la synchronisation des données (très importante pour la résolution des conflits) ou le propriétaire par défaut d'un objet pour les vérifications d'autorisation. Mary a créé ce billet de blog, donc :

```
#set($myMap.owner ="Mary")
#set($myMap.defaultOwners = ["Admins", "Editors"])
```

Contexte

Maintenant que vous savez comment effectuer des vérifications logiques dans les AWS AppSync résolveurs avec VTL, examinez l'objet de contexte :

```
$util.qr($myMap.put("context", $context))
```

Celui-ci contient toutes les informations auxquelles vous pouvez accéder dans votre demande GraphQL. Pour une explication détaillée, consultez la référence sur le contexte.

Filtrage

Jusqu'ici dans ce didacticiel, toutes les informations provenant de votre fonction Lambda ont été renvoyées à la requête GraphQL avec une transformation JSON très simple :

```
$util.toJson($context.result)
```

La logique VTL est tout aussi puissante lorsque vous obtenez des réponses à partir d'une source de données, en particulier lors de la réalisation de vérifications d'autorisation sur les ressources. Passons en revue quelques exemples. Tout d'abord, essayez de modifier votre modèle de réponse comme suit :

```
#set($data = {
    "id" : "456",
    "meta" : "Valid Response"
})

$util.toJson($data)
```

Quoi qu'il se passe avec votre opération GraphQL, les valeurs codées en dur sont renvoyées au client. Modifiez cela légèrement pour que le champ `meta` soit renseigné à partir de la réponse Lambda, définie précédemment dans le didacticiel dans la valeur `elseIfCheck` lors de l'examen des conditions :

```
#set($data = {
    "id" : "456"
})

#foreach($item in $context.result.entrySet())
    #if($item.key == "elseIfCheck")
        $util.qr($data.put("meta", $item.value))
    #end
#end

$util.toJson($data)
```

`$context.result` est une carte. Vous pouvez donc utiliser `entrySet()` pour appliquer une logique sur les clés ou les valeurs renvoyées. Etant donné que `$context.identity` contient des informations sur l'utilisateur qui a effectué l'opération GraphQL, si vous renvoyez les informations d'autorisation à partir de la source de données, vous pouvez décider de renvoyer des données complètes, des données partielles ou aucune donnée à un utilisateur en fonction de votre logique. Modifiez votre modèle de réponse pour qu'il ressemble à ce qui suit :

```
#if($context.result["id"] == 123)
    $util.toJson($context.result)
  #else
    $util.unauthorized()
#end
```

Si vous exécutez votre requête GraphQL, les données seront renvoyées comme d'habitude. Toutefois, si vous modifiez l'argument id en spécifiant autre chose que 123 (`query test { get(id:456 meta:"badrequest"){} }`), vous obtenez un message d'échec d'autorisation.

Vous trouverez d'autres exemples de scénarios d'autorisation dans la section sur les cas d'utilisation des autorisations.

Annexe - Exemple de modèle

Si vous avez suivi ce didacticiel, vous avez peut-être créé le modèle ci-dessous étape par étape. Si vous ne l'avez pas fait, nous l'incluons ci-dessous pour que vous puissiez le copier à des fins de test.

Modèle de demande

```
#set( $myMap = {
  "id": $context.arguments.id,
  "meta": "stuff",
  "upperMeta" : "$context.arguments.meta.toUpperCase()"
} )

##This is how you would do it in two steps with a "quiet reference" and you can use it
 for invoking methods, such as .put() to add items to a Map
#set ($myMap2 = {})
$util.qr($myMap2.put("id", "first value"))

## Properties are created with a dot notation
#set($myMap.myProperty = "ABC")
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])
#set($myMap.jsonProperty = {
    "AppSync" : "Offline and Realtime",
    "Cognito" : "AuthN and AuthZ"
})

##When you are inside a string and just have ${} without ! it means stuff inside curly
 braces are a reference
#set($firstname = "Jeff")
$util.qr($myMap.put("Firstname", "${firstname}"))

#set($bigstring = "This is a long string, I want to pull out everything after the
 comma")
#set ($comma = $bigstring.indexOf(','))
#set ($comma = $comma +2)
#set ($substring = $bigstring.substring($comma))
$util.qr($myMap.put("substring", "${substring}"))

##Classic for-each loop over N items:
#set($start = 0)
#set($end = 5)
#set($range = [$start..$end])
#foreach($i in $range)          ##Can also use range operator directly like
 #foreach($item in [1...5])
   ##$util.qr($myMap.put($i, "abc"))
   ##$util.qr($myMap.put($i, $i.toString()+"foo")) ##Concat variable with string
   $util.qr($myMap.put($i, "${i}foo"))     ##Reference a variable in a string with
 "${varname)"
#end
```

```
##Operators don't work
#set($x = 5)
#set($y = 7)
#set($z = $x + $y)
#set($x-y = $x - $y)
#set($xy = $x * $y)
#set($xDIVy = $x / $y)
#set($xMODy = $x % $y)
$util.qr($myMap.put("z", $z))
$util.qr($myMap.put("x-y", $x-y))
$util.qr($myMap.put("x*y", $xy))
$util.qr($myMap.put("x/y", $xDIVy))
$util.qr($myMap.put("x|y", $xMODy))

##arrays
#set($array = ["first"])
#set($idx = 0)
$util.qr($myMap.put("array", $array[$idx]))
##initialize array vals on create
#set($arr2 = [42, "a string", 21, "test"])
$util.qr($myMap.put("arr2", $arr2[$idx]))
$util.qr($myMap.put("isEmpty", $array.isEmpty()))  ##Returns false
$util.qr($myMap.put("size", $array.size()))
##Get and set items in an array
$util.qr($myMap.put("set", $array.set(0, 'changing array value')))
$util.qr($myMap.put("get", $array.get(0)))

##Lookup by name from a Map/dictionary in a similar way:
#set($result = {
    "Author" : "Nadia",
    "Topic" : "GraphQL"
})
$util.qr($myMap.put("Author", $result["Author"]))


##Conditional examples
#if(!$array.isEmpty())
$util.qr($myMap.put("ifCheck", "Array not empty"))
#else
$util.qr($myMap.put("ifCheck", "Your array is empty"))
#end

#if ($arr2.size() == 0)
```

```
$util.qr($myMap.put("elseIfCheck", "You forgot to put anything into this array!"))
#elseif ($arr2.size() == 1)
$util.qr($myMap.put("elseIfCheck", "Good start but please add more stuff"))
#else
$util.qr($myMap.put("elseIfCheck", "Good job!"))
#end

##Above showed negation(!) and equality (==), we can also use OR, AND, >, <, >=, <=,
 and !=
#set($T = true)
#set($F = false)
#if ($T || $F)
  $util.qr($myMap.put("OR", "TRUE"))
#end

#if ($T && $F)
  $util.qr($myMap.put("AND", "TRUE"))
#end

##Using the foreach loop counter - $foreach.count
#foreach ($item in $arr2)
  #set($idx = "item" + $foreach.count)
  $util.qr($myMap.put($idx, $item))
#end

##Using a Map and plucking out keys/vals
#set($hashmap = {
    "DynamoDB" : "https://aws.amazon.com/dynamodb/",
    "Amplify" : "https://github.com/aws/aws-amplify",
    "DynamoDB2" : "https://aws.amazon.com/dynamodb/",
    "Amplify2" : "https://github.com/aws/aws-amplify"
})

#foreach ($key in $hashmap.keySet())
    #if($foreach.count > 2)
        #break
    #end
    $util.qr($myMap.put($key, $hashmap.get($key)))
#end

##concatenate strings
#set($s1 = "Hello")
#set($s2 = " World")
$util.qr($myMap.put("concat","$s1$s2"))
```

```
$util.qr($myMap.put("concat2","Second $s1 World"))

$util.qr($myMap.put("context", $context))

{
    "version" : "2017-02-28",
    "operation": "Invoke",
    "payload": $util.toJson($myMap)
}
```

Modèle de réponse

```
#set($data = {
"id" : "456"
})
#foreach($item in $context.result.entrySet())   ##$context.result is a MAP so we use
 entrySet()
    #if($item.key == "ifCheck")
        $util.qr($data.put("meta", "$item.value"))
    #end
#end

##Uncomment this out if you want to test and remove the below #if check
##$util.toJson($data)

#if($context.result["id"] == 123)
    $util.toJson($context.result)
  #else
    $util.unauthorized()
#end
```

Référence contextuelle du modèle de mappage Resolver

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

AWS AppSyncdéfinit un ensemble de variables et de fonctions permettant de travailler avec des modèles de mappage de résolveurs. Cela facilite les opérations logiques sur les données avec GraphQL. Le présent document décrit ces fonctions et fournit des exemples d'utilisation des modèles.

Accès à la variable `$context`

La variable `$context` est un mappage qui contient toutes les informations contextuelles pour l'appel de votre résolveur. Elle présente la structure suivante :

```
{
   "arguments" : { ... },
   "source" : { ... },
   "result" : { ... },
   "identity" : { ... },
   "request" : { ... },
   "info": { ... }
}
```

> **Note**
>
> Si vous essayez d'accéder à une entrée de dictionnaire/de carte (telle qu'une entrée dans`context`) à l'aide de sa clé pour récupérer la valeur, le Velocity Template Language (VTL) vous permet d'utiliser directement la notation. `<dictionary-element>.<key-name>` Toutefois, cela peut ne pas fonctionner dans tous les cas, notamment lorsque les noms des clés comprennent des caractères spéciaux (par exemple, un caractère de soulignement « _ »). Nous vous recommandons de toujours utiliser la notation `<dictionary-element>.get("<key-name>")`.

Chaque champ dans le mappage `$context` est défini comme suit :

Champs de `$context`

`arguments`

Carte qui contient tous les arguments GraphQL pour ce champ.

`identity`

Objet qui contient des informations sur l'appelant. Pour en savoir plus sur la structure de ce champ, consultez Identité.

`source`

Carte contenant la résolution du champ parent.

`stash`

Le stash est une carte disponible dans chaque modèle de mappage de résolveur et de fonction. La même instance stash perdure pendant une exécution de résolveur. Cela signifie que vous pouvez utiliser le stash pour transmettre des données arbitraires sur des modèles de mappage de demande et de réponse, et sur des fonctions dans un résolveur de pipeline. Le stash expose les mêmes méthodes que la structure de données Java Map.

`result`

Un conteneur pour les résultats de ce résolveur. Ce champ n'est disponible que pour les modèles de mappage des réponses.

Par exemple, si vous résolvez le `author` champ de la requête suivante :

```
query {
    getPost(id: 1234) {
        postId
        title
        content
        author {
            id
            name
        }
    }
}
```

Ensuite, la variable `$context` complète, disponible lors du traitement d'un modèle de mappage de réponse, peut être :

```
{
  "arguments" : {
    id: "1234"
  },
  "source": {},
  "result" : {
      "postId": "1234",
```

```
      "title": "Some title",
      "content": "Some content",
      "author": {
        "id": "5678",
        "name": "Author Name"
      }
  },
  "identity" : {
      "sourceIp" : ["x.x.x.x"],
      "userArn" : "arn:aws:iam::123456789012:user/appsync",
      "accountId" : "666666666666",
      "user" : "AIDAAAAAAAAAAAAAAAAA"
  }
}
```

`prev.result`

Le résultat de toute opération précédente exécutée dans un résolveur de pipeline.

Si l'opération précédente était le modèle de mappage Before du résolveur de pipeline, elle `$ctx.prev.result` représente le résultat de l'évaluation du modèle et est mise à la disposition de la première fonction du pipeline.

Si l'opération précédente est la première fonction, alors `$ctx.prev.result` représente le résultat de la première fonction et il est disponible pour la seconde fonction du pipeline.

Si l'opération précédente était la dernière fonction, elle `$ctx.prev.result` représente la sortie de la dernière fonction et est mise à la disposition du modèle After mapping du résolveur de pipeline.

`info`

Objet qui contient des informations sur la demande GraphQL. Pour obtenir la structure de ce champ, veuillez consulter Infos.

Identity

La section `identity` contient les informations sur l'appelant. La forme de cette section dépend du type d'autorisation de votre API AWS AppSync.

Pour plus d'informations sur les options AWS AppSync de sécurité, consultez la section Autorisation et authentification.

Autorisation **`API_KEY`**

Le `identity` champ n'est pas renseigné.

Autorisation **`AWS_LAMBDA`**

`identity`Il contient la `resolverContext` clé, contenant le même `resolverContext` contenu renvoyé par la fonction Lambda autorisant la demande.

Autorisation **`AWS_IAM`**

`identity`Il a la forme suivante :

```
{
    "accountId" : "string",
    "cognitoIdentityPoolId" : "string",
    "cognitoIdentityId" : "string",
    "sourceIp" : ["string"],
    "username" : "string", // IAM user principal
    "userArn" : "string",
    "cognitoIdentityAuthType" : "string", // authenticated/unauthenticated based on
 the identity type
    "cognitoIdentityAuthProvider" : "string" // the auth provider that was used to
 obtain the credentials
}
```

Autorisation **`AMAZON_COGNITO_USER_POOLS`**

`identity`Il a la forme suivante :

```
{
    "sub" : "uuid",
    "issuer" : "string",
    "username" : "string"
    "claims" : { ... },
    "sourceIp" : ["x.x.x.x"],
    "defaultAuthStrategy" : "string"
}
```

Chaque champ est défini comme suit :

`accountId`

L'identifiant du AWS compte de l'appelant.

`claims`

Demandes de l'utilisateur.

`cognitoIdentityAuthType`

Authentifié ou non authentifié en fonction du type d'identité.

`cognitoIdentityAuthProvider`

Liste séparée par des virgules des informations du fournisseur d'identité externe utilisées pour obtenir les informations d'identification utilisées pour signer la demande.

`cognitoIdentityId`

L'identifiant Amazon Cognito de l'appelant.

`cognitoIdentityPoolId`

L'ID du pool d'identités Amazon Cognito associé à l'appelant.

`defaultAuthStrategy`

Stratégie d'autorisation par défaut pour cet appelant (`ALLOW` ou `DENY`).

`issuer`

Émetteur du jeton.

`sourceIp`

Adresse IP source de l'appelant qui AWS AppSync reçoit. Si la demande n'inclut pas l'`x-forwarded-for`en-tête, la valeur IP source ne contient qu'une seule adresse IP provenant de la connexion TCP. Si la demande inclut un en-tête `x-forwarded-for`, l'adresse IP source est une liste d'adresses IP de l'en-tête `x-forwarded-for` en plus de l'adresse IP de la connexion TCP.

`sub`

UUID de l'utilisateur authentifié.

`user`

Utilisateur IAM.

`userArn`

Le nom de ressource Amazon (ARN) de l'utilisateur IAM.

`username`

Nom de l'utilisateur authentifié. En cas d'autorisation `AMAZON_COGNITO_USER_POOLS`, la valeur de username est la valeur de l'attribut cognito:username. Dans le cas d'`AWS_IAM`une autorisation, la valeur du nom d'utilisateur est la valeur de l'AWSutilisateur principal. Si vous utilisez l'autorisation IAM avec des informations d'identification provenant de pools d'identités Amazon Cognito, nous vous recommandons d'utiliser. `cognitoIdentityId`

En-têtes de demande d'accès

AWS AppSync permet de transmettre des en-têtes personnalisés depuis les clients et d'y accéder dans vos résolveurs GraphQL en utilisant. `$context.request.headers` Vous pouvez ensuite utiliser les valeurs d'en-tête pour des actions telles que l'insertion de données dans une source de données ou les contrôles d'autorisation. Vous pouvez utiliser un ou plusieurs en-têtes de demande à l'`$curl`aide d'une clé d'API depuis la ligne de commande, comme indiqué dans les exemples suivants :

Exemple d'en-tête unique

Supposons que vous définissiez un en-tête `custom` avec la valeur `nadia`, comme suit :

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:nadia" -H "x-api-key:<API-
KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo\", when: \"Next Friday!
\", where: \"Here!\") {id name when where description}}"}' https://<ENDPOINT>/graphql
```

Il est alors possible d'y accéder avec `$context.request.headers.custom`. Par exemple, il peut se trouver dans la VTL suivante pour DynamoDB :

```
"custom": $util.dynamodb.toDynamoDBJson($context.request.headers.custom)
```

Exemple d'en-tête multiple

Vous pouvez également transmettre plusieurs en-têtes dans une seule demande et y accéder dans le modèle de mappage des résolveurs. Par exemple, si l'`custom`en-tête est défini avec deux valeurs :

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:bailey" -H "custom:nadia"
 -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo
\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}"}'
 https://<ENDPOINT>/graphql
```

Vous pouvez ensuite y accéder comme s'il s'agissait d'un tableau : par exemple, `$context.request.headers.custom[1]`.

> **Note**
>
> AWS AppSyncn'expose pas l'en-tête du cookie dans`$context.request.headers`.

Accédez au nom de domaine personnalisé de la demande

AWS AppSyncprend en charge la configuration d'un domaine personnalisé que vous pouvez utiliser pour accéder à votre GraphQL et aux points de terminaison en temps réel de vos API. Lorsque vous faites une demande avec un nom de domaine personnalisé, vous pouvez obtenir le nom de domaine en utilisant`$context.request.domainName`.

Lorsque vous utilisez le nom de domaine du point de terminaison GraphQL par défaut, la valeur est. `null`

Infos

La section `info` contient des informations sur la demande GraphQL. Cette section se présente sous la forme suivante :

```
{
    "fieldName": "string",
    "parentTypeName": "string",
    "variables": { ... },
    "selectionSetList": ["string"],
    "selectionSetGraphQL": "string"
}
```

Chaque champ est défini comme suit :

`fieldName`

Nom du champ en cours de résolution.

`parentTypeName`

Nom du type parent du champ en cours de résolution.

`variables`

Carte qui contient toutes les variables transmises dans la demande GraphQL.

`selectionSetList`

Représentation sous forme de liste des champs du jeu de sélection GraphQL. Les champs dotés d'un alias sont référencés uniquement par le nom de l'alias, et non par le nom du champ. L'exemple suivant détaille cela.

`selectionSetGraphQL`

Représentation sous forme de chaîne du jeu de sélection, formatée en langage SDL GraphQL. Bien que les fragments ne soient pas fusionnés dans le jeu de sélection, les fragments intégrés sont conservés, comme le montre l'exemple suivant.

> **Note**
>
> Lorsque vous utilisez `$utils.toJson()` on`context.info`, les valeurs renvoyées `selectionSetGraphQL` et celles `selectionSetList` renvoyées ne sont pas sérialisées par défaut.

Par exemple, si vous résolvez le champ `getPost` de la requête suivante :

```
query {
  getPost(id: $postId) {
    postId
    title
    secondTitle: title
    content
    author(id: $authorId) {
      authorId
      name
    }
    secondAuthor(id: "789") {
      authorId
    }
    ... on Post {
      inlineFrag: comments: {
        id
      }
```

```
    }
    ... postFrag
  }
}

fragment postFrag on Post {
  postFrag: comments: {
    id
  }
}
```

Ensuite, la variable `$context.info` complète, disponible lors du traitement d'un modèle de mappage, peut être :

```
{
  "fieldName": "getPost",
  "parentTypeName": "Query",
  "variables": {
    "postId": "123",
    "authorId": "456"
  },
  "selectionSetList": [
    "postId",
    "title",
    "secondTitle"
    "content",
    "author",
    "author/authorId",
    "author/name",
    "secondAuthor",
    "secondAuthor/authorId",
    "inlineFragComments",
    "inlineFragComments/id",
    "postFragComments",
    "postFragComments/id"
  ],
  "selectionSetGraphQL": "{\n  getPost(id: $postId) {\n    postId\n    title\n
 secondTitle: title\n    content\n    author(id: $authorId) {\n      authorId\n
 name\n    }\n    secondAuthor(id: \"789\") {\n      authorId\n    }\n    ... on Post
 {\n      inlineFrag: comments {\n        id\n      }\n    }\n    ... postFrag\n  }\n}"
}
```

`selectionSetListn`'expose que les champs appartenant au type actuel. Si le type actuel est une interface ou une union, seuls les champs sélectionnés appartenant à l'interface sont exposés. Par exemple, selon le schéma suivant :

```
type Query {
    node(id: ID!): Node
}

interface Node {
    id: ID
}

type Post implements Node {
    id: ID
    title: String
    author: String
}

type Blog implements Node {
    id: ID
    title: String
    category: String
}
```

Et la requête suivante :

```
query {
    node(id: "post1") {
        id
        ... on Post {
            title
        }

        ... on Blog {
            title
        }
    }
}
```

Lors de l'appel `$ctx.info.selectionSetList` à la résolution du `Query.node` champ, seul `id` est exposé :

```
"selectionSetList": [
    "id"
]
```

Assainissement des entrées

Les applications doivent assainir les entrées non approuvées pour empêcher toute utilisation non prévue de ces applications par une tierce partie. Comme il `$context` contient des entrées utilisateur dans des propriétés telles que`$context.arguments`,`$context.identity`,`$context.result`,`$context.info.variables`, et`$context.request.headers`, il faut veiller à nettoyer leurs valeurs dans les modèles de mappage.

Étant donné que les modèles de mappage représentent JSON, l'assainissement des entrées consiste à échapper les caractères réservés JSON des chaînes représentant les entrées utilisateur. Il est recommandé d'utiliser l'utilitaire `$util.toJson()` pour échapper les caractères réservés JSON des valeurs de chaîne sensibles lors de leur insertion dans un modèle de mappage.

Par exemple, dans le modèle de mappage de requêtes Lambda suivant, parce que nous avons accédé à une chaîne de saisie client non sécurisée (`$context.arguments.id`), nous l'avons encapsulée `$util.toJson()` pour empêcher les caractères JSON non échappés de casser le modèle JSON.

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": $util.toJson($context.arguments.id)
    }
}
```

Contrairement au modèle de mappage ci-dessous, où nous insérons directement `$context.arguments.id` sans désinfection. Cela ne fonctionne pas pour les chaînes contenant des guillemets non échappés ou d'autres caractères réservés JSON, et cela peut entraîner l'échec de votre modèle.

```
## DO NOT DO THIS
{
    "version": "2017-02-28",
```

```
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "$context.arguments.id" ## Unsafe! Do not insert $context string
 values without escaping JSON characters.
    }
}
```

Référence de l'utilitaire du modèle de mappage Resolver

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

AWS AppSyncdéfinit un ensemble d'utilitaires que vous pouvez utiliser dans un résolveur GraphQL pour simplifier les interactions avec les sources de données. Certains de ces utilitaires sont destinés à une utilisation générale avec n'importe quelle source de données, comme la génération d'identifiants ou d'horodatages. D'autres sont spécifiques à un type de source de données.

Rubriques

Aides utilitaires dans $util

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

La `$util` variable contient des méthodes utilitaires générales pour vous aider à travailler avec les données. Sauf indication contraire, tous les utilitaires emploient le jeu de caractères UTF-8.

Utilitaires d'analyse JSON

Liste des utilitaires d'analyse JSON

`$util.parseJson(String) : Object`

Prend la chaîne JSON (obtenue à l'aide de stringify) et renvoie une représentation objet du résultat.

`$util.toJson(Object) : String`

Accepte un objet et retourne une représentation JSON de cet objet obtenue à l'aide de stringify.

Utilitaires d'encodage

Liste des utilitaires d'encodage

`$util.urlEncode(String) : String`

Renvoie la chaîne en entrée sous la forme d'une chaîne codée `application/x-www-form-urlencoded`.

`$util.urlDecode(String) : String`

Décode une chaîne codée `application/x-www-form-urlencoded` sous sa forme initiale non codée.

`$util.base64Encode( byte[] ) : String`

Code les données d'entrée en une chaîne codée en base64.

`$util.base64Decode(String) : byte[]`

Décode les données d'une chaîne encodée en base64.

Utilitaires de génération d'identifiants

Liste des utilitaires de génération d'identifiants

`$util.autoId() : String`

Renvoie un UUID 128 bits généré de façon aléatoire.

`$util.autoUlid() : String`

Renvoie un ULID (identifiant lexicographiquement sortable universel unique) de 128 bits généré aléatoirement.

`$util.autoKsuid() : String`

Renvoie un KSUID (K-Sortable Unique Identifier) de 128 bits généré aléatoirement en base62 et codé sous forme de chaîne d'une longueur de 27.

Utils d'erreur

Liste des utilitaires d'erreur

`$util.error(String)`

Lève une erreur personnalisée. Utilisez-le dans les modèles de mappage de demandes ou de réponses pour détecter une erreur dans la demande ou dans le résultat de l'appel.

`$util.error(String, String)`

Lève une erreur personnalisée. Utilisez-le dans les modèles de mappage de demandes ou de réponses pour détecter une erreur dans la demande ou dans le résultat de l'appel. Vous pouvez également spécifier un`errorType`.

`$util.error(String, String, Object)`

Lève une erreur personnalisée. Utilisez-le dans les modèles de mappage de demandes ou de réponses pour détecter une erreur dans la demande ou dans le résultat de l'appel. Vous pouvez également spécifier un `errorType` et un `data` champ. La valeur `data` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL.

> **Note**
>
> `data`sera filtré en fonction de l'ensemble de sélection de requêtes.

`$util.error(String, String, Object, Object)`

Lève une erreur personnalisée. Peut être utilisé dans les modèles de mappage de demande ou de réponse si le modèle détecte une erreur associée à la demande ou au résultat de l'appel. En outre, un `errorType` champ, un `data` champ et un `errorInfo` champ peuvent être spécifiés. La valeur `data` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL.

> **Note**
>
> `data`sera filtré en fonction de l'ensemble de sélection de requêtes. La valeur `errorInfo` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL.
> `errorInf`one sera PAS filtré en fonction de l'ensemble de sélection de requêtes.

`$util.appendError(String)`

Ajoute une erreur personnalisée. Peut être utilisé dans les modèles de mappage de demande ou de réponse si le modèle détecte une erreur associée à la demande ou au résultat de l'appel. Contrairement à `$util.error(String)`, l'évaluation du modèle n'est pas interrompue et, par conséquent, les données peuvent être retournées à l'appelant.

`$util.appendError(String, String)`

Ajoute une erreur personnalisée. Peut être utilisé dans les modèles de mappage de demande ou de réponse si le modèle détecte une erreur associée à la demande ou au résultat de l'appel. En outre, il est possible de spécifier un champ `errorType`. Contrairement à `$util.error(String, String)`, l'évaluation du modèle n'est pas interrompue et, par conséquent, les données peuvent être retournées à l'appelant.

`$util.appendError(String, String, Object)`

Ajoute une erreur personnalisée. Peut être utilisé dans les modèles de mappage de demande ou de réponse si le modèle détecte une erreur associée à la demande ou au résultat de l'appel. Il

est également possible de spécifier un champ `errorType` et un champ `data`. Contrairement à `$util.error(String, String, Object)`, l'évaluation du modèle n'est pas interrompue et, par conséquent, les données peuvent être retournées à l'appelant. La valeur `data` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL.

> **Note**
>
> `data`sera filtré en fonction de l'ensemble de sélection de requêtes.

`$util.appendError(String, String, Object, Object)`

Ajoute une erreur personnalisée. Peut être utilisé dans les modèles de mappage de demande ou de réponse si le modèle détecte une erreur associée à la demande ou au résultat de l'appel. En outre, un `errorType` champ, un `data` champ et un `errorInfo` champ peuvent être spécifiés. Contrairement à `$util.error(String, String, Object, Object)`, l'évaluation du modèle n'est pas interrompue et, par conséquent, les données peuvent être retournées à l'appelant. La valeur `data` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL.

> **Note**
>
> `data`sera filtré en fonction de l'ensemble de sélection de requêtes. La valeur `errorInfo` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL.
> `errorInf`one sera PAS filtré en fonction de l'ensemble de sélection de requêtes.

Utilitaires de validation des conditions

Liste des utilitaires de validation des conditions

`$util.validate(Boolean, String) : void`

Si la condition est fausse, lancez un CustomTemplateException avec le message spécifié.

`$util.validate(Boolean, String, String) : void`

Si la condition est fausse, lancez un CustomTemplateException avec le message et le type d'erreur spécifiés.

```
$util.validate(Boolean, String, String, Object) : void
```

Si la condition est fausse, lancez un CustomTemplateException avec le message et le type d'erreur spécifiés, ainsi que les données à renvoyer dans la réponse.

Utilitaires de comportement nuls

Liste d'utilitaires de comportement nulle

```
$util.isNull(Object) : Boolean
```

Renvoie la valeur true si l'objet fourni a la valeur null.

```
$util.isNullOrEmpty(String) : Boolean
```

Renvoie la valeur true si les données fournies ont une valeur null ou une chaîne vide. Sinon, la valeur renvoyée est false.

```
$util.isNullOrBlank(String) : Boolean
```

Renvoie la valeur true si les données fournies ont une valeur null ou une chaîne vide. Sinon, la valeur renvoyée est false.

```
$util.defaultIfNull(Object, Object) : Object
```

Renvoie le premier objet si la valeur n'est pas null. Sinon, renvoie le deuxième objet en tant qu'« objet par défaut ».

```
$util.defaultIfNullOrEmpty(String, String) : String
```

Renvoie la première chaîne si la valeur n'est pas null ou vide. Sinon, renvoie la deuxième chaîne en tant que « chaîne (string) par défaut ».

```
$util.defaultIfNullOrBlank(String, String) : String
```

Renvoie la première chaîne si la valeur n'est pas null ou vide. Sinon, renvoie la deuxième chaîne en tant que « chaîne (string) par défaut ».

Utilitaires de correspondance de modèles

Liste d'utilitaires correspondant au type et au modèle

`$util.typeOf(Object) : String`

Renvoie une chaîne décrivant le type de l'objet. Identifications de type prises en charge : « Null », « Number », « String », « Map », « List », « Boolean ». Si un type ne peut pas être identifié, le type de retour est « Object ».

`$util.matches(String, String) : Boolean`

Renvoie la valeur true si le modèle spécifié dans le premier argument correspond aux données fournies dans le deuxième argument. Le modèle doit être une expression régulière, telle que `$util.matches("a*b", "aaaaab")`. La fonctionnalité est basée sur Pattern, que vous pouvez référencer à titre de documentation ultérieure.

`$util.authType() : String`

Renvoie une chaîne décrivant le type d'authentification multiple utilisé par une demande, renvoyant soit « IAM Authorization », « User Pool Authorization », « Open ID Connect Authorization », soit « API Key Authorization ».

Utilitaires de validation d'objets

Liste des utilitaires de validation d'objets

`$util.isString(Object) : Boolean`

Renvoie vrai si l'objet est une chaîne.

`$util.isNumber(Object) : Boolean`

Renvoie vrai si l'objet est un nombre.

`$util.isBoolean(Object) : Boolean`

Renvoie vrai si l'objet est un booléen.

`$util.isList(Object) : Boolean`

Renvoie vrai si l'objet est une liste.

`$util.isMap(Object) : Boolean`

Renvoie la valeur true si l'objet est une carte.

CloudWatch utilitaires de journalisation

CloudWatch liste des utilitaires de journalisation

`$util.log.info(Object) : Void`

Enregistre la représentation sous forme de chaîne de l'objet fourni dans le flux de journal demandé lorsque la journalisation au niveau de la demande et au niveau du champ est activée avec le niveau de CloudWatch journalisation sur une API. `ALL`

`$util.log.info(String, Object...) : Void`

Enregistre la représentation sous forme de chaîne des objets fournis dans le flux de journal demandé lorsque la journalisation au niveau de la demande et au niveau du champ est activée avec le niveau de CloudWatch journalisation sur une API. `ALL` Cet utilitaire remplacera toutes les variables indiquées par « {} » dans le premier format d'entrée String par la représentation sous forme de chaîne des objets fournis dans l'ordre.

`$util.log.error(Object) : Void`

Enregistre la représentation sous forme de chaîne de l'objet fourni dans le flux de journal demandé lorsque la CloudWatch journalisation au niveau du champ est activée avec le niveau du journal `ERROR` ou le niveau du journal `ALL` sur une API.

`$util.log.error(String, Object...) : Void`

Enregistre la représentation sous forme de chaîne des objets fournis dans le flux de journal demandé lorsque la CloudWatch journalisation au niveau du champ est activée avec le niveau du journal `ERROR` ou le niveau du journal `ALL` sur une API. Cet utilitaire remplacera toutes les variables indiquées par « {} » dans le premier format d'entrée String par la représentation sous forme de chaîne des objets fournis dans l'ordre.

Utilitaires de comportement des valeurs renvoyées

Liste des utilitaires relatifs au comportement des valeurs renvoyées

`$util.qr()` et `$util.quiet()`

Exécute une instruction VTL tout en supprimant la valeur renvoyée. Cela est utile pour exécuter des méthodes sans utiliser d'espaces réservés temporaires, tels que l'ajout d'éléments à une carte. Par exemple :

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

Devient :

```
#set ($myMap = {})
$util.qr($myMap.put("id", "first value"))
```

$util.escapeJavaScript(String) : String

Renvoie la chaîne d'entrée sous forme de chaîne JavaScript échappée.

$util.urlEncode(String) : String

Renvoie la chaîne en entrée sous la forme d'une chaîne codée `application/x-www-form-urlencoded`.

$util.urlDecode(String) : String

Décode une chaîne codée `application/x-www-form-urlencoded` sous sa forme initiale non codée.

$util.base64Encode(byte[]) : String

Code les données d'entrée en une chaîne codée en base64.

$util.base64Decode(String) : byte[]

Décode les données d'une chaîne encodée en base64.

$util.parseJson(String) : Object

Prend la chaîne JSON (obtenue à l'aide de stringify) et renvoie une représentation objet du résultat.

$util.toJson(Object) : String

Accepte un objet et retourne une représentation JSON de cet objet obtenue à l'aide de stringify.

$util.autoId() : String

Renvoie un UUID 128 bits généré de façon aléatoire.

`$util.autoUlid() : String`

Renvoie un ULID (identifiant lexicographiquement sortable universel unique) de 128 bits généré aléatoirement.

`$util.autoKsuid() : String`

Renvoie un KSUID (K-Sortable Unique Identifier) de 128 bits généré aléatoirement en base62 et codé sous forme de chaîne d'une longueur de 27.

`$util.unauthorized()`

Lève `Unauthorized` pour le champ en cours de résolution. Utilisez-le dans les modèles de mappage de demandes ou de réponses pour déterminer s'il convient d'autoriser l'appelant à résoudre le champ.

`$util.error(String)`

Lève une erreur personnalisée. Utilisez-le dans les modèles de mappage de demandes ou de réponses pour détecter une erreur dans la demande ou dans le résultat de l'appel.

`$util.error(String, String)`

Lève une erreur personnalisée. Utilisez-le dans les modèles de mappage de demandes ou de réponses pour détecter une erreur dans la demande ou dans le résultat de l'appel. Vous pouvez également spécifier un`errorType`.

`$util.error(String, String, Object)`

Lève une erreur personnalisée. Utilisez-le dans les modèles de mappage de demandes ou de réponses pour détecter une erreur dans la demande ou dans le résultat de l'appel. Vous pouvez également spécifier un `errorType` et un `data` champ. La valeur `data` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL. Remarque : `data` est filtré en fonction du jeu de sélection de la requête.

`$util.error(String, String, Object, Object)`

Lève une erreur personnalisée. Peut être utilisé dans les modèles de mappage de demande ou de réponse si le modèle détecte une erreur associée à la demande ou au résultat de l'appel. En outre, il est possible de spécifier un champ `errorType`, un champ `data` et un champ `errorInfo`. La valeur `data` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL. Remarque : `data` est filtré en fonction du jeu de sélection de la requête. La valeur `errorInfo` sera ajoutée au bloc `error` correspondant à

l'intérieur d'`errors` dans la réponse GraphQL. Remarque : `errorInfo` N'EST PAS filtré en fonction du jeu de sélection de la requête.

`$util.appendError(String)`

Ajoute une erreur personnalisée. Peut être utilisé dans les modèles de mappage de demande ou de réponse si le modèle détecte une erreur associée à la demande ou au résultat de l'appel. Contrairement à `$util.error(String)`, l'évaluation du modèle n'est pas interrompue et, par conséquent, les données peuvent être retournées à l'appelant.

`$util.appendError(String, String)`

Ajoute une erreur personnalisée. Peut être utilisé dans les modèles de mappage de demande ou de réponse si le modèle détecte une erreur associée à la demande ou au résultat de l'appel. En outre, il est possible de spécifier un champ `errorType`. Contrairement à `$util.error(String, String)`, l'évaluation du modèle n'est pas interrompue et, par conséquent, les données peuvent être retournées à l'appelant.

`$util.appendError(String, String, Object)`

Ajoute une erreur personnalisée. Peut être utilisé dans les modèles de mappage de demande ou de réponse si le modèle détecte une erreur associée à la demande ou au résultat de l'appel. Il est également possible de spécifier un champ `errorType` et un champ `data`. Contrairement à `$util.error(String, String, Object)`, l'évaluation du modèle n'est pas interrompue et, par conséquent, les données peuvent être retournées à l'appelant. La valeur `data` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL. Remarque : `data` est filtré en fonction du jeu de sélection de la requête.

`$util.appendError(String, String, Object, Object)`

Ajoute une erreur personnalisée. Peut être utilisé dans les modèles de mappage de demande ou de réponse si le modèle détecte une erreur associée à la demande ou au résultat de l'appel. En outre, il est possible de spécifier un champ `errorType`, un champ `data` et un champ `errorInfo`. Contrairement à `$util.error(String, String, Object, Object)`, l'évaluation du modèle n'est pas interrompue et, par conséquent, les données peuvent être retournées à l'appelant. La valeur `data` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL. Remarque : `data` est filtré en fonction du jeu de sélection de la requête. La valeur `errorInfo` sera ajoutée au bloc `error` correspondant à l'intérieur d'`errors` dans la réponse GraphQL. Remarque : `errorInfo` N'EST PAS filtré en fonction du jeu de sélection de la requête.

`$util.validate(Boolean, String) : void`

Si la condition est fausse, lancez un CustomTemplateException avec le message spécifié.

`$util.validate(Boolean, String, String) : void`

Si la condition est fausse, lancez un CustomTemplateException avec le message et le type d'erreur spécifiés.

`$util.validate(Boolean, String, String, Object) : void`

Si la condition est fausse, lancez un CustomTemplateException avec le message et le type d'erreur spécifiés, ainsi que les données à renvoyer dans la réponse.

`$util.isNull(Object) : Boolean`

Renvoie la valeur true si l'objet fourni a la valeur null.

`$util.isNullOrEmpty(String) : Boolean`

Renvoie la valeur true si les données fournies ont une valeur null ou une chaîne vide. Sinon, la valeur renvoyée est false.

`$util.isNullOrBlank(String) : Boolean`

Renvoie la valeur true si les données fournies ont une valeur null ou une chaîne vide. Sinon, la valeur renvoyée est false.

`$util.defaultIfNull(Object, Object) : Object`

Renvoie le premier objet si la valeur n'est pas null. Sinon, renvoie le deuxième objet en tant qu'« objet par défaut ».

`$util.defaultIfNullOrEmpty(String, String) : String`

Renvoie la première chaîne si la valeur n'est pas null ou vide. Sinon, renvoie la deuxième chaîne en tant que « chaîne (string) par défaut ».

`$util.defaultIfNullOrBlank(String, String) : String`

Renvoie la première chaîne si la valeur n'est pas null ou vide. Sinon, renvoie la deuxième chaîne en tant que « chaîne (string) par défaut ».

`$util.isString(Object) : Boolean`

Renvoie la valeur true si l'objet est une chaîne.

`$util.isNumber(Object) : Boolean`

Renvoie la valeur true si l'objet est un nombre.

`$util.isBoolean(Object) : Boolean`

Renvoie la valeur true si l'objet est une valeur booléenne.

`$util.isList(Object) : Boolean`

Renvoie la valeur true si l'objet est une liste.

`$util.isMap(Object) : Boolean`

Renvoie la valeur true si l'objet est une map.

`$util.typeOf(Object) : String`

Renvoie une chaîne décrivant le type de l'objet. Identifications de type prises en charge : « Null », « Number », « String », « Map », « List », « Boolean ». Si un type ne peut pas être identifié, le type de retour est « Object ».

`$util.matches(String, String) : Boolean`

Renvoie la valeur true si le modèle spécifié dans le premier argument correspond aux données fournies dans le deuxième argument. Le modèle doit être une expression régulière, telle que `$util.matches("a*b", "aaaaab")`. La fonctionnalité est basée sur Pattern, que vous pouvez référencer à titre de documentation ultérieure.

`$util.authType() : String`

Renvoie une chaîne décrivant le type d'authentification multiple utilisé par une demande, renvoyant soit « IAM Authorization », « User Pool Authorization », « Open ID Connect Authorization », soit « API Key Authorization ».

`$util.log.info(Object) : Void`

Enregistre la représentation sous forme de chaîne de l'objet fourni dans le flux de journal demandé lorsque la journalisation au niveau de la demande et au niveau du champ est activée avec le niveau de CloudWatch journalisation sur une API. `ALL`

`$util.log.info(String, Object...) : Void`

Enregistre la représentation sous forme de chaîne des objets fournis dans le flux de journal demandé lorsque la journalisation au niveau de la demande et au niveau du champ est activée avec le niveau de CloudWatch journalisation sur une API. `ALL` Cet utilitaire remplacera toutes les variables indiquées par « {} » dans le premier format d'entrée String par la représentation sous forme de chaîne des objets fournis dans l'ordre.

`$util.log.error(Object) : Void`

Enregistre la représentation sous forme de chaîne de l'objet fourni dans le flux de journal demandé lorsque la CloudWatch journalisation au niveau du champ est activée avec le niveau du journal `ERROR` ou le niveau du journal `ALL` sur une API.

`$util.log.error(String, Object...) : Void`

Enregistre la représentation sous forme de chaîne des objets fournis dans le flux de journal demandé lorsque la CloudWatch journalisation au niveau du champ est activée avec le niveau du journal `ERROR` ou le niveau du journal `ALL` sur une API. Cet utilitaire remplacera toutes les variables indiquées par « {} » dans le premier format d'entrée String par la représentation sous forme de chaîne des objets fournis dans l'ordre.

`$util.escapeJavaScript(String) : String`

Renvoie la chaîne d'entrée sous forme de chaîne JavaScript échappée.

Autorisation du résolveur

Liste d'autorisations du résolveur

`$util.unauthorized()`

Lève `Unauthorized` pour le champ en cours de résolution. Utilisez-le dans les modèles de mappage de demandes ou de réponses pour déterminer s'il convient d'autoriser l'appelant à résoudre le champ.

AWS AppSync directives

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

AWS AppSync expose des directives pour faciliter la productivité des développeurs lors de l'écriture en VTL.

Utilitaires de directives

`#return(Object)`

Vous `#return(Object)` permet de revenir prématurément à partir de n'importe quel modèle de mappage. `#return(Object)`est analogue au mot clé return dans les langages de programmation, car il retournera à partir du bloc logique le plus proche. L'utilisation de `#return(Object)` l'intérieur d'un modèle de mappage du résolveur reviendra du résolveur. De plus, l'utilisation `#return(Object)` à partir d'un modèle de mappage de fonctions revient à la fonction et poursuit l'exécution soit vers la fonction suivante du pipeline, soit vers le modèle de mappage de réponse du résolveur.

`#return`

La `#return` directive présente les mêmes comportements que`#return(Object)`, mais elle `null` sera renvoyée à la place.

Des aides temporelles dans $util.time

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

La variable `$util.time` contient les méthodes datetime pour aider à générer les horodatages, à convertir d'un format datetime à l'autre et à analyser les chaînes datetime. La syntaxe des formats datetime est basée sur DateTimeFormatterlaquelle vous pouvez vous référer pour obtenir de la documentation supplémentaire. Nous fournissons quelques exemples ci-dessous, ainsi qu'une liste des méthodes et des descriptions disponibles.

Temps et utilités

Liste des heures et des outils

`$util.time.nowISO8601() : String`

Renvoie une représentation String (chaîne) d'UTC au format ISO8601.

`$util.time.nowEpochSeconds() : long`

Renvoie le nombre de secondes entre l'époque Unix 1970-01-01T00:00:00Z et maintenant.

`$util.time.nowEpochMilliSeconds() : long`

Renvoie le nombre de millisecondes entre l'époque Unix 1970-01-01T00:00:00Z et maintenant.

`$util.time.nowFormatted(String) : String`

Renvoie une chaîne de l'horodatage actuel (UTC) à l'aide du format spécifié à partir d'un type d'entrée String (chaîne).

`$util.time.nowFormatted(String, String) : String`

Renvoie une chaîne de l'horodatage actuel pour un fuseau horaire à l'aide du format et du fuseau spécifiés à partir de types d'entrée String (chaîne).

`$util.time.parseFormattedToEpochMilliSeconds(String, String) : Long`

Analyse un horodatage transmis sous forme de chaîne avec un format, puis renvoie l'horodatage en millisecondes depuis l'époque.

`$util.time.parseFormattedToEpochMilliSeconds(String, String, String) : Long`

Analyse un horodatage transmis sous forme de chaîne avec un format et un fuseau horaire, puis renvoie l'horodatage en millisecondes depuis l'époque.

`$util.time.parseISO8601ToEpochMilliSeconds(String) : Long`

Analyse un horodatage ISO8601 transmis sous forme de chaîne, puis renvoie l'horodatage en millisecondes depuis l'époque.

`$util.time.epochMilliSecondsToSeconds(long) : long`

Convertit une époque Unix en millisecondes en une époque Unix en secondes.

`$util.time.epochMilliSecondsToISO8601(long) : String`

Convertit un horodatage en millisecondes d'époque en un horodatage ISO8601.

`$util.time.epochMilliSecondsToFormatted(long, String) : String`

Convertit l'horodatage d'une époque en millisecondes, transmis sous sa forme la plus longue, en un horodatage formaté selon le format fourni en UTC.

`$util.time.epochMilliSecondsToFormatted(long, String, String) : String`

Convertit un horodatage en millisecondes d'époque, transmis sous forme de long, en un horodatage formaté selon le format fourni dans le fuseau horaire fourni.

Exemples de fonctions autonomes

```
$util.time.nowISO8601()                                              :
 2018-02-06T19:01:35.749Z
$util.time.nowEpochSeconds()                                         : 1517943695
$util.time.nowEpochMilliSeconds()                                    : 1517943695750
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ")                      : 2018-02-06
 19:01:35+0000
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ", "+08:00")            : 2018-02-07
 03:01:35+0800
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ", "Australia/Perth") : 2018-02-07
 03:01:35+0800
```

Exemples de conversion

```
#set( $nowEpochMillis = 1517943695758 )
$util.time.epochMilliSecondsToSeconds($nowEpochMillis)
    : 1517943695
$util.time.epochMilliSecondsToISO8601($nowEpochMillis)
    : 2018-02-06T19:01:35.758Z
$util.time.epochMilliSecondsToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ")
    : 2018-02-06 19:01:35+0000
$util.time.epochMilliSecondsToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ",
 "+08:00") : 2018-02-07 03:01:35+0800
```

Exemples d'analyse syntaxique

```
$util.time.parseISO8601ToEpochMilliSeconds("2018-02-01T17:21:05.180+08:00")
              : 1517476865180
$util.time.parseFormattedToEpochMilliSeconds("2018-02-02 01:19:22+0800", "yyyy-MM-dd
 HH:mm:ssZ")     : 1517505562000
$util.time.parseFormattedToEpochMilliSeconds("2018-02-02 01:19:22", "yyyy-MM-dd
 HH:mm:ss", "+08:00") : 1517505562000
```

Utilisation avec des scalaires AWS AppSync définis

Les formats suivants sont compatibles avec `AWSDate`, `AWSDateTime`, et `AWSTime`.

```
$util.time.nowFormatted("yyyy-MM-dd[XXX]", "-07:00:30")              :
 2018-07-11-07:00
```

```
$util.time.nowFormatted("yyyy-MM-dd'T'HH:mm:ss[XXXXX]", "-07:00:30")  :
 2018-07-11T15:14:15-07:00:30
```

Répertorier les assistants dans $util.list

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

`$util.list`contient des méthodes destinées à faciliter les opérations de liste courantes, telles que la suppression ou la conservation d'éléments d'une liste pour filtrer les cas d'utilisation.

Répertorier les utilitaires

`$util.list.copyAndRetainAll(List, List) : List`

Fait une copie superficielle de la liste fournie dans le premier argument tout en ne conservant que les éléments spécifiés dans le second argument, s'ils sont présents. Tous les autres éléments sont supprimés de la copie.

`$util.list.copyAndRemoveAll(List, List) : List`

Effectue une copie superficielle de la liste fournie dans le premier argument tout en supprimant tous les éléments dont l'élément est spécifié dans le deuxième argument, s'ils sont présents. Tous les autres éléments sont conservés dans la copie.

`$util.list.sortList(List, Boolean, String) : List`

Trie une liste d'objets, qui est fournie dans le premier argument. Si le deuxième argument est vrai, la liste est triée par ordre décroissant ; si le second argument est faux, la liste est triée par ordre croissant. Le troisième argument est le nom de chaîne de la propriété utilisée pour trier une liste d'objets personnalisés. S'il s'agit d'une liste de chaînes, d'entiers, de flottants ou de doubles uniquement, le troisième argument peut être n'importe quelle chaîne aléatoire. Si tous les objets ne sont pas de la même classe, la liste d'origine est renvoyée. Seules les listes contenant un maximum de 1 000 objets sont prises en charge. Voici un exemple d'utilisation de cet utilitaire :

```
 INPUT:      $util.list.sortList([{"description":"youngest", "age":5},
{"description":"middle", "age":45}, {"description":"oldest", "age":85}], false,
 "description")
```

```
OUTPUT:     [{"description":"middle", "age":45}, {"description":"oldest",
"age":85}, {"description":"youngest", "age":5}]
```

Assistants cartographiques dans $util.map

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

`$util.map`contient des méthodes destinées à faciliter les opérations cartographiques courantes, telles que la suppression ou la conservation d'éléments d'une carte pour filtrer les cas d'utilisation.

Utilitaires cartographiques

`$util.map.copyAndRetainAllKeys(Map, List) : Map`

Effectue une copie superficielle de la première carte en ne conservant que les clés spécifiées dans la liste, si elles sont présentes. Toutes les autres clés sont supprimées de la copie.

`$util.map.copyAndRemoveAllKeys(Map, List) : Map`

Effectue une copie superficielle de la première carte tout en supprimant toutes les entrées où la clé est spécifiée dans la liste, si elles sont présentes. Toutes les autres clés sont conservées dans la copie.

Assistants DynamoDB dans $util.dynamodb

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

`$util.dynamodb`contient des méthodes d'assistance qui facilitent l'écriture et la lecture de données dans Amazon DynamoDB, telles que le mappage automatique des types et le formatage. Ces méthodes sont conçues pour mapper automatiquement les types primitifs et les listes au format d'entrée DynamoDB approprié, qui est `Map` un des formats. `{ "TYPE" : VALUE }`

Par exemple, auparavant, un modèle de mappage de demandes destiné à créer un nouvel élément dans DynamoDB aurait pu ressembler à ceci :

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "id" : { "S" : "$util.autoId()" }
    },
    "attributeValues" : {
         "title" : { "S" : $util.toJson($ctx.args.title) },
         "author" : { "S" : $util.toJson($ctx.args.author) },
         "version" : { "N", $util.toJson($ctx.args.version) }
    }
}
```

Si nous voulions ajouter des champs à l'objet, nous devrions mettre à jour la requête GraphQL dans le schéma, ainsi que le modèle de mappage de demande. Cependant, nous pouvons désormais restructurer notre modèle de mappage des demandes afin qu'il récupère automatiquement les nouveaux champs ajoutés dans notre schéma et les ajoute à DynamoDB avec les types appropriés :

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

Dans l'exemple précédent, nous utilisons l'`$util.dynamodb.toDynamoDBJson(...)`assistant pour prendre automatiquement l'identifiant généré et le convertir en représentation DynamoDB d'un attribut de chaîne. Nous prenons ensuite tous les arguments, les convertissons en leurs représentations DynamoDB et les affichons dans `attributeValues` le champ du modèle.

Chaque assistant possède deux versions : une version qui renvoie un objet (par exemple, `$util.dynamodb.toString(...)`) et une version qui renvoie l'objet en tant que chaîne JSON (par exemple, `$util.dynamodb.toStringJson(...)`). Dans l'exemple précédent, nous avons utilisé la version qui renvoie les données sous la forme d'une chaîne JSON. Si vous souhaitez

manipuler l'objet avant qu'il ne soit utilisé dans le modèle, vous pouvez choisir à la place de renvoyer un objet :

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
    },

    #set( $myFoo = $util.dynamodb.toMapValues($ctx.args) )
    #set( $myFoo.version = $util.dynamodb.toNumber(1) )
    #set( $myFoo.timestamp = $util.dynamodb.toString($util.time.nowISO8601()))

    "attributeValues" : $util.toJson($myFoo)
}
```

Dans l'exemple précédent, nous retournons les arguments convertis comme map au lieu d'une chaîne JSON, ajoutons les champs `version` et `timestamp` avant de les exporter vers le champ `attributeValues` du modèle à l'aide de `$util.toJson(...)`.

La version JSON de chacun des assistants est équivalente à l'encapsulage de la version autre que la version JSON dans `$util.toJson(...)`. Par exemple, les instructions suivantes sont exactement identiques :

```
$util.toStringJson("Hello, World!")
$util.toJson($util.toString("Hello, World!"))
```

vers DynamoDB

Liste des utilitaires de ToDynamoDB

`$util.dynamodb.toDynamoDB(Object) : Map`

Outil général de conversion d'objets pour DynamoDB qui convertit les objets d'entrée en une représentation DynamoDB appropriée. La façon dont il représente certains types est clairement arrêtée : par exemple, il utilise les listes (« L ») plutôt que les ensembles (« SS », « NS », « BS »). Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

Exemple de chaîne

```
Input:      $util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

Exemple de numéro

```
Input:      $util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

Exemple booléen

```
Input:      $util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

Exemple de liste

```
Input:      $util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

Exemple de carte

```
Input:      $util.dynamodb.toDynamoDB({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
                           { "S" : "boop" }
                       ]
```

```
                }
            }
        }
```

`$util.dynamodb.toDynamoDBJson(Object) : String`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toDynamoDB(Object) : Map`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

Utilitaires ToString

Liste des utilitaires ToString

`$util.dynamodb.toString(String) : String`

Convertit une chaîne d'entrée au format de chaîne DynamoDB. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      $util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

`$util.dynamodb.toStringJson(String) : Map`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toString(String) : String`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

`$util.dynamodb.toStringSet(List<String>) : Map`

Convertit une liste contenant des chaînes au format de jeu de chaînes DynamoDB. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      $util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

`$util.dynamodb.toStringSetJson(List<String>) : String`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toStringSet(List<String>) : Map`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

Utils ToNumber

Liste des utilitaires ToNumber

`$util.dynamodb.toNumber(Number) : Map`

Convertit un nombre au format numérique DynamoDB. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      $util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

`$util.dynamodb.toNumberJson(Number) : String`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toNumber(Number) : Map`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

`$util.dynamodb.toNumberSet(List<Number>) : Map`

Convertit une liste de nombres au format d'ensemble de numéros DynamoDB. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      $util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

`$util.dynamodb.toNumberSetJson(List<Number>) : String`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toNumberSet(List<Number>) : Map`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

Utilitaires ToBinary

Liste des utilitaires ToBinary

`$util.dynamodb.toBinary(String) : Map`

Convertit les données binaires codées sous forme de chaîne base64 au format binaire DynamoDB. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      $util.dynamodb.toBinary("foo")
```

```
Output:     { "B" : "foo" }
```

`$util.dynamodb.toBinaryJson(String) : String`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toBinary(String) : Map`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

`$util.dynamodb.toBinarySet(List<String>) : Map`

Convertit une liste de données binaires codées sous forme de chaînes base64 au format d'ensemble binaire DynamoDB. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      $util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

`$util.dynamodb.toBinarySetJson(List<String>) : String`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toBinarySet(List<String>) : Map`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

Utilitaires ToBoolean

Liste des utilitaires ToBoolean

`$util.dynamodb.toBoolean(Boolean) : Map`

Convertit un booléen au format booléen DynamoDB approprié. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      $util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

`$util.dynamodb.toBooleanJson(Boolean) : String`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toBoolean(Boolean) : Map`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

Utilitaires ToNull

Liste des utilitaires ToNull

`$util.dynamodb.toNull() : Map`

Renvoie une valeur nulle au format DynamoDB nul. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      $util.dynamodb.toNull()
Output:     { "NULL" : null }
```

`$util.dynamodb.toNullJson() : String`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toNull() : Map`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

Utilitaires ToList

Liste des utilitaires ToList

`$util.dynamodb.toList(List) : Map`

Convertit une liste d'objets au format de liste DynamoDB. Chaque élément de la liste est également converti au format DynamoDB approprié. La façon dont il représente certains objets imbriqués est clairement arrêtée : par exemple, il utilise les listes (« L ») plutôt que les ensembles (« SS », « NS », « BS »). Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      $util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

`$util.dynamodb.toListJson(List) : String`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toList(List) : Map`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

Utilitaires TomaP

Liste des utilitaires de TomaP

`$util.dynamodb.toMap(Map) : Map`

Convertit une carte au format de carte DynamoDB. Chaque valeur de la carte est également convertie au format DynamoDB approprié. La façon dont il représente certains objets imbriqués est clairement arrêtée : par exemple, il utilise les listes (« L ») plutôt que les ensembles (« SS », « NS », « BS »). Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      $util.dynamodb.toMap({ "foo": "bar", "baz" : 1234, "beep": [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
                           { "S" : "boop" }
                       ]
                   }
               }
           }
```

`$util.dynamodb.toMapJson(Map) : String`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toMap(Map) : Map`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

`$util.dynamodb.toMapValues(Map) : Map`

Crée une copie de la carte dans laquelle chaque valeur a été convertie au format DynamoDB approprié. La façon dont il représente certains objets imbriqués est clairement arrêtée : par exemple, il utilise les listes (« L ») plutôt que les ensembles (« SS », « NS », « BS »).

```
Input:      $util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
```

```
Output:     {
               "foo"  : { "S" : "bar" },
               "baz"  : { "N" : 1234 },
               "beep" : {
                   "L" : [
                       { "S" : "boop" }
                   ]
               }
           }
```

> **Note**
>
> Cela est légèrement différent `$util.dynamodb.toMap(Map) : Map` car il renvoie uniquement le contenu de la valeur d'attribut DynamoDB, mais pas la valeur d'attribut complète elle-même. Par exemple, les instructions suivantes sont exactement identiques :
>
> ```
> $util.dynamodb.toMapValues($map)
> $util.dynamodb.toMap($map).get("M")
> ```

`$util.dynamodb.toMapValuesJson(Map) : String`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toMapValues(Map) : Map`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

Utilitaires de l'objet S3

Liste des utilitaires de S3Object

`$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`

Convertit la clé, le compartiment et la région en représentation de l'objet DynamoDB S3. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      $util.dynamodb.toS3Object("foo", "bar", region = "baz")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo", \"bucket\" : \"bar", \"region
\" : \"baz" } }" }
```

`$util.dynamodb.toS3ObjectJson(String key, String bucket, String region) : String`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

`$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map`

Convertit la clé, le compartiment, la région et la version facultative en représentation de l'objet DynamoDB S3. Cela renvoie un objet qui décrit la valeur de l'attribut DynamoDB.

```
Input:      $util.dynamodb.toS3Object("foo", "bar", "baz", "beep")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region
\" : \"baz\", \"version\" = \"beep\" } }" }
```

`$util.dynamodb.toS3ObjectJson(String key, String bucket, String region, String version) : String`

Identique à la valeur de l'attribut DynamoDB`$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map`, mais renvoie la valeur de l'attribut DynamoDB sous forme de chaîne codée JSON.

`$util.dynamodb.fromS3ObjectJson(String) : Map`

Accepte la valeur de chaîne d'un objet DynamoDB S3 et renvoie une carte contenant la clé, le compartiment, la région et la version facultative.

```
Input:      $util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
 \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :
 "beep" }
```

Assistants Amazon RDS dans $util.rds

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

`$util.rds`contient des méthodes d'assistance qui formatent les opérations Amazon RDS en supprimant les données superflues dans les sorties de résultats

Liste des utilitaires de $util.rds

`$util.rds.toJsonString(String serializedSQLResult): String`

Renvoie un `String` en transformant le format de résultat brut des opérations de l'API de données Amazon Relational Database Service (Amazon RDS) sous forme de chaînes en une chaîne plus concise. La chaîne renvoyée est une liste sérialisée d'enregistrements SQL du jeu de résultats. Chaque enregistrement est représenté sous la forme d'un ensemble de paires clé-valeur. Les clés sont les noms de colonnes correspondantes.

Si l'instruction correspondante dans l'entrée était une requête SQL qui provoque une mutation (par exemple INSERT, UPDATE, DELETE), une liste vide est renvoyée. Par exemple, la requête `select * from Books limit 2` fournit le résultat brut de l'opération Amazon RDS Data :

```
{
    "sqlStatementResults": [
        {
            "numberOfRecordsUpdated": 0,
            "records": [
                [
                    {
                        "stringValue": "Mark Twain"
                    },
                    {
                        "stringValue": "Adventures of Huckleberry Finn"
                    },
                    {
                        "stringValue": "978-1948132817"
                    }
                ],
                [
                    {
                        "stringValue": "Jack London"
                    },
                    {
                        "stringValue": "The Call of the Wild"
                    },
                    {
                        "stringValue": "978-1948132275"
                    }
```

```
            ]
        ],
        "columnMetadata": [
            {
                "isSigned": false,
                "isCurrency": false,
                "label": "author",
                "precision": 200,
                "typeName": "VARCHAR",
                "scale": 0,
                "isAutoIncrement": false,
                "isCaseSensitive": false,
                "schemaName": "",
                "tableName": "Books",
                "type": 12,
                "nullable": 0,
                "arrayBaseColumnType": 0,
                "name": "author"
            },
            {
                "isSigned": false,
                "isCurrency": false,
                "label": "title",
                "precision": 200,
                "typeName": "VARCHAR",
                "scale": 0,
                "isAutoIncrement": false,
                "isCaseSensitive": false,
                "schemaName": "",
                "tableName": "Books",
                "type": 12,
                "nullable": 0,
                "arrayBaseColumnType": 0,
                "name": "title"
            },
            {
                "isSigned": false,
                "isCurrency": false,
                "label": "ISBN-13",
                "precision": 15,
                "typeName": "VARCHAR",
                "scale": 0,
                "isAutoIncrement": false,
                "isCaseSensitive": false,
```

```
                    "schemaName": "",
                    "tableName": "Books",
                    "type": 12,
                    "nullable": 0,
                    "arrayBaseColumnType": 0,
                    "name": "ISBN-13"
                }
            ]
        }
    ]
}
```

`util.rds.toJsonString`C'est :

```
[
  {
    "author": "Mark Twain",
    "title": "Adventures of Huckleberry Finn",
    "ISBN-13": "978-1948132817"
  },
  {
    "author": "Jack London",
    "title": "The Call of the Wild",
    "ISBN-13": "978-1948132275"
  },
]
```

`$util.rds.toJsonObject(String serializedSQLResult): Object`

C'est la même chose que`util.rds.toJsonString`, mais le résultat est un JSON`Object`.

assistants HTTP dans $util.http

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

L'`$util.http`utilitaire fournit des méthodes d'assistance que vous pouvez utiliser pour gérer les paramètres des requêtes HTTP et pour ajouter des en-têtes de réponse.

Liste des utilitaires de $util.http

`$util.http.copyHeaders(Map) : Map`

Copie l'en-tête de la carte sans l'ensemble restreint d'en-têtes HTTP. Vous pouvez l'utiliser pour transférer les en-têtes de requête vers votre point de terminaison HTTP en aval.

```
{
    ...
    "params": {
        ...
        "headers": $util.http.copyHeaders($ctx.request.headers),
        ...
    },
    ...
}
```

`$util.http.addResponseHeader(String, Object)`

Ajoute un seul en-tête personnalisé avec le nom (`String`) et la valeur (`Object`) de la réponse. Les limites suivantes s'appliquent :

- Les noms d'en-têtes ne peuvent correspondre à aucun des AWS AppSync en-têtes existants AWS ou restreints.
- Les noms d'en-tête ne peuvent pas commencer par des préfixes restreints, tels que `x-amzn-` ou `x-amz-`.
- La taille des en-têtes de réponse personnalisés ne peut pas dépasser 4 Ko. Cela inclut les noms et les valeurs des en-têtes.
- Vous devez définir chaque en-tête de réponse une fois par opération GraphQL. Toutefois, si vous définissez plusieurs fois un en-tête personnalisé portant le même nom, la définition la plus récente apparaît dans la réponse. Tous les en-têtes sont pris en compte dans la limite de taille d'en-tête, quel que soit leur nom.

```
...
$util.http.addResponseHeader("itemsCount", 7)
$util.http.addResponseHeader("render", $ctx.args.render)
...
```

`$util.http.addResponseHeaders(Map)`

Ajoute plusieurs en-têtes de réponse à la réponse à partir de la carte de noms (`String`) et de valeurs (`Object`) spécifiée. Les mêmes limites répertoriées pour la `addResponseHeader(String, Object)` méthode s'appliquent également à cette méthode.

```
...
#set($headersMap = {})
$util.qr($headersMap.put("headerInt", 12))
$util.qr($headersMap.put("headerString", "stringValue"))
$util.qr($headersMap.put("headerObject", {"field1": 7, "field2": "string"}))
$util.http.addResponseHeaders($headersMap)
...
```

Assistants XML dans $util.xml

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

`$util.xml`contient des méthodes d'assistance qui peuvent faciliter la traduction des réponses XML en JSON ou en dictionnaire.

Liste des utilitaires de $util.xml

`$util.xml.toMap(String) : Map`

Convertit une chaîne XML en dictionnaire.

```
Input:

<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
</posts>
```

```
Output (JSON representation):

{
  "posts":{
    "post":{
      "id":1,
      "title":"Getting started with GraphQL"
    }
  }
}


Input:

<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
<post>
  <id>2</id>
  <title>Getting started with AWS AppSync</title>
</post>
</posts>

Output (JSON representation):

{
  "posts":{
    "post":[
        {
          "id":1,
          "title":"Getting started with GraphQL"
        },
        {
          "id":2,
          "title":"Getting started with AWS AppSync"
        }
    ]
  }
}
```

`$util.xml.toJsonString(String) : String`

Convertit une chaîne XML en chaîne JSON. Ceci est similaire à TomAp, sauf que la sortie est une chaîne. Cela est utile si vous souhaitez convertir et renvoyer directement la réponse XML à partir d'un objet HTTP vers JSON.

`$util.xml.toJsonString(String, Boolean) : String`

Convertit une chaîne XML en chaîne JSON avec un paramètre booléen facultatif pour déterminer si vous souhaitez coder le JSON par chaîne.

Aides à la transformation dans $util.transform

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

`$util.transform`contient des méthodes d'assistance qui facilitent l'exécution d'opérations complexes sur des sources de données, telles que les opérations de filtrage Amazon DynamoDB.

Aides à la transformation

Liste des outils d'aide à la transformation

`$util.transform.toDynamoDBFilterExpression(Map) : Map`

Convertit une chaîne d'entrée en une expression de filtre à utiliser avec DynamoDB.

```
Input:

$util.transform.toDynamoDBFilterExpression({
    "title":{
      "contains":"Hello World"
    }
  })

Output:

{
```

```
    "expression" : "contains(#title, :title_contains)"
    "expressionNames" : {
        "#title" : "title",
    },
    "expressionValues" : {
        ":title_contains" : { "S" : "Hello World" }
    },
}
```

`$util.transform.toElasticsearchQueryDSL(Map) : Map`

Convertit l'entrée donnée en son expression OpenSearch Query DSL équivalente, en la renvoyant sous forme de chaîne JSON.

```
Input:

$util.transform.toElasticsearchQueryDSL({
    "upvotes":{
        "ne":15,
        "range":[
            10,
            20
        ]
    },
    "title":{
        "eq":"hihihi",
        "wildcard":"h*i"
    }
  })

Output:
{
    "bool":{
      "must":[
          {
            "bool":{
              "must":[
                  {
                    "bool":{
                      "must_not":{
                        "term":{
                          "upvotes":15
                        }
```

```
                    }
                  }
                },
                {
                  "range":{
                    "upvotes":{
                      "gte":10,
                      "lte":20
                    }
                  }
                }
              ]
            }
          },
          {
            "bool":{
              "must":[
                {
                  "term":{
                    "title":"hihihi"
                  }
                },
                {
                "wildcard":{
                    "title":"h*i"
                  }
                }
              ]
            }
          }
      ]
    }
}
```

L'opérateur par défaut est supposé être AND.

Filtres d'abonnement pour les aides à la transformation

Transformation Helpers (abonnement, filtres, liste d'utilitaires)

`$util.transform.toSubscriptionFilter(Map) : Map`

Convertit un objet `Map` d'entrée en objet d'`SubscriptionFilter`expression. La `$util.transform.toSubscriptionFilter` méthode est utilisée comme entrée dans l'`$extensions.setSubscriptionFilter()`extension. Pour plus d'informations, consultez la section Extensions.

`$util.transform.toSubscriptionFilter(Map, List) : Map`

Convertit un objet `Map` d'entrée en objet d'`SubscriptionFilter`expression. La `$util.transform.toSubscriptionFilter` méthode est utilisée comme entrée dans l'`$extensions.setSubscriptionFilter()`extension. Pour plus d'informations, consultez la section Extensions.

Le premier argument est l'objet `Map` d'entrée converti en objet d'`SubscriptionFilter`expression. Le deuxième argument est un nom `List` de champ qui est ignoré dans le premier objet `Map` d'entrée lors de la construction de l'objet `SubscriptionFilter` d'expression.

`$util.transform.toSubscriptionFilter(Map, List, Map) : Map`

Convertit un objet `Map` d'entrée en objet d'`SubscriptionFilter`expression. La `$util.transform.toSubscriptionFilter` méthode est utilisée comme entrée dans l'`$extensions.setSubscriptionFilter()`extension. Pour plus d'informations, consultez la section Extensions.

Le premier argument est l'objet `Map` d'entrée converti en objet d'`SubscriptionFilter`expression, le deuxième argument est un nom `List` de champ qui sera ignoré dans le premier objet `Map` d'entrée, et le troisième argument est un objet `Map` d'entrée soumis à des règles strictes qui est inclus lors de la construction de l'objet `SubscriptionFilter` d'expression. Ces règles strictes sont incluses dans l'objet `SubscriptionFilter` d'expression de telle sorte qu'au moins l'une des règles soit satisfaite pour passer le filtre d'abonnement.

Arguments du filtre d'abonnement

Le tableau suivant explique comment les arguments des utilitaires suivants sont définis :

- `$util.transform.toSubscriptionFilter(Map) : Map`
- `$util.transform.toSubscriptionFilter(Map, List) : Map`
- `$util.transform.toSubscriptionFilter(Map, List, Map) : Map`

Argument 1: Map

L'argument 1 est un `Map` objet dont les valeurs clés sont les suivantes :

- noms de champs
- « et »
- « ou »

Pour les noms de champs sous forme de clés, les conditions relatives aux entrées de ces champs sont sous la forme de`"operator" : "value"`.

L'exemple suivant montre comment des entrées peuvent être ajoutées au `Map` :

```
"field_name" : {
                    "operator1" : value
               }

## We can have multiple conditions for the same field_name:

"field_name" : {
                    "operator1" : value
                    "operator2" : value
                    .
                    .
                    .
               }
```

Lorsqu'un champ comporte au moins deux conditions, toutes ces conditions sont considérées comme utilisant l'opération OR.

L'entrée `Map` peut également comporter des touches « et » et « ou », ce qui implique que toutes les entrées qu'elles contiennent doivent être jointes en utilisant la logique AND ou OR en fonction de la clé. Les valeurs clés « et » et « ou » supposent un ensemble de conditions.

```
"and" : [
```

```
            {
                "field_name1" : {
                    "operator1" : value
                }
             },

             {
                "field_name2" : {
                    "operator1" : value
                }
             },
             .
             .
        ].
```

Notez que vous pouvez imbriquer « et » et « ou ». C'est-à-dire que vous pouvez avoir imbriqué « et » /"ou » dans un autre bloc « et » /"ou ». Toutefois, cela ne fonctionne pas pour les champs simples.

```
"and" : [

            {
                "field_name1" : {
                    "operator" : value
                }
             },

             {
                "or" : [
                            {
                                "field_name2" : {
                                    "operator" : value
                                }
                            },

                            {
                                "field_name3" : {
                                    "operator" : value
                                }
                            }
```

```
                        ].
```

L'exemple suivant montre une entrée de l'argument 1 utilisant`$util.transform.toSubscriptionFilter(Map) : Map`.

Entrée (s)

Argument 1 : Carte :

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Sortie

Le résultat est un `Map` objet :

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Admin"
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
```

```
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "gte",
          "value": 20
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Admin"
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "gte",
          "value": 20
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
```

```
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 2: List

L'argument 2 contient des noms `List` de champs qui ne doivent pas être pris en compte dans l'entrée `Map` (argument 1) lors de la construction de l'objet `SubscriptionFilter` d'expression. Ils `List` peuvent également être vides.

L'exemple suivant montre les entrées de l'argument 1 et de l'argument 2 en utilisant`$util.transform.toSubscriptionFilter(Map, List) : Map`.

Entrée (s)

Argument 1 : Carte :

```
{

  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
```

```
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argument 2 : Liste :

```
["percentageUp", "author"]
```

Sortie

Le résultat est un `Map` objet :

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
```

```
      ]
    }
  ]
}
```

Argument 3: Map

L'argument 3 est un `Map` objet dont les noms de champs sont des valeurs clés (il ne peut pas y avoir « et » ou « ou »). Pour les noms de champs sous forme de clés, les conditions relatives à ces champs sont des entrées sous la forme de`"operator" : "value"`. Contrairement à l'argument 1, l'argument 3 ne peut pas avoir plusieurs conditions dans la même clé. De plus, l'argument 3 ne contient pas de clause « et » ou « ou », il n'y a donc pas non plus d'imbrication.

L'argument 3 représente une liste de règles strictes, qui sont ajoutées à l'objet `SubscriptionFilter` d'expression afin qu'au moins l'une de ces conditions soit remplie pour passer le filtre.

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.
```

L'exemple suivant montre les entrées de l'argument 1, de l'argument 2 et de l'argument 3 en utilisant`$util.transform.toSubscriptionFilter(Map, List, Map) : Map`.

Entrée (s)

Argument 1 : Carte :

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
```

```
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argument 2 : Liste :

```
["percentageUp", "author"]
```

Argument 3 : Carte :

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
}
```

Sortie

Le résultat est un `Map` objet :

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        },
        {
          "fieldName": "upvotes",
          "operator": "gte",
          "value": 250
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
```

```
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Person1"
        }
      ]
    }
  ]
}
```

Assistants mathématiques dans $util.math

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

`$util.math`contient des méthodes pour faciliter les opérations mathématiques courantes.

Liste des utilitaires de $util.math

`$util.math.roundNum(Double) : Integer`

Prend un double et l'arrondit à l'entier le plus proche.

`$util.math.minVal(Double, Double) : Double`

Prend deux doubles et renvoie la valeur minimale entre les deux doubles.

`$util.math.maxVal(Double, Double) : Double`

Prend deux doubles et renvoie la valeur maximale entre les deux doubles.

`$util.math.randomDouble() : Double`

Renvoie un double aléatoire compris entre 0 et 1.

> **Important**
>
> Cette fonction ne doit pas être utilisée pour tout ce qui nécessite un caractère aléatoire à entropie élevé (par exemple, la cryptographie).

`$util.math.randomWithinRange(Integer, Integer) : Integer`

Renvoie une valeur entière aléatoire comprise dans la plage spécifiée, le premier argument indiquant la valeur inférieure de la plage et le second la valeur supérieure de la plage.

> **Important**
>
> Cette fonction ne doit pas être utilisée pour tout ce qui nécessite un caractère aléatoire à entropie élevé (par exemple, la cryptographie).

assistants de chaîne dans $util.str

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

`$util.str`contient des méthodes pour faciliter les opérations courantes sur les chaînes de caractères.

Liste des utilitaires de $util.str

`$util.str.toUpper(String) : String`

Prend une chaîne et la convertit pour qu'elle soit entièrement en majuscules.

`$util.str.toLower(String) : String`

Prend une chaîne de caractères et la convertit en minuscules.

`$util.str.toReplace(String, String, String) : String`

Remplace une sous-chaîne dans une chaîne par une autre chaîne. Le premier argument indique la chaîne sur laquelle effectuer l'opération de remplacement. Le deuxième argument indique

la sous-chaîne à remplacer. Le troisième argument indique la chaîne par laquelle remplacer le deuxième argument. Voici un exemple d'utilisation de cet utilitaire :

```
INPUT:      $util.str.toReplace("hello world", "hello", "mellow")
OUTPUT:     "mellow world"
```

`$util.str.normalize(String, String) : String`

Normalise une chaîne à l'aide de l'une des quatre formes de normalisation Unicode : NFC, NFD, NFKC ou NFKD. Le premier argument est la chaîne à normaliser. Le deuxième argument est « nfc », « nfd », « nfkc » ou « nfkd », spécifiant le type de normalisation à utiliser pour le processus de normalisation.

Extensions

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

`$extensions`contient un ensemble de méthodes permettant d'effectuer des actions supplémentaires dans vos résolveurs.

$extensions. evictFromApiCache (chaîne, chaîne, objet) : objet

Permet d'expulser un élément du cache AWS AppSync côté serveur. Le premier argument est le nom du type. Le deuxième argument est le nom du champ. Le troisième argument est un objet contenant des éléments de paire clé-valeur qui spécifient la valeur de la clé de mise en cache. Vous devez placer les éléments dans l'objet dans le même ordre que les clés de mise en cache dans le résolveur mis en cache. `cachingKey`

> **Note**
>
> Cet utilitaire ne fonctionne que pour les mutations, pas pour les requêtes.

$extensions. setSubscriptionFilter(filterJsonObject)

Définit des filtres d'abonnement améliorés. Chaque événement de notification d'abonnement est évalué par rapport aux filtres d'abonnement fournis et envoie des notifications aux clients si tous les filtres sont conformes`true`. L'argument est `filterJsonObject` tel que décrit ci-dessous.

> **Note**
>
> Vous pouvez utiliser cette méthode d'extension uniquement dans les modèles de mappage des réponses d'un résolveur d'abonnement.

$extensions. setSubscriptionInvalidationFiltre (filterJsonObject)

Définit les filtres d'invalidation des abonnements. Les filtres d'abonnement sont évalués par rapport à la charge utile d'invalidation, puis invalident un abonnement donné s'ils sont évalués à. `true` L'argument est `filterJsonObject` tel que décrit ci-dessous.

> **Note**
>
> Vous pouvez utiliser cette méthode d'extension uniquement dans les modèles de mappage des réponses d'un résolveur d'abonnement.

Argument : filterJsonObject

L'objet JSON définit des filtres d'abonnement ou d'invalidation. Il s'agit d'un ensemble de filtres dans un`filterGroup`. Chaque filtre est un ensemble de filtres individuels.

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]
```

```
        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

Chaque filtre possède trois attributs :

- `fieldName`— Le champ du schéma GraphQL.
- `operator`— Le type d'opérateur.
- `value`— Les valeurs à comparer à la `fieldName` valeur de la notification d'abonnement.

Voici un exemple d'attribution de ces attributs :

```
{
 "fieldName" : "severity",
 "operator" : "le",
 "value" : $context.result.severity
}
```

Domaine : FieldName

Le type de chaîne `fieldName` fait référence à un champ défini dans le schéma GraphQL qui correspond `fieldName` à la charge utile de notification d'abonnement. Lorsqu'une correspondance est trouvée, le champ `value` du schéma GraphQL est comparé à celui du filtre `value` de notification d'abonnement. Dans l'exemple suivant, le `fieldName` filtre correspond au `service` champ défini dans un type GraphQL donné. Si la charge utile de notification contient un `service` champ avec un `value` équivalent à`AWS AppSync`, le filtre est évalué comme suit : `true`

```
{
 "fieldName" : "service",
 "operator" : "eq",
```

```
 "value" : "AWS AppSync"
}
```

Champ : valeur

La valeur peut être d'un type différent en fonction de l'opérateur :

- Un nombre unique ou un booléen
  - Exemples de chaînes :`"test"`, `"service"`
  - Exemples de chiffres :1,2, `45.75`
  - Exemples booléens :, `true` `false`
- Paires de nombres ou de chaînes
  - Exemple de paire de chaînes :`["test1","test2"]`, `["start","end"]`
  - Exemple de paire de numéros :`[1,4]`,`[67,89]`, `[12.45, 95.45]`
- Tableaux de nombres ou de chaînes
  - Exemple de tableau de chaînes : `["test1","test2","test3","test4","test5"]`
  - Exemple de tableau numérique :`[1,2,3,4,5]`, `[12.11,46.13,45.09,12.54,13.89]`

Domaine : opérateur

Chaîne distinguant majuscules et minuscules avec les valeurs possibles suivantes :

Opérateur	Description	Types de valeurs possibles
eq	Égal à	entier, flottant, chaîne de caractères, booléen
ne	Non égal à	entier, flottant, chaîne de caractères, booléen
le	Inférieur ou égal à	entier, flottant, chaîne
lt	Inférieur à	entier, flottant, chaîne
gm	Supérieur ou égal à	entier, flottant, chaîne
gt	Supérieure à	entier, flottant, chaîne

contient	Vérifie la présence d'une sous-séquence ou d'une valeur dans l'ensemble.	entier, flottant, chaîne
NE CONTIENT PAS	Vérifie l'absence de sous-séquence ou l'absence de valeur dans l'ensemble.	entier, flottant, chaîne
Commence par	Vérifie la présence d'un préfixe.	chaîne
dans	Vérifie les éléments correspondants figurant dans la liste.	Tableau de nombres entiers, flottants ou chaînes
notIn	Vérifie les éléments correspondants qui ne figurent pas dans la liste.	Tableau de nombres entiers, flottants ou chaînes
between	Entre deux valeurs	entier, flottant, chaîne
Contient n'importe quel	Contient des éléments communs	entier, flottant, chaîne

Le tableau suivant décrit comment chaque opérateur est utilisé dans la notification d'abonnement.

eq (equal)

L'`eq`opérateur évalue `true` si la valeur du champ de notification d'abonnement correspond et est strictement égale à la valeur du filtre. Dans l'exemple suivant, le filtre détermine `true` si la notification d'abonnement contient un `service` champ dont la valeur est équivalente à`AWS AppSync`.

Types de valeurs possibles : entier, flottant, chaîne, booléen

```
{
 "fieldName" : "service",
 "operator" : "eq",
 "value" : "AWS AppSync"
}
```

ne (not equal)

L'`ne`opérateur détermine `true` si la valeur du champ de notification d'abonnement est différente de la valeur du filtre. Dans l'exemple suivant, le filtre détermine `true` si la notification d'abonnement contient un `service` champ dont la valeur est différente de`AWS AppSync`.

Types de valeurs possibles : entier, flottant, chaîne, booléen

```
{
 "fieldName" : "service",
 "operator" : "ne",
 "value" : "AWS AppSync"
}
```

le (less or equal)

L'`le`opérateur détermine `true` si la valeur du champ de notification d'abonnement est inférieure ou égale à la valeur du filtre. Dans l'exemple suivant, le filtre détermine `true` si la notification d'abonnement contient un `size` champ dont la valeur est inférieure ou égale à`5`.

Types de valeurs possibles : entier, flottant, chaîne

```
{
 "fieldName" : "size",
 "operator" : "le",
 "value" : 5
}
```

lt (less than)

L'`lt`opérateur détermine `true` si la valeur du champ de notification d'abonnement est inférieure à la valeur du filtre. Dans l'exemple suivant, le filtre détermine `true` si la notification d'abonnement contient un `size` champ dont la valeur est inférieure `5` à.

Types de valeurs possibles : entier, flottant, chaîne

```
{
 "fieldName" : "size",
 "operator" : "lt",
 "value" : 5
}
```

ge (greater or equal)

L'`ge`opérateur détermine `true` si la valeur du champ de notification d'abonnement est supérieure ou égale à la valeur du filtre. Dans l'exemple suivant, le filtre détermine `true` si la notification d'abonnement contient un `size` champ dont la valeur est supérieure ou égale à`5`.

Types de valeurs possibles : entier, flottant, chaîne

```
{
 "fieldName" : "size",
 "operator" : "ge",
 "value" : 5
}
```

gt (greater than)

L'`gt`opérateur détermine `true` si la valeur du champ de notification d'abonnement est supérieure à la valeur du filtre. Dans l'exemple suivant, le filtre détermine `true` si la notification d'abonnement contient un `size` champ dont la valeur est supérieure à`5`.

Types de valeurs possibles : entier, flottant, chaîne

```
{
 "fieldName" : "size",
 "operator" : "gt",
 "value" : 5
}
```

contains

L'`contains`opérateur vérifie la présence d'une sous-chaîne, d'une sous-séquence ou d'une valeur dans un ensemble ou un élément unique. Un filtre avec l'`contains`opérateur détermine `true` si la valeur du champ de notification d'abonnement contient la valeur du filtre. Dans l'exemple suivant, le filtre détermine `true` si la notification d'abonnement contient un `seats` champ contenant la valeur `10` du tableau.

Types de valeurs possibles : entier, flottant, chaîne

```
{
 "fieldName" : "seats",
 "operator" : "contains",
```

```
 "value" : 10
}
```

Dans un autre exemple, le filtre détermine `true` si la notification d'abonnement contient un `event` champ `launch` sous forme de sous-chaîne.

```
{
 "fieldName" : "event",
 "operator" : "contains",
 "value" : "launch"
}
```

notContains

L'`notContains`opérateur vérifie l'absence de sous-chaîne, de sous-séquence ou de valeur dans un ensemble ou un élément unique. Le filtre avec l'`notContains`opérateur détermine `true` si la valeur du champ de notification d'abonnement ne contient pas la valeur du filtre. Dans l'exemple suivant, le filtre détermine `true` si la notification d'abonnement contient un `seats` champ dans lequel la valeur du tableau ne contient pas la valeur`10`.

Types de valeurs possibles : entier, flottant, chaîne

```
{
 "fieldName" : "seats",
 "operator" : "notContains",
 "value" : 10
}
```

Dans un autre exemple, le filtre détermine `true` si la notification d'abonnement contient une valeur de `event` champ sans `launch` sa sous-séquence.

```
{
 "fieldName" : "event",
 "operator" : "notContains",
 "value" : "launch"
}
```

beginsWith

L'`beginsWith`opérateur vérifie la présence d'un préfixe dans une chaîne. Le filtre contenant l'`beginsWith`opérateur détermine `true` si la valeur du champ de notification d'abonnement

commence par la valeur du filtre. Dans l'exemple suivant, le filtre détermine `true` si la notification d'abonnement contient un `service` champ dont la valeur commence `AWS` par.

Type de valeur possible : chaîne

```
{
 "fieldName" : "service",
 "operator" : "beginsWith",
 "value" : "AWS"
}
```

in

L'`in`opérateur vérifie les éléments correspondants dans un tableau. Le filtre contenant l'`in`opérateur détermine `true` si la valeur du champ de notification d'abonnement existe dans un tableau. Dans l'exemple suivant, le filtre détermine `true` si la notification d'abonnement contient un `severity` champ contenant l'une des valeurs présentes dans le tableau :`[1,2,3]`.

Type de valeur possible : tableau de nombres entiers, flottants ou chaînes

```
{
 "fieldName" : "severity",
 "operator" : "in",
 "value" : [1,2,3]
}
```

notIn

L'`notIn`opérateur vérifie s'il n'y a pas d'éléments manquants dans un tableau. Le filtre contenant l'`notIn`opérateur détermine `true` si la valeur du champ de notification d'abonnement n'existe pas dans le tableau. Dans l'exemple suivant, le filtre détermine `true` si la notification d'abonnement contient un `severity` champ contenant l'une des valeurs absentes du tableau :`[1,2,3]`.

Type de valeur possible : tableau de nombres entiers, flottants ou chaînes

```
{
 "fieldName" : "severity",
 "operator" : "notIn",
 "value" : [1,2,3]
}
```

between

L'`between`opérateur vérifie les valeurs comprises entre deux nombres ou chaînes. Le filtre contenant l'`between`opérateur détermine `true` si la valeur du champ de notification d'abonnement se situe entre la paire de valeurs du filtre. Dans l'exemple suivant, le filtre détermine `true` si la notification d'abonnement contient un `severity` champ contenant des valeurs`2`,`3`,`4`.

Types de valeurs possibles : paire d'entiers, de nombres flottants ou de chaînes

```
{
 "fieldName" : "severity",
 "operator" : "between",
 "value" : [1,5]
}
```

containsAny

L'`containsAny`opérateur vérifie la présence d'éléments communs dans les tableaux. Un filtre avec l'`containsAny`opérateur évalue `true` si l'intersection entre la valeur définie du champ de notification d'abonnement et la valeur définie du filtre n'est pas vide. Dans l'exemple suivant, le filtre détermine `true` si la notification d'abonnement contient un `seats` champ dont la valeur du tableau contient soit `10` ou`15`. Cela signifie que le filtre évaluera `true` si la valeur du `seats` champ de la notification d'abonnement est égale à `[10,11]` ou`[15,20,30]`.

Types de valeurs possibles : entier, flottant ou chaîne

```
{
 "fieldName" : "seats",
 "operator" : "containsAny",
 "value" : [10, 15]
}
```

Logique AND

Vous pouvez combiner plusieurs filtres à l'aide de la logique AND en définissant plusieurs entrées dans l'`filters`objet du `filterGroup` tableau. Dans l'exemple suivant, les filtres évaluent `true` si la notification d'abonnement contient un `userId` champ avec une valeur équivalente à `1` ET une valeur de `group` champ égale à `Admin` ou`Developer`.

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

Logique OR

Vous pouvez combiner plusieurs filtres à l'aide de la logique OR en définissant plusieurs objets de filtre dans le `filterGroup` tableau. Dans l'exemple suivant, les filtres évaluent `true` si la notification d'abonnement contient un `userId` champ avec une valeur équivalente à 1 OU une valeur de `group` champ égale à `Admin` ou`Developer`.

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
                {
```

```
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
            ]

        }
    ]
}
```

Exceptions

Notez que l'utilisation des filtres est soumise à plusieurs restrictions :

- Dans l'`filters`objet, il peut y avoir un maximum de cinq `fieldName` éléments uniques par filtre. Cela signifie que vous pouvez combiner un maximum de cinq `fieldName` objets individuels à l'aide de la logique AND.
- Il peut y avoir un maximum de vingt valeurs pour l'`containsAny`opérateur.
- Il peut y avoir un maximum de cinq valeurs pour les `notIn` opérateurs `in` et.
- Chaque chaîne peut comporter un maximum de 256 caractères.
- Chaque comparaison de chaînes fait la distinction majuscules/minuscules.
- Le filtrage des objets imbriqués permet d'obtenir jusqu'à cinq niveaux de filtrage imbriqués.
- Chacun `filterGroup` peut en avoir un maximum de 10`filters`. Cela signifie que vous pouvez en combiner un maximum de 10 `filters` en utilisant la logique OR.
  - L'`in`opérateur est un cas particulier de la logique OR. Dans l'exemple suivant, il y en a deux `filters` :

```
{
    "filterGroup": [
        {
           "filters" : [
                {
                   "fieldName" : "userId",
                   "operator" : "eq",
                   "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "in",
```

```
                    "value" : ["Admin", "Developer"]
                }
            ]
        }
    ]
}
```

Le groupe de filtres précédent est évalué comme suit et est pris en compte dans le calcul de la limite maximale de filtres :

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "eq",
                    "value" : "Admin"
                }
           ]
        },
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "eq",
                    "value" : "Developer"
                }
           ]
        }
    ]
}
```

$extensions.invalidateSubscriptions () invalidationJsonObject

Utilisé pour initier l'invalidation d'un abonnement suite à une mutation. L'argument est `invalidationJsonObject` tel que décrit ci-dessous.

> **Note**
>
> Cette extension ne peut être utilisée que dans les modèles de mappage des réponses des résolveurs de mutations.
> Vous ne pouvez utiliser qu'au maximum cinq appels de `$extensions.invalidateSubscriptions()` méthode uniques dans une seule demande. Si vous dépassez cette limite, vous recevrez une erreur GraphQL.

Argument : invalidationJsonObject

`invalidationJsonObject`définit les éléments suivants :

- `subscriptionField`— L'abonnement au schéma GraphQL à invalider. Un seul abonnement, défini sous la forme d'une chaîne dans le`subscriptionField`, est considéré comme invalide.
- `payload`— Une liste de paires clé-valeur utilisée comme entrée pour invalider les abonnements si le filtre d'invalidation est évalué par rapport à leurs valeurs. `true`

  L'exemple suivant invalide les clients abonnés et connectés utilisant l'on`UserDelete`abonnement lorsque le filtre d'invalidation défini dans le résolveur d'abonnement est évalué par rapport à la valeur. `true` `payload`

```
$extensions.invalidateSubscriptions({
        "subscriptionField": "onUserDelete",
        "payload": {
                "group": "Developer"
                "type" : "Full-Time"
      }
    })
```

Référence du modèle de mappage Resolver pour DynamoDB

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Le résolveur AWS AppSync DynamoDB vous permet d'utiliser GraphQL pour stocker et récupérer des données dans les tables Amazon DynamoDB existantes de votre compte. Ce résolveur fonctionne en vous permettant de mapper une requête GraphQL entrante en un appel DynamoDB, puis de mapper la réponse DynamoDB à GraphQL. Cette section décrit les modèles de mappage pour les opérations DynamoDB prises en charge.

GetItem

Le document de mappage des demandes vous permet de `GetItem` demander au résolveur AWS AppSync DynamoDB d'envoyer une `GetItem` demande à DynamoDB et de spécifier :

- La clé de l'élément dans DynamoDB
- S'il convient d'utiliser une lecture cohérente ou non

Le document de mappage `GetItem` a la structure suivante :

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true,
    "projection" : {
        ...
    }
}
```

Les champs sont définis comme suit :

Champs de GetItem

GetItem liste des champs

`version`

Version de la définition du modèle. `2017-02-28` et `2018-05-29` sont actuellement prises en charge. Cette valeur est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `GetItem`, ce champ doit être défini sur `GetItem`. Cette valeur est obligatoire.

`key`

La clé de l'élément dans DynamoDB. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la façon de spécifier une « valeur saisie », voir Système de types (mappage des demandes). Cette valeur est obligatoire.

`consistentRead`

S'il faut ou non effectuer une lecture très cohérente avec DynamoDB. Ce champ est facultatif et `false` est la valeur définie par défaut.

`projection`

Projection utilisée pour spécifier les attributs à renvoyer par l'opération DynamoDB. Pour plus d'informations sur les projections, voir Projections. Ce champ est facultatif.

L'élément renvoyé par DynamoDB est automatiquement converti en types primitifs GraphQL et JSON, et est disponible dans le contexte de mappage (). `$context.result`

Pour plus d'informations sur la conversion de type DynamoDB, voir Système de types (mappage des réponses).

Pour plus d'informations sur les modèles de mappage des réponses, consultez la section Vue d'ensemble des modèles de mappage Resolver.

Exemple

L'exemple suivant est un modèle de mappage pour une requête GraphQL : `getThing(foo: String!, bar: String!)`

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
        "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
    },
    "consistentRead" : true
}
```

Pour de plus amples informations sur l'API de DynamoDB `GetItem`, veuillez consulter la Documentation sur les API de DynamoDB.

PutItem

Le document de mappage des demandes vous permet de `PutItem` demander au résolveur AWS AppSync DynamoDB d'envoyer une `PutItem` demande à DynamoDB et de spécifier les éléments suivants :

- La clé de l'élément dans DynamoDB
- Contenu complet de l'élément (composé de `key` et de `attributeValues`)
- Conditions de réussite de l'opération

Le document de mappage `PutItem` a la structure suivante :

```
{
    "version" : "2018-05-29",
    "operation" : "PutItem",
    "customPartitionKey" : "foo",
    "populateIndexFields" : boolean value,
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "attributeValues" : {
        "baz" : ... typed value
    },
    "condition" : {
       ...
    },
```

```
    "_version" : 1
}
```

Les champs sont définis comme suit :

PutItem champs

PutItem liste des champs

`version`

Version de la définition du modèle. `2017-02-28` et `2018-05-29` sont actuellement prises en charge. Cette valeur est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `PutItem`, ce champ doit être défini sur `PutItem`. Cette valeur est obligatoire.

`key`

La clé de l'élément dans DynamoDB. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la façon de spécifier une « valeur saisie », voir Système de types (mappage des demandes). Cette valeur est obligatoire.

`attributeValues`

Le reste des attributs de l'élément doit être placé dans DynamoDB. Pour plus d'informations sur la façon de spécifier une « valeur saisie », voir Système de types (mappage des demandes). Ce champ est facultatif.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Si aucune condition n'est spécifiée, la requête `PutItem` remplace toute entrée existante pour cet élément. Pour plus d'informations sur les conditions, consultez la section Expressions de condition. Cette valeur est facultative.

`_version`

Valeur numérique représentant la dernière version connue d'un élément. Cette valeur est facultative. Ce champ est utilisé pour la détection de conflits et n'est pris en charge que sur les sources de données versionnées.

`customPartitionKey`

Lorsqu'elle est activée, cette valeur de chaîne modifie le format des `ds_pk` enregistrements `ds_sk` et utilisés par la table de synchronisation delta lorsque le contrôle des versions a été activé (pour plus d'informations, voir Détection des conflits et synchronisation dans le guide du AWS AppSync développeur). Lorsque cette option est activée, le traitement de l'`populateIndexFields`entrée est également activé. Ce champ est facultatif.

`populateIndexFields`

Valeur booléenne qui, lorsqu'elle est activée en même temps que le **`customPartitionKey`**, crée de nouvelles entrées pour chaque enregistrement de la table de synchronisation delta, en particulier dans les colonnes `gsi_ds_pk` et`gsi_ds_sk`. Pour plus d'informations, consultez la section Détection et synchronisation des conflits dans le manuel du AWS AppSync développeur. Ce champ est facultatif.

L'élément écrit dans DynamoDB est automatiquement converti en types primitifs GraphQL et JSON et est disponible dans le contexte de mappage (). `$context.result`

Pour plus d'informations sur la conversion de type DynamoDB, voir Système de types (mappage des réponses).

Pour plus d'informations sur les modèles de mappage des réponses, consultez la section Vue d'ensemble des modèles de mappage Resolver.

Exemple 1

L'exemple suivant est un modèle de mappage pour une mutation GraphQL. `updateThing(foo: String!, bar: String!, name: String!, version: Int!)`

S'il n'existe aucun élément avec la clé spécifiée, il est créé. S'il existe déjà un élément avec la clé spécifiée, il est remplacé.

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
        "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
    },
    "attributeValues" : {
```

```
        "name"    : $util.dynamodb.toDynamoDBJson($ctx.args.name),
        "version" : $util.dynamodb.toDynamoDBJson($ctx.args.version)
    }
}
```

Exemple 2

L'exemple suivant est un modèle de mappage pour une mutation GraphQL. `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)`

Cet exemple vérifie que le champ de l'élément actuellement dans DynamoDB est défini `version` sur. `expectedVersion`

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
        "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
    },
    "attributeValues" : {
        "name"    : $util.dynamodb.toDynamoDBJson($ctx.args.name),
        #set( $newVersion = $context.arguments.expectedVersion + 1 )
        "version" : $util.dynamodb.toDynamoDBJson($newVersion)
    },
    "condition" : {
        "expression" : "version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" : $util.dynamodb.toDynamoDBJson($expectedVersion)
        }
    }
}
```

Pour de plus amples informations sur l'API de DynamoDB `PutItem`, veuillez consulter la Documentation sur les API de DynamoDB.

UpdateItem

Le document de mappage des demandes vous permet de `UpdateItem` demander au résolveur AWS AppSync DynamoDB d'envoyer une `UpdateItem` demande à DynamoDB et de spécifier les éléments suivants :

- La clé de l'élément dans DynamoDB
- Expression de mise à jour décrivant comment mettre à jour l'élément dans DynamoDB
- Conditions de réussite de l'opération

Le document de mappage `UpdateItem` a la structure suivante :

```
{
    "version" : "2018-05-29",
    "operation" : "UpdateItem",
    "customPartitionKey" : "foo",
    "populateIndexFields" : boolean value,
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "update" : {
        "expression" : "someExpression",
        "expressionNames" : {
           "#foo" : "foo"
       },
       "expressionValues" : {
           ":bar" : ... typed value
       }
    },
    "condition" : {
        ...
    },
    "_version" : 1
}
```

Les champs sont définis comme suit :

UpdateItem champs

UpdateItem liste des champs

`version`

Version de la définition du modèle. `2017-02-28` et `2018-05-29` sont actuellement prises en charge. Cette valeur est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `UpdateItem`, ce champ doit être défini sur `UpdateItem`. Cette valeur est obligatoire.

`key`

La clé de l'élément dans DynamoDB. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la spécification d'une « valeur saisie », consultez Système de types (mappage des demandes). Cette valeur est obligatoire.

`update`

La `update` section vous permet de spécifier une expression de mise à jour qui décrit comment mettre à jour l'élément dans DynamoDB. Pour plus d'informations sur la façon d'écrire des expressions de mise à jour, consultez la documentation UpdateExpressions DynamoDB. Cette section est obligatoire.

La section `update` possède trois composants :

`expression`

Expression de mise à jour. Cette valeur est obligatoire.

`expressionNames`

Substituts des espaces réservés de nom des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé utilisé dans le`expression`, et la valeur doit être une chaîne correspondant au nom d'attribut de l'élément dans DynamoDB. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de nom des attributs de l'expression utilisés dans l'`expression`.

`expressionValues`

Substituts des espaces réservés de valeur des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé de valeur utilisé dans l'`expression`, et la valeur doit être typée. Pour plus d'informations sur la façon de spécifier une « valeur saisie », voir Système de types (mappage des demandes). Cela doit être spécifié. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de valeur des attributs de l'expression utilisés dans l'`expression`.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Si aucune condition n'est spécifiée, la requête `UpdateItem` met à jour l'entrée existante quel que soit son état actuel. Pour plus d'informations sur les conditions, consultez la section Expressions de condition. Cette valeur est facultative.

`_version`

Valeur numérique représentant la dernière version connue d'un élément. Cette valeur est facultative. Ce champ est utilisé pour la détection de conflits et n'est pris en charge que sur les sources de données versionnées.

`customPartitionKey`

Lorsqu'elle est activée, cette valeur de chaîne modifie le format des `ds_pk` enregistrements `ds_sk` et utilisés par la table de synchronisation delta lorsque le contrôle des versions a été activé (pour plus d'informations, voir Détection des conflits et synchronisation dans le guide du AWS AppSync développeur). Lorsque cette option est activée, le traitement de l'`populateIndexFields`entrée est également activé. Ce champ est facultatif.

`populateIndexFields`

Valeur booléenne qui, lorsqu'elle est activée en même temps que le **`customPartitionKey`**, crée de nouvelles entrées pour chaque enregistrement de la table de synchronisation delta, en particulier dans les colonnes `gsi_ds_pk` et`gsi_ds_sk`. Pour plus d'informations, consultez la section Détection et synchronisation des conflits dans le manuel du AWS AppSync développeur. Ce champ est facultatif.

L'élément mis à jour dans DynamoDB est automatiquement converti en types primitifs GraphQL et JSON et est disponible dans le contexte de mappage (). `$context.result`

Pour plus d'informations sur la conversion de type DynamoDB, voir Système de types (mappage des réponses).

Pour plus d'informations sur les modèles de mappage des réponses, consultez la section Vue d'ensemble des modèles de mappage Resolver.

Exemple 1

L'exemple suivant est un modèle de mappage pour la mutation GraphQL. `upvote(id: ID!)`

Dans cet exemple, les champs et d'un élément de DynamoDB sont `upvotes` incrémentés de `version` 1.

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "update" : {
        "expression" : "ADD #votefield :plusOne, version :plusOne",
        "expressionNames" : {
            "#votefield" : "upvotes"
        },
        "expressionValues" : {
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

Exemple 2

L'exemple suivant est un modèle de mappage pour une mutation GraphQL. `updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)`

Il s'agit d'un exemple complexe qui inspecte les arguments et génère dynamiquement l'expression de mise à jour incluant uniquement les arguments qui ont été fournis par le client. Par exemple, si `title` et `author` sont omis, ils ne sont pas mis à jour. Si un argument est spécifié mais que sa valeur l'est`null`, ce champ est supprimé de l'objet dans DynamoDB. Enfin, l'opération comporte une condition qui vérifie si le champ de l'élément actuellement dans DynamoDB est `version` défini comme suit : `expectedVersion`

```
{
    "version" : "2017-02-28",

    "operation" : "UpdateItem",

    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },

    ## Set up some space to keep track of things we're updating **
```

```
    #set( $expNames  = {} )
    #set( $expValues = {} )
    #set( $expSet = {} )
    #set( $expAdd = {} )
    #set( $expRemove = [] )

    ## Increment "version" by 1 **
    $!{expAdd.put("version", ":newVersion")}
    $!{expValues.put(":newVersion", { "N" : 1 })}

    ## Iterate through each argument, skipping "id" and "expectedVersion" **
    #foreach( $entry in $context.arguments.entrySet() )
        #if( $entry.key != "id" && $entry.key != "expectedVersion" )
            #if( (!$entry.value) && ("$!{entry.value}" == "") )
                ## If the argument is set to "null", then remove that attribute from
the item in DynamoDB **

                #set( $discard = ${expRemove.add("#${entry.key}")} )
                $!{expNames.put("#${entry.key}", "$entry.key")}
            #else
                ## Otherwise set (or update) the attribute on the item in DynamoDB **

                $!{expSet.put("#${entry.key}", ":${entry.key}")}
                $!{expNames.put("#${entry.key}", "$entry.key")}

                #if( $entry.key == "ups" || $entry.key == "downs" )
                    $!{expValues.put(":${entry.key}", { "N" : $entry.value })}
                #else
                    $!{expValues.put(":${entry.key}", { "S" : "${entry.value}" })}
                #end
            #end
        #end
    #end

    ## Start building the update expression, starting with attributes we're going to
SET **
    #set( $expression = "" )
    #if( !${expSet.isEmpty()} )
        #set( $expression = "SET" )
        #foreach( $entry in $expSet.entrySet() )
            #set( $expression = "${expression} ${entry.key} = ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
```

```
        #end
    #end

    ## Continue building the update expression, adding attributes we're going to ADD **
    #if( !${expAdd.isEmpty()} )
        #set( $expression = "${expression} ADD" )
        #foreach( $entry in $expAdd.entrySet() )
            #set( $expression = "${expression} ${entry.key} ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Continue building the update expression, adding attributes we're going to REMOVE
**
    #if( !${expRemove.isEmpty()} )
        #set( $expression = "${expression} REMOVE" )

        #foreach( $entry in $expRemove )
            #set( $expression = "${expression} ${entry}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Finally, write the update expression into the document, along with any
expressionNames and expressionValues **
    "update" : {
        "expression" : "${expression}"
        #if( !${expNames.isEmpty()} )
            ,"expressionNames" : $utils.toJson($expNames)
        #end
        #if( !${expValues.isEmpty()} )
            ,"expressionValues" : $utils.toJson($expValues)
        #end
    },

    "condition" : {
        "expression"       : "version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" :
$util.dynamodb.toDynamoDBJson($ctx.args.expectedVersion)
```

```
        }
    }
}
```

Pour de plus amples informations sur l'API de DynamoDB `UpdateItem`, veuillez consulter la Documentation sur les API de DynamoDB.

DeleteItem

Le document de mappage des demandes vous permet de `DeleteItem` demander au résolveur AWS AppSync DynamoDB d'envoyer une `DeleteItem` demande à DynamoDB et de spécifier les éléments suivants :

- La clé de l'élément dans DynamoDB
- Conditions de réussite de l'opération

Le document de mappage `DeleteItem` a la structure suivante :

```
{
    "version" : "2018-05-29",
    "operation" : "DeleteItem",
    "customPartitionKey" : "foo",
    "populateIndexFields" : boolean value,
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "condition" : {
        ...
    },
    "_version" : 1
}
```

Les champs sont définis comme suit :

Champs de DeleteItem

DeleteItemliste des champs

`version`

Version de la définition du modèle. `2017-02-28` et `2018-05-29` sont actuellement prises en charge. Cette valeur est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `DeleteItem`, ce champ doit être défini sur `DeleteItem`. Cette valeur est obligatoire.

`key`

La clé de l'élément dans DynamoDB. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la spécification d'une « valeur saisie », consultez Système de types (mappage des demandes). Cette valeur est obligatoire.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Si aucune condition n'est spécifiée, la requête `DeleteItem` supprime un élément existant quel que soit son état actuel. Pour plus d'informations sur les conditions, consultez la section Expressions de condition. Cette valeur est facultative.

`_version`

Valeur numérique représentant la dernière version connue d'un élément. Cette valeur est facultative. Ce champ est utilisé pour la détection de conflits et n'est pris en charge que sur les sources de données versionnées.

`customPartitionKey`

Lorsqu'elle est activée, cette valeur de chaîne modifie le format des `ds_pk` enregistrements `ds_sk` et utilisés par la table de synchronisation delta lorsque le contrôle des versions a été activé (pour plus d'informations, voir Détection des conflits et synchronisation dans le guide du AWS AppSync développeur). Lorsque cette option est activée, le traitement de l'`populateIndexFields`entrée est également activé. Ce champ est facultatif.

`populateIndexFields`

Valeur booléenne qui, lorsqu'elle est activée en même temps que le **`customPartitionKey`**, crée de nouvelles entrées pour chaque enregistrement de la table de synchronisation delta, en particulier dans les colonnes `gsi_ds_pk` et`gsi_ds_sk`. Pour plus d'informations, consultez la section Détection et synchronisation des conflits dans le manuel du AWS AppSync développeur. Ce champ est facultatif.

L'élément supprimé de DynamoDB est automatiquement converti en types primitifs GraphQL et JSON et est disponible dans le contexte de mappage (). `$context.result`

Pour plus d'informations sur la conversion de type DynamoDB, voir Système de types (mappage des réponses).

Pour plus d'informations sur les modèles de mappage des réponses, consultez la section Vue d'ensemble des modèles de mappage Resolver.

Exemple 1

L'exemple suivant est un modèle de mappage pour une mutation GraphQL. `deleteItem(id: ID!)` S'il existe déjà un élément avec cet ID, il est supprimé.

```
{
    "version" : "2017-02-28",
    "operation" : "DeleteItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

Exemple 2

L'exemple suivant est un modèle de mappage pour une mutation GraphQL. `deleteItem(id: ID!, expectedVersion: Int!)` S'il existe déjà un élément avec cet ID, il est supprimé, mais uniquement si son champ `version` est défini sur `expectedVersion` :

```
{
    "version" : "2017-02-28",
    "operation" : "DeleteItem",
```

```
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "condition" : {
        "expression"       : "attribute_not_exists(id) OR version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" : $util.dynamodb.toDynamoDBJson($expectedVersion)
        }
    }
}
```

Pour de plus amples informations sur l'API de DynamoDB `DeleteItem`, veuillez consulter la Documentation sur les API de DynamoDB.

Query

Le document de mappage des demandes vous permet de `Query` demander au résolveur AWS AppSync DynamoDB d'envoyer une `Query` demande à DynamoDB et de spécifier les éléments suivants :

- Expression de la clé
- Quel index utiliser
- Tout filtre supplémentaire
- Combien d'articles renvoyer
- S'il convient d'utiliser une lecture cohérente
- Sens de la requête (vers l'avant ou l'arrière)
- Jeton de pagination

Le document de mappage `Query` a la structure suivante :

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "query" : {
        "expression" : "some expression",
        "expressionNames" : {
            "#foo" : "foo"
        },
```

```
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "index" : "fooIndex",
    "nextToken" : "a pagination token",
    "limit" : 10,
    "scanIndexForward" : true,
    "consistentRead" : false,
    "select" : "ALL_ATTRIBUTES" | "ALL_PROJECTED_ATTRIBUTES" | "SPECIFIC_ATTRIBUTES",
    "filter" : {
        ...
    },
    "projection" : {
        ...
    }
}
```

Les champs sont définis comme suit :

Champs de requête

Liste des champs de requête

`version`

Version de la définition du modèle. `2017-02-28` et `2018-05-29` sont actuellement prises en charge. Cette valeur est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `Query`, ce champ doit être défini sur `Query`. Cette valeur est obligatoire.

`query`

Cette `query` section vous permet de spécifier une expression de condition clé qui décrit les éléments à récupérer depuis DynamoDB. Pour plus d'informations sur la façon d'écrire des expressions de conditions clés, consultez la documentation KeyConditions DynamoDB. Cette section doit être spécifiée.

`expression`

Expression de la requête. Ce champ doit être spécifié.

`expressionNames`

Substituts des espaces réservés de nom des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé utilisé dans le`expression`, et la valeur doit être une chaîne correspondant au nom d'attribut de l'élément dans DynamoDB. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de nom des attributs de l'expression utilisés dans l'`expression`.

`expressionValues`

Substituts des espaces réservés de valeur des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé de valeur utilisé dans l'`expression`, et la valeur doit être typée. Pour plus d'informations sur la façon de spécifier une « valeur saisie », voir Système de types (mappage des demandes). Cette valeur est obligatoire. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de valeur des attributs de l'expression utilisés dans l'`expression`.

`filter`

Filtre supplémentaire qui peut être utilisé pour filtrer les résultats de DynamoDB avant qu'ils soient renvoyés. Pour plus d'informations sur les filtres, consultez Filters (Filtres). Ce champ est facultatif.

`index`

Nom de l'index à interroger. L'opération de requête DynamoDB vous permet de rechercher une clé de hachage sur les index secondaires locaux et les index secondaires globaux en plus de l'index de clé primaire. Si cela est spécifié, cela indique à DynamoDB d'interroger l'index spécifié. Si elle ne l'est pas, l'index de clé primaire est interrogé.

`nextToken`

Jeton de pagination pour continuer une requête précédente. Il a été obtenu à partir d'une requête précédente. Ce champ est facultatif.

`limit`

Nombre maximum d'éléments à évaluer (pas nécessairement le nombre d'éléments correspondants). Ce champ est facultatif.

`scanIndexForward`

Valeur booléenne indiquant s'il convient d'interroger vers l'avant ou vers l'arrière. Ce champ est facultatif et contient `true` par défaut.

`consistentRead`

Un booléen indiquant s'il faut utiliser des lectures cohérentes lors de l'interrogation de DynamoDB. Ce champ est facultatif et contient `false` par défaut.

`select`

Par défaut, le résolveur AWS AppSync DynamoDB renvoie uniquement les attributs projetés dans l'index. Si un plus grand nombre d'attributs est requis, vous pouvez définir ce champ. Ce champ est facultatif. Les valeurs prises en charge sont :

`ALL_ATTRIBUTES`

Renvoie tous les attributs de l'élément depuis la table ou l'index spécifié. Si vous interrogez un index secondaire local, DynamoDB extrait l'élément entier de la table parent pour chaque élément correspondant de l'index. Si l'index est configuré de façon à projeter tous les attributs de l'élément, toutes les données peuvent être obtenues à partir de l'index secondaire local, et aucune extraction n'est nécessaire.

`ALL_PROJECTED_ATTRIBUTES`

Autorisé seulement lorsque vous interrogez un index. Extrait tous les attributs qui ont été projetés dans l'index. Si l'index est configuré de façon à projeter tous les attributs, la valeur renvoyée revient à spécifier `ALL_ATTRIBUTES`.

`SPECIFIC_ATTRIBUTES`

Renvoie uniquement les attributs répertoriés dans le `projection` « s `expression` ». Cette valeur de retour revient à spécifier les « `projection` s » `expression` sans spécifier de valeur pour`Select`.

`projection`

Projection utilisée pour spécifier les attributs à renvoyer par l'opération DynamoDB. Pour plus d'informations sur les projections, voir Projections. Ce champ est facultatif.

Les résultats de DynamoDB sont automatiquement convertis en types primitifs GraphQL et JSON et sont disponibles dans le contexte de mappage (). `$context.result`

Pour plus d'informations sur la conversion de type DynamoDB, voir Système de types (mappage des réponses).

Pour plus d'informations sur les modèles de mappage des réponses, consultez la section Vue d'ensemble des modèles de mappage Resolver.

Les résultats ont la structure suivante :

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

Les champs sont définis comme suit :

`items`

Liste contenant les éléments renvoyés par la requête DynamoDB.

`nextToken`

S'il peut y avoir plusieurs résultats, `nextToken` contient un jeton de pagination que vous pouvez utiliser dans une autre requête. Notez que cela AWS AppSync chiffre et masque le jeton de pagination renvoyé par DynamoDB. Cela empêche que les données provenant de votre table ne soient accidentellement communiquées au mandataire. Notez également que ces jetons de pagination ne peuvent pas être utilisés dans les différents résolveurs.

`scannedCount`

Nombre d'éléments correspondant à l'expression de condition de requête, avant qu'une expression de filtre (si elle est présente) ne soit appliquée.

Exemple

L'exemple suivant est un modèle de mappage pour une requête GraphQL. `getPosts(owner: ID!)`

Dans cet exemple, un index secondaire global sur une table est interrogé afin de renvoyer toutes les publications détenues par l'ID spécifié.

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "query" : {
        "expression" : "ownerId = :ownerId",
        "expressionValues" : {
            ":ownerId" : $util.dynamodb.toDynamoDBJson($context.arguments.owner)
```

```
        }
    },
    "index" : "owner-index"
}
```

Pour de plus amples informations sur l'API de DynamoDB `Query`, veuillez consulter la Documentation sur les API de DynamoDB.

Analyser

Le document de mappage des demandes vous permet de `Scan` demander au résolveur AWS AppSync DynamoDB d'envoyer une `Scan` demande à DynamoDB et de spécifier les éléments suivants :

- Filtre pour exclure des résultats
- Quel index utiliser
- Combien d'articles renvoyer
- S'il convient d'utiliser une lecture cohérente
- Jeton de pagination
- Analyses parallèles

Le document de mappage `Scan` a la structure suivante :

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "index" : "fooIndex",
    "limit" : 10,
    "consistentRead" : false,
    "nextToken" : "aPaginationToken",
    "totalSegments" : 10,
    "segment" : 1,
    "filter" : {
        ...
    },
    "projection" : {
        ...
    }
}
```

Les champs sont définis comme suit :

Numériser les champs

Numériser la liste des champs

`version`

Version de la définition du modèle. `2017-02-28` et `2018-05-29` sont actuellement prises en charge. Cette valeur est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `Scan`, ce champ doit être défini sur `Scan`. Cette valeur est obligatoire.

`filter`

Filtre qui peut être utilisé pour filtrer les résultats de DynamoDB avant qu'ils ne soient renvoyés. Pour plus d'informations sur les filtres, consultez Filters (Filtres). Ce champ est facultatif.

`index`

Nom de l'index à interroger. L'opération de requête DynamoDB vous permet de rechercher une clé de hachage sur les index secondaires locaux et les index secondaires globaux en plus de l'index de clé primaire. Si cela est spécifié, cela indique à DynamoDB d'interroger l'index spécifié. Si elle ne l'est pas, l'index de clé primaire est interrogé.

`limit`

Nombre maximal d'éléments à évaluer simultanément. Ce champ est facultatif.

`consistentRead`

Un booléen qui indique s'il faut utiliser des lectures cohérentes lors de l'interrogation de DynamoDB. Ce champ est facultatif et contient `false` par défaut.

`nextToken`

Jeton de pagination pour continuer une requête précédente. Il a été obtenu à partir d'une requête précédente. Ce champ est facultatif.

`select`

Par défaut, le résolveur AWS AppSync DynamoDB renvoie uniquement les attributs projetés dans l'index. Si un plus grand nombre d'attributs est requis, ce champ peut être défini. Ce champ est facultatif. Les valeurs prises en charge sont :

`ALL_ATTRIBUTES`

Renvoie tous les attributs de l'élément depuis la table ou l'index spécifié. Si vous interrogez un index secondaire local, DynamoDB extrait l'élément entier de la table parent pour chaque élément correspondant de l'index. Si l'index est configuré de façon à projeter tous les attributs de l'élément, toutes les données peuvent être obtenues à partir de l'index secondaire local, et aucune extraction n'est nécessaire.

`ALL_PROJECTED_ATTRIBUTES`

Autorisé seulement lorsque vous interrogez un index. Extrait tous les attributs qui ont été projetés dans l'index. Si l'index est configuré de façon à projeter tous les attributs, la valeur renvoyée revient à spécifier `ALL_ATTRIBUTES`.

`SPECIFIC_ATTRIBUTES`

Renvoie uniquement les attributs répertoriés dans le `projection` « s `expression` ». Cette valeur de retour revient à spécifier les « `projection` s » `expression` sans spécifier de valeur pour`Select`.

`totalSegments`

Nombre de segments pour partitionner la table lors de l'exécution d'une analyse parallèle. Ce champ est facultatif, mais doit être spécifié si `segment` est spécifié.

`segment`

Segment de table de cette opération lorsque vous effectuez une analyse parallèle. Ce champ est facultatif, mais doit être spécifié si `totalSegments` est spécifié.

`projection`

Projection utilisée pour spécifier les attributs à renvoyer par l'opération DynamoDB. Pour plus d'informations sur les projections, voir Projections. Ce champ est facultatif.

Les résultats renvoyés par le scan DynamoDB sont automatiquement convertis en types primitifs GraphQL et JSON et sont disponibles dans le contexte de mappage (). `$context.result`

Pour plus d'informations sur la conversion de type DynamoDB, voir Système de types (mappage des réponses).

Pour plus d'informations sur les modèles de mappage des réponses, consultez la section Vue d'ensemble des modèles de mappage Resolver.

Les résultats ont la structure suivante :

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

Les champs sont définis comme suit :

`items`

Liste contenant les éléments renvoyés par le scan DynamoDB.

`nextToken`

S'il peut y avoir plusieurs résultats, `nextToken` contient un jeton de pagination que vous pouvez utiliser dans une autre requête. AWS AppSync chiffre et masque le jeton de pagination renvoyé par DynamoDB. Cela empêche que les données provenant de votre table ne soient accidentellement communiquées au mandataire. D'autre part, ces jetons de pagination ne peuvent pas être utilisés dans différents résolveurs.

`scannedCount`

Nombre d'éléments récupérés par DynamoDB avant l'application d'une expression de filtre (le cas échéant).

Exemple 1

L'exemple suivant est un modèle de mappage pour la requête GraphQL :. `allPosts`

Dans cet exemple, toutes les entrées de la table sont renvoyées.

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
}
```

Exemple 2

L'exemple suivant est un modèle de mappage pour la requête GraphQL :. `postsMatching(title: String!)`

Dans cet exemple, toutes les entrées de la table sont renvoyées lorsque le titre commence par l'argument `title`.

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "filter" : {
        "expression" : "begins_with(title, :title)",
        "expressionValues" : {
            ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
        },
    }
}
```

Pour de plus amples informations sur l'API de DynamoDB `Scan`, veuillez consulter la Documentation sur les API de DynamoDB.

Sync

Le document de mappage des `Sync` demandes vous permet de récupérer tous les résultats d'une table DynamoDB, puis de ne recevoir que les données modifiées depuis votre dernière requête (le delta est mis à jour). `Sync`les demandes ne peuvent être adressées qu'à des sources de données DynamoDB versionnées. Vous pouvez spécifier les valeurs suivantes :

- Filtre pour exclure des résultats
- Combien d'articles renvoyer
- Jeton de pagination.
- Lorsque votre dernière opération `Sync` a été lancée

Le document de mappage `Sync` a la structure suivante :

```
{
    "version" : "2018-05-29",
    "operation" : "Sync",
    "basePartitionKey": "Base Tables PartitionKey",
    "deltaIndexName": "delta-index-name",
    "limit" : 10,
    "nextToken" : "aPaginationToken",
    "lastSync" :  1550000000000,
```

```
    "filter" : {
        ...
    }
}
```

Les champs sont définis comme suit :

Synchroniser les champs

Liste des champs de synchronisation

`version`

Version de la définition du modèle. `2018-05-29` est le seul à être pris en charge actuellement. Cette valeur est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération `Sync`, ce champ doit être défini sur `Sync`. Cette valeur est obligatoire.

`filter`

Filtre qui peut être utilisé pour filtrer les résultats de DynamoDB avant qu'ils ne soient renvoyés. Pour plus d'informations sur les filtres, consultez Filters (Filtres). Ce champ est facultatif.

`limit`

Nombre maximal d'éléments à évaluer simultanément. Ce champ est facultatif. Si cette option est omise, la limite par défaut sera définie sur `100` éléments. La valeur maximale de ce champ est `1000` éléments.

`nextToken`

Jeton de pagination pour continuer une requête précédente. Il a été obtenu à partir d'une requête précédente. Ce champ est facultatif.

`lastSync`

Le moment, en millisecondes Epoch, où la dernière opération `Sync` réussie a commencé. Si spécifié, seuls les éléments qui ont changé après `lastSync` sont retournés. Ce champ est facultatif et ne doit être renseigné qu'après avoir récupéré toutes les pages d'une opération `Sync` initiale. Si cette option est omise, les résultats de la table Base seront retournés, sinon les résultats de la table Delta seront retournés.

`basePartitionKey`

La clé de partition de la table de base utilisée lors de l'exécution d'une `Sync` opération. Ce champ permet d'effectuer une `Sync` opération lorsque la table utilise une clé de partition personnalisée. Il s'agit d'un champ facultatif.

`deltaIndexName`

L'index utilisé pour l'`Sync`opération. Cet index est nécessaire pour permettre une `Sync` opération sur l'ensemble de la table delta store lorsque la table utilise une clé de partition personnalisée. L'`Sync`opération sera effectuée sur le GSI (créé sur `gsi_ds_pk` et`gsi_ds_sk`). Ce champ est facultatif.

Les résultats renvoyés par la synchronisation DynamoDB sont automatiquement convertis en types primitifs GraphQL et JSON et sont disponibles dans le contexte de mappage (). `$context.result`

Pour plus d'informations sur la conversion de type DynamoDB, voir Système de types (mappage des réponses).

Pour plus d'informations sur les modèles de mappage des réponses, consultez la section Vue d'ensemble des modèles de mappage Resolver.

Les résultats ont la structure suivante :

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10,
    startedAt = 1550000000000
}
```

Les champs sont définis comme suit :

`items`

Liste contenant les éléments renvoyés par la synchronisation.

`nextToken`

S'il peut y avoir plusieurs résultats, `nextToken` contient un jeton de pagination que vous pouvez utiliser dans une autre requête. AWS AppSync chiffre et masque le jeton de pagination

renvoyé par DynamoDB. Cela empêche que les données provenant de votre table ne soient accidentellement communiquées au mandataire. D'autre part, ces jetons de pagination ne peuvent pas être utilisés dans différents résolveurs.

`scannedCount`

Nombre d'éléments récupérés par DynamoDB avant l'application d'une expression de filtre (le cas échéant).

`startedAt`

Le moment, en millisecondes Epoch, où l'opération de synchronisation a commencé, que vous pouvez stocker localement et utiliser dans une autre requête comme argument `lastSync`. Si un jeton de pagination a été inclus dans la requête, cette valeur sera la même que celle renvoyée par la requête pour la première page de résultats.

Exemple 1

L'exemple suivant est un modèle de mappage pour la requête GraphQL :. `syncPosts(nextToken: String, lastSync: AWSTimestamp)`

Dans cet exemple, si `lastSync` est omis, toutes les entrées de la table de base sont renvoyées. Si `lastSync` est fourni, seules les entrées de la table de synchronisation delta qui ont changé depuis `lastSync` sont renvoyées.

```
{
    "version" : "2018-05-29",
    "operation" : "Sync",
    "limit": 100,
    "nextToken": $util.toJson($util.defaultIfNull($ctx.args.nextToken, null)),
    "lastSync": $util.toJson($util.defaultIfNull($ctx.args.lastSync, null))
}
```

BatchGetItem

Le document de mappage des demandes vous permet de `BatchGetItem` demander au résolveur AWS AppSync DynamoDB d'envoyer une `BatchGetItem` demande à DynamoDB pour récupérer plusieurs éléments, éventuellement sur plusieurs tables. Pour ce modèle de requête, vous devez spécifier les valeurs suivantes :

- Les noms de tables à partir desquels récupérer les éléments

- Les clés des éléments à récupérer dans chaque table

Les limites `BatchGetItem` DynamoDB s'appliquent et aucune expression de condition ne peut être fournie.

Le document de mappage `BatchGetItem` a la structure suivante :

```
{
    "version" : "2018-05-29",
    "operation" : "BatchGetItem",
    "tables" : {
        "table1": {
           "keys": [
              ## Item to retrieve Key
              {
                   "foo" : ... typed value,
                   "bar" : ... typed value
              },
              ## Item2 to retrieve Key
              {
                   "foo" : ... typed value,
                   "bar" : ... typed value
              }
            ],
            "consistentRead": true|false,
            "projection" : {
                ...
            }
        },
        "table2": {
           "keys": [
              ## Item3 to retrieve Key
              {
                   "foo" : ... typed value,
                   "bar" : ... typed value
              },
              ## Item4 to retrieve Key
              {
                   "foo" : ... typed value,
                   "bar" : ... typed value
              }
            ],
            "consistentRead": true|false,
```

```
            "projection" : {
                 ...
            }
        }
    }
}
```

Les champs sont définis comme suit :

Champs de BatchGetItem

BatchGetItemliste des champs

`version`

Version de la définition du modèle. Seul `2018-05-29` est pris en charge. Cette valeur est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `BatchGetItem`, ce champ doit être défini sur `BatchGetItem`. Cette valeur est obligatoire.

`tables`

Les tables DynamoDB à partir desquelles récupérer les éléments. La valeur est une carte où les noms de table sont spécifiés en tant que clés de la carte. Vous devez fournir au moins une table. Cette valeur `tables` est obligatoire.

`keys`

Liste des clés DynamoDB représentant la clé primaire des éléments à récupérer. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la façon de spécifier une « valeur saisie », voir Système de types (mappage des demandes).

`consistentRead`

S'il faut utiliser une lecture cohérente lors de l'exécution d'une GetItemopération. Cette valeur est facultative et est définie comme faux par défaut.

`projection`

Projection utilisée pour spécifier les attributs à renvoyer par l'opération DynamoDB. Pour plus d'informations sur les projections, voir Projections. Ce champ est facultatif.

Objets à mémoriser :

- Si un élément n'a pas été récupéré à partir de la table, un élément null s'affiche dans le bloc de données pour cette table.
- Les résultats d'invocation sont triés par table, en fonction de l'ordre dans lequel ils ont été fournis dans le modèle de mappage des demandes.
- Chaque `Get` commande contenue dans a `BatchGetItem` est atomique, mais un lot peut être partiellement traité. Si un lot est traité partiellement en raison d'une erreur, les clés non traitées sont renvoyées dans le cadre du résultat de l'appel dans le bloc unprocessedKeys.
- `BatchGetItem` est limité à 100 clés.

Pour l'exemple de modèle de mappage de requête suivant :

```
{
  "version": "2018-05-29",
  "operation": "BatchGetItem",
  "tables": {
    "authors": [
        {
          "author_id": {
            "S": "a1"
          }
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          }
        }
    ],
  }
}
```

Le résultat de l'appel disponible dans `$ctx.result` est le suivant :

```
{
```

```
    "data": {
      "authors": [null],
      "posts": [
         # Was retrieved
         {
           "author_id": "a1",
           "post_id": "p2",
           "post_title": "title",
           "post_description": "description",
         }
      ]
    },
    "unprocessedKeys": {
      "authors": [
         # This item was not processed due to an error
         {
           "author_id": "a1"
         }
       ],
      "posts": []
    }
}
```

`$ctx.error` contient des détails sur l'erreur. Les données clés, unprocessedKeys et la clé de chaque table fournie dans le modèle de mappage de requête sont assurés d'être présents dans le résultat de l'appel. Les éléments ayant été supprimés apparaissent dans le bloc de données. Les éléments qui n'ont pas été traités sont marqués comme null dans le bloc de données et sont placés dans le bloc unprocessedKeys.

Pour un exemple plus complet, suivez le didacticiel DynamoDB Batch avec ici AppSync Tutoriel : Résolveurs de lots DynamoDB.

BatchDeleteItem

Le document de mappage des demandes vous permet de `BatchDeleteItem` demander au résolveur AWS AppSync DynamoDB de `BatchWriteItem` demander à DynamoDB de supprimer plusieurs éléments, éventuellement sur plusieurs tables. Pour ce modèle de requête, vous devez spécifier les valeurs suivantes :

- Les noms de tables à partir desquels supprimer les éléments
- Les clés des éléments à supprimer dans chaque table

Les limites `BatchWriteItem` DynamoDB s'appliquent et aucune expression de condition ne peut être fournie.

Le document de mappage `BatchDeleteItem` a la structure suivante :

```
{
    "version" : "2018-05-29",
    "operation" : "BatchDeleteItem",
    "tables" : {
        "table1": [
        ## Item to delete Key
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        },
        ## Item2 to delete Key
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        }],
        "table2": [
        ## Item3 to delete Key
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        },
        ## Item4 to delete Key
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        }],
    }
}
```

Les champs sont définis comme suit :

Champs de BatchDeleteItem

BatchDeleteItemliste des champs

`version`

Version de la définition du modèle. Seul `2018-05-29` est pris en charge. Cette valeur est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `BatchDeleteItem`, ce champ doit être défini sur `BatchDeleteItem`. Cette valeur est obligatoire.

`tables`

Les tables DynamoDB dont les éléments doivent être supprimés. Chaque table est une liste de clés DynamoDB représentant la clé primaire des éléments à supprimer. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la façon de spécifier une « valeur saisie », voir Système de types (mappage des demandes). Vous devez fournir au moins une table. La `tables` valeur est obligatoire.

Objets à mémoriser :

- Contrairement à l'opération `DeleteItem`, l'élément complètement supprimé n'est pas renvoyé dans la réponse. Seule la clé passée est renvoyée.
- Si un élément n'a pas été supprimé à partir de la table, un élément null s'affiche dans le bloc de données pour cette table.
- Les résultats d'invocation sont triés par table, en fonction de l'ordre dans lequel ils ont été fournis dans le modèle de mappage des demandes.
- Chaque `Delete` commande à l'intérieur de a `BatchDeleteItem` est atomique. Cependant, un lot peut être partiellement traité. Si un lot est traité partiellement en raison d'une erreur, les clés non traitées sont renvoyées dans le cadre du résultat de l'appel dans le bloc unprocessedKeys.
- `BatchDeleteItem` est limité à 25 clés.

Pour l'exemple de modèle de mappage de requête suivant :

```
{
  "version": "2018-05-29",
  "operation": "BatchDeleteItem",
  "tables": {
    "authors": [
        {
          "author_id": {
            "S": "a1"
          }
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          }
        }
    ],
  }
}
```

Le résultat de l'appel disponible dans `$ctx.result` est le suivant :

```
{
   "data": {
     "authors": [null],
     "posts": [
        # Was deleted
        {
          "author_id": "a1",
          "post_id": "p2"
        }
     ]
   },
   "unprocessedKeys": {
     "authors": [
        # This key was not processed due to an error
        {
          "author_id": "a1"
        }
```

```
      ],
      "posts": []
   }
}
```

`$ctx.error` contient des détails sur l'erreur. Les données clés, unprocessedKeys et la clé de chaque table fournie dans le modèle de mappage de requête sont assurés d'être présents dans le résultat de l'appel. Les éléments ayant été supprimés sont présents dans le bloc de données. Les éléments qui n'ont pas été traités sont marqués comme null dans le bloc de données et sont placés dans le bloc unprocessedKeys.

Pour un exemple plus complet, suivez le didacticiel DynamoDB Batch avec ici AppSync Tutoriel : Résolveurs de lots DynamoDB.

BatchPutItem

Le document de mappage des demandes vous permet de `BatchPutItem` demander au résolveur AWS AppSync DynamoDB d'envoyer une `BatchWriteItem` demande à DynamoDB pour placer plusieurs éléments, éventuellement sur plusieurs tables. Pour ce modèle de requête, vous devez spécifier les valeurs suivantes :

- Les noms de tables dans lesquels placer les éléments
- Les éléments complets à placer dans chaque table

Les limites `BatchWriteItem` DynamoDB s'appliquent et aucune expression de condition ne peut être fournie.

Le document de mappage `BatchPutItem` a la structure suivante :

```
{
    "version" : "2018-05-29",
    "operation" : "BatchPutItem",
    "tables" : {
        "table1": [
        ## Item to put
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        },
        ## Item2 to put
```

```
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        }],
        "table2": [
        ## Item3 to put
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        },
        ## Item4 to put
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        }],
    }
}
```

Les champs sont définis comme suit :

Champs de BatchPutItem

BatchPutItemliste des champs

`version`

Version de la définition du modèle. Seul `2018-05-29` est pris en charge. Cette valeur est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `BatchPutItem`, ce champ doit être défini sur `BatchPutItem`. Cette valeur est obligatoire.

`tables`

Les tables DynamoDB dans lesquelles placer les éléments. Chaque entrée de table représente une liste d'éléments DynamoDB à insérer pour cette table spécifique. Vous devez fournir au moins une table. Cette valeur est obligatoire.

Objets à mémoriser :

- Les éléments entièrement insérés sont renvoyés dans la réponse, en cas de succès.

- Si un élément n'a pas été inséré dans la table, un élément null s'affiche dans le bloc de données pour cette table.
- Les éléments insérés sont triés par table, en fonction de l'ordre dans lequel ils ont été fournis dans le modèle de mappage des demandes.
- Chaque `Put` commande contenue dans a `BatchPutItem` est atomique, mais un lot peut être partiellement traité. Si un lot est traité partiellement en raison d'une erreur, les clés non traitées sont renvoyées dans le cadre du résultat de l'appel dans le bloc unprocessedKeys.
- `BatchPutItem` est limité à 25 éléments.

Pour l'exemple de modèle de mappage de requête suivant :

```
{
  "version": "2018-05-29",
  "operation": "BatchPutItem",
  "tables": {
    "authors": [
        {
          "author_id": {
            "S": "a1"
          },
          "author_name": {
            "S": "a1_name"
          }
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          },
          "post_title": {
            "S": "title"
          }
        }
    ],
  }
}
```

Le résultat de l'appel disponible dans `$ctx.result` est le suivant :

```
{
   "data": {
     "authors": [
         null
     ],
     "posts": [
        # Was inserted
        {
          "author_id": "a1",
          "post_id": "p2",
          "post_title": "title"
        }
     ]
   },
   "unprocessedItems": {
     "authors": [
        # This item was not processed due to an error
        {
          "author_id": "a1",
          "author_name": "a1_name"
        }
      ],
     "posts": []
   }
}
```

`$ctx.error` contient des détails sur l'erreur. Les données clés, unprocessedItems et la clé de chaque table fournie dans le modèle de mappage de requête sont assurés d'être présents dans le résultat de l'appel. Les éléments ayant été insérés sont dans le bloc de données. Les éléments qui n'ont pas été traités sont marqués comme null dans le bloc de données et sont placés dans le bloc unprocessedItems.

Pour un exemple plus complet, suivez le didacticiel DynamoDB Batch avec ici AppSync Tutoriel : Résolveurs de lots DynamoDB.

TransactGetItems

Le document de mappage des demandes vous permet de `TransactGetItems` demander au résolveur AWS AppSync DynamoDB d'envoyer une `TransactGetItems` demande à DynamoDB

pour récupérer plusieurs éléments, éventuellement sur plusieurs tables. Pour ce modèle de requête, vous devez spécifier les valeurs suivantes :

- Nom de la table de chaque élément de requête dans lequel extraire l'élément
- La clé de chaque élément de requête à récupérer à partir de chaque table

Les limites `TransactGetItems` DynamoDB s'appliquent et aucune expression de condition ne peut être fournie.

Le document de mappage `TransactGetItems` a la structure suivante :

```
{
    "version": "2018-05-29",
    "operation": "TransactGetItems",
    "transactItems": [
       ## First request item
       {
           "table": "table1",
           "key": {
               "foo": ... typed value,
               "bar": ... typed value
           },
           "projection" : {
                ...
           }
       },
       ## Second request item
       {
           "table": "table2",
           "key": {
               "foo": ... typed value,
               "bar": ... typed value
           },
           "projection" : {
                ...
           }
       }
    ]
}
```

Les champs sont définis comme suit :

Champs de TransactGetItems

TransactGetItemsliste des champs

`version`

Version de la définition du modèle. Seul `2018-05-29` est pris en charge. Cette valeur est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `TransactGetItems`, ce champ doit être défini sur `TransactGetItems`. Cette valeur est obligatoire.

`transactItems`

Les éléments de requête à inclure. La valeur est un tableau d'éléments de requête. Au moins un élément de requête doit être fourni. Cette valeur `transactItems` est obligatoire.

`table`

La table DynamoDB à partir de laquelle récupérer l'élément. La valeur est une chaîne du nom de la table. Cette valeur `table` est obligatoire.

`key`

La clé DynamoDB représentant la clé primaire de l'élément à récupérer. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la façon de spécifier une « valeur saisie », voir Système de types (mappage des demandes).

`projection`

Projection utilisée pour spécifier les attributs à renvoyer par l'opération DynamoDB. Pour plus d'informations sur les projections, voir Projections. Ce champ est facultatif.

Objets à mémoriser :

- Si une transaction réussit, l'ordre des éléments récupérés dans le bloc `items` sera le même que celui des éléments de la requête.
- Les transactions sont effectuées d'une all-or-nothing manière ou d'une autre. Si un élément de requête provoque une erreur, la transaction entière ne sera pas effectuée et les détails de l'erreur seront retournés.

- Un élément de requête qui ne peut pas être récupéré n'est pas une erreur. Au lieu de cela, un élément null apparaît dans le bloc éléments dans la position correspondante.
- Si l'erreur d'une transaction est TransactionCanceledExceptionsurvenue, le `cancellationReasons` bloc sera rempli. L'ordre des motifs d'annulation dans le bloc `cancellationReasons` sera le même que l'ordre des éléments de demande.
- `TransactGetItems` est limité à 25 éléments de demande.

Pour l'exemple de modèle de mappage de requête suivant :

```
{
    "version": "2018-05-29",
    "operation": "TransactGetItems",
    "transactItems": [
       ## First request item
       {
           "table": "posts",
           "key": {
               "post_id": {
                 "S": "p1"
               }
           }
       },
       ## Second request item
       {
           "table": "authors",
           "key": {
               "author_id": {
                 "S": a1
               }
           }
       }
    ]
}
```

Si la transaction réussit et que seul le premier élément demandé est extrait, le résultat d'appel disponible dans `$ctx.result` est le suivant :

```
{
    "items": [
       {
           // Attributes of the first requested item
```

```
            "post_id": "p1",
            "post_title": "title",
            "post_description": "description"
        },
        // Could not retrieve the second requested item
        null,
    ],
    "cancellationReasons": null
}
```

Si la transaction échoue en raison TransactionCanceledExceptiondu premier élément de demande, le résultat de l'invocation disponible `$ctx.result` est le suivant :

```
{
    "items": null,
    "cancellationReasons": [
        {
            "type":"Sample error type",
            "message":"Sample error message"
        },
        {
            "type":"None",
            "message":"None"
        }
    ]
}
```

`$ctx.error` contient des détails sur l'erreur. Les éléments clés et cancellationReasons sont nécessairement présents dans `$ctx.result`.

Pour un exemple plus complet, suivez le didacticiel sur les transactions DynamoDB AppSync avec ici Tutoriel : résolveurs de transactions DynamoDB.

TransactWriteItems

Le document de mappage des demandes vous permet de `TransactWriteItems` demander au résolveur AWS AppSync DynamoDB de `TransactWriteItems` demander à DynamoDB d'écrire plusieurs éléments, éventuellement dans plusieurs tables. Pour ce modèle de requête, vous devez spécifier les valeurs suivantes :

- Nom de la table de destination de chaque élément de requête

- L'opération de chaque élément de demande à effectuer. Quatre types d'opérations sont pris en charge : PutItemUpdateItem, DeleteItem, et ConditionCheck
- La clé de chaque élément de demande à écrire

Les limites `TransactWriteItems` DynamoDB s'appliquent.

Le document de mappage `TransactWriteItems` a la structure suivante :

```
{
    "version": "2018-05-29",
    "operation": "TransactWriteItems",
    "transactItems": [
       {
           "table": "table1",
           "operation": "PutItem",
           "key": {
               "foo": ... typed value,
               "bar": ... typed value
           },
           "attributeValues": {
               "baz": ... typed value
           },
           "condition": {
               "expression": "someExpression",
               "expressionNames": {
                   "#foo": "foo"
               },
               "expressionValues": {
                   ":bar": ... typed value
               },
               "returnValuesOnConditionCheckFailure": true|false
           }
       },
       {
           "table":"table2",
           "operation": "UpdateItem",
           "key": {
               "foo": ... typed value,
               "bar": ... typed value
           },
           "update": {
               "expression": "someExpression",
```

```
            "expressionNames": {
                "#foo": "foo"
            },
            "expressionValues": {
                ":bar": ... typed value
            }
        },
        "condition": {
            "expression": "someExpression",
            "expressionNames": {
                "#foo":"foo"
            },
            "expressionValues": {
                ":bar": ... typed value
            },
            "returnValuesOnConditionCheckFailure": true|false
        }
    },
    {
        "table": "table3",
        "operation": "DeleteItem",
        "key":{
            "foo": ... typed value,
            "bar": ... typed value
        },
        "condition":{
            "expression": "someExpression",
            "expressionNames": {
                "#foo": "foo"
            },
            "expressionValues": {
                ":bar": ... typed value
            },
            "returnValuesOnConditionCheckFailure": true|false
        }
    },
    {
        "table": "table4",
        "operation": "ConditionCheck",
        "key":{
            "foo": ... typed value,
            "bar": ... typed value
        },
        "condition":{
```

```
                "expression": "someExpression",
                "expressionNames": {
                    "#foo": "foo"
                },
                "expressionValues": {
                    ":bar": ... typed value
                },
                "returnValuesOnConditionCheckFailure": true|false
            }
        }
    ]
}
```

Champs de TransactWriteItems

TransactWriteItemsliste des champs

Les champs sont définis comme suit :

`version`

Version de la définition du modèle. Seul `2018-05-29` est pris en charge. Cette valeur est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `TransactWriteItems`, ce champ doit être défini sur `TransactWriteItems`. Cette valeur est obligatoire.

`transactItems`

Les éléments de requête à inclure. La valeur est un tableau d'éléments de requête. Au moins un élément de requête doit être fourni. Cette valeur `transactItems` est obligatoire.

Pour `PutItem`, les champs sont définis comme suit :

`table`

La table DynamoDB de destination. La valeur est une chaîne du nom de la table. Cette valeur `table` est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `PutItem`, ce champ doit être défini sur `PutItem`. Cette valeur est obligatoire.

`key`

La clé DynamoDB représentant la clé primaire de l'élément à insérer. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la façon de spécifier une « valeur saisie », voir Système de types (mappage des demandes). Cette valeur est obligatoire.

`attributeValues`

Le reste des attributs de l'élément doit être placé dans DynamoDB. Pour plus d'informations sur la façon de spécifier une « valeur saisie », voir Système de types (mappage des demandes). Ce champ est facultatif.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Si aucune condition n'est spécifiée, la requête `PutItem` remplace toute entrée existante pour cet élément. Vous pouvez spécifier s'il faut récupérer l'élément existant en cas d'échec de la vérification de l'état. Pour plus d'informations sur les conditions transactionnelles, consultez la section Expressions des conditions de transaction. Cette valeur est facultative.

Pour `UpdateItem`, les champs sont définis comme suit :

`table`

Table DynamoDB à mettre à jour. La valeur est une chaîne du nom de la table. Cette valeur `table` est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `UpdateItem`, ce champ doit être défini sur `UpdateItem`. Cette valeur est obligatoire.

`key`

La clé DynamoDB représentant la clé primaire de l'élément à mettre à jour. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la façon de spécifier une « valeur saisie », voir Système de types (mappage des demandes). Cette valeur est obligatoire.

`update`

La `update` section vous permet de spécifier une expression de mise à jour qui décrit comment mettre à jour l'élément dans DynamoDB. Pour plus d'informations sur la façon d'écrire des expressions de mise à jour, consultez la documentation UpdateExpressions DynamoDB. Cette section est obligatoire.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Si aucune condition n'est spécifiée, la requête `UpdateItem` met à jour l'entrée existante quel que soit son état actuel. Vous pouvez spécifier s'il faut récupérer l'élément existant en cas d'échec de la vérification de l'état. Pour plus d'informations sur les conditions transactionnelles, consultez la section Expressions des conditions de transaction. Cette valeur est facultative.

Pour `DeleteItem`, les champs sont définis comme suit :

`table`

Table DynamoDB dans laquelle vous souhaitez supprimer l'élément. La valeur est une chaîne du nom de la table. Cette valeur `table` est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `DeleteItem`, ce champ doit être défini sur `DeleteItem`. Cette valeur est obligatoire.

`key`

La clé DynamoDB représentant la clé primaire de l'élément à supprimer. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la façon de spécifier une « valeur saisie », voir Système de types (mappage des demandes). Cette valeur est obligatoire.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Si aucune condition n'est spécifiée, la requête `DeleteItem` supprime un élément existant quel que soit son état actuel. Vous pouvez spécifier s'il faut récupérer l'élément existant en cas d'échec de la vérification de l'état. Pour

plus d'informations sur les conditions transactionnelles, consultez la section Expressions des conditions de transaction. Cette valeur est facultative.

Pour `ConditionCheck`, les champs sont définis comme suit :

`table`

Table DynamoDB dans laquelle vérifier la condition. La valeur est une chaîne du nom de la table. Cette valeur `table` est obligatoire.

`operation`

L'opération DynamoDB à effectuer. Pour que vous puissiez effectuer l'opération DynamoDB `ConditionCheck`, ce champ doit être défini sur `ConditionCheck`. Cette valeur est obligatoire.

`key`

La clé DynamoDB représentant la clé primaire de l'élément à vérifier. Les éléments DynamoDB peuvent avoir une seule clé de hachage ou une clé de hachage et une clé de tri, selon la structure de la table. Pour plus d'informations sur la façon de spécifier une « valeur saisie », voir Système de types (mappage des demandes). Cette valeur est obligatoire.

`condition`

Condition permettant de déterminer si la demande doit réussir ou non, en fonction de l'état de l'objet déjà dans DynamoDB. Vous pouvez spécifier s'il faut récupérer l'élément existant en cas d'échec de la vérification de l'état. Pour plus d'informations sur les conditions transactionnelles, consultez la section Expressions des conditions de transaction. Cette valeur est obligatoire.

Objets à mémoriser :

- Seules les clés des éléments de demande sont renvoyées dans la réponse, si elles réussissent. L'ordre des clés sera le même que l'ordre des éléments de demande.
- Les transactions sont effectuées d'une all-or-nothing manière ou d'une autre. Si un élément de requête provoque une erreur, la transaction entière ne sera pas effectuée et les détails de l'erreur seront retournés.
- Aucun élément de demande ne peut cibler le même élément. Sinon, ils provoqueront une TransactionCanceledExceptionerreur.

- Si l'erreur d'une transaction est TransactionCanceledExceptionsurvenue, le `cancellationReasons` bloc sera rempli. Si la vérification de l'état d'un élément de demande échoue et que vous n'avez pas spécifié `returnValuesOnConditionCheckFailure` comme étant `false`, l'élément existant dans la table est récupéré et stocké `item` à la position correspondante du bloc `cancellationReasons`.
- `TransactWriteItems` est limité à 25 éléments de demande.

Pour l'exemple de modèle de mappage de requête suivant :

```
{
    "version": "2018-05-29",
    "operation": "TransactWriteItems",
    "transactItems": [
       {
           "table": "posts",
           "operation": "PutItem",
           "key": {
               "post_id": {
                   "S": "p1"
               }
           },
           "attributeValues": {
               "post_title": {
                   "S": "New title"
               },
               "post_description": {
                   "S": "New description"
               }
           },
           "condition": {
               "expression": "post_title = :post_title",
               "expressionValues": {
                   ":post_title": {
                       "S": "Expected old title"
                   }
               }
           }
       },
       {
           "table":"authors",
           "operation": "UpdateItem",
```

```
            "key": {
                "author_id": {
                    "S": "a1"
                },
            },
            "update": {
                "expression": "SET author_name = :author_name",
                "expressionValues": {
                    ":author_name": {
                        "S": "New name"
                    }
                }
            },
        }
    ]
}
```

Si la transaction réussit, le résultat d'appel disponible dans `$ctx.result` est le suivant :

```
{
    "keys": [
       // Key of the PutItem request
       {
           "post_id": "p1",
       },
       // Key of the UpdateItem request
       {
           "author_id": "a1"
       }
    ],
    "cancellationReasons": null
}
```

Si la transaction échoue en raison de l'échec de la vérification de l'état de la `PutItem` demande, le résultat de l'invocation disponible `$ctx.result` est le suivant :

```
{
    "keys": null,
    "cancellationReasons": [
       {
           "item": {
               "post_id": "p1",
               "post_title": "Actual old title",
```

```
                "post_description": "Old description"
            },
            "type": "ConditionCheckFailed",
            "message": "The condition check failed."
        },
        {
            "type": "None",
            "message": "None"
        }
    ]
}
```

`$ctx.error` contient des détails sur l'erreur. Les clés clés et cancellationReasons sont nécessairement présentes dans `$ctx.result`.

Pour un exemple plus complet, suivez le didacticiel sur les transactions DynamoDB AppSync avec ici Tutoriel : résolveurs de transactions DynamoDB.

Système de types (mappage des demandes)

Lorsque vous utilisez le résolveur AWS AppSync DynamoDB pour appeler vos tables DynamoDB AWS AppSync , vous devez connaître le type de chaque valeur à utiliser dans cet appel. Cela est dû au fait que DynamoDB prend en charge un plus grand nombre de primitives de type que GraphQL ou JSON (telles que les ensembles et les données binaires). AWS AppSync a besoin de quelques conseils lors de la traduction entre GraphQL et DynamoDB, sinon il devra émettre des hypothèses sur la manière dont les données sont structurées dans votre table.

Pour plus d'informations sur les types de données DynamoDB, consultez les descripteurs de types de données DynamoDB et la documentation sur les types de données.

Une valeur DynamoDB est représentée par un objet JSON contenant une seule paire clé-valeur. La clé indique le type DynamoDB et la valeur indique la valeur elle-même. Dans l'exemple suivant, la clé `S` indique que la valeur est une chaîne, et la valeur `identifier` est la valeur de chaîne elle-même.

```
{ "S" : "identifier" }
```

Notez que l'objet JSON ne peut pas avoir plus d'une paire clé-valeur. Si plusieurs paires clé-valeur sont spécifiées, le document de mappage des requêtes n'est pas analysé.

Une valeur DynamoDB est utilisée n'importe où dans un document de mappage de requêtes où vous devez spécifier une valeur. Vous devrez notamment procéder ainsi dans les sections suivantes : `key`

et `attributeValue`, ainsi que la section `expressionValues` des sections d'expression. Dans l'exemple suivant, la `identifier` valeur DynamoDB String est affectée au `id` champ dans `key` une section (peut-être dans `GetItem` un document de mappage de demandes).

```
"key" : {
   "id" : { "S" : "identifier" }
}
```

Types pris en charge

AWS AppSync prend en charge les types de scalaire, de document et d'ensemble DynamoDB suivants :

Type de chaîne **S**

Valeur de chaîne unique. Une valeur de chaîne DynamoDB est indiquée par :

```
{ "S" : "some string" }
```

Voici un exemple d'utilisation :

```
"key" : {
   "id" : { "S" : "some string" }
}
```

Type d'ensemble de chaîne **SS**

Ensemble de valeurs de chaîne. La valeur d'un ensemble de chaînes DynamoDB est indiquée par :

```
{ "SS" : [ "first value", "second value", ... ] }
```

Voici un exemple d'utilisation :

```
"attributeValues" : {
   "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
}
```

Type de nombre **N**

Valeur numérique unique. La valeur d'un numéro DynamoDB est indiquée par :

```
{ "N" : 1234 }
```

Voici un exemple d'utilisation :

```
"expressionValues" : {
   ":expectedVersion" : { "N" : 1 }
}
```

Type d'ensemble de nombres **NS**

Ensemble de valeurs de nombres. La valeur d'un ensemble de numéros DynamoDB est indiquée par :

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

Voici un exemple d'utilisation :

```
"attributeValues" : {
   "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }
}
```

Type binaire **B**

Valeur binaire. Une valeur binaire DynamoDB est désignée par :

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

Notez que la valeur est en fait une chaîne, où la chaîne est la représentation codée en base64 des données binaires. AWS AppSync décode cette chaîne dans sa valeur binaire avant de l'envoyer à DynamoDB. AWS AppSync utilise le schéma de décodage base64 tel que défini par la RFC 2045 : tout caractère qui n'est pas dans l'alphabet base64 est ignoré.

Voici un exemple d'utilisation :

```
"attributeValues" : {
   "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

Type d'ensemble binaire **BS**

Ensemble de valeurs binaires. La valeur d'un ensemble binaire DynamoDB est désignée par :

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

Notez que la valeur est en fait une chaîne, où la chaîne est la représentation codée en base64 des données binaires. AWS AppSync décode cette chaîne dans sa valeur binaire avant de l'envoyer à DynamoDB. AWS AppSync utilise le schéma de décodage base64 tel que défini par la RFC 2045 : tout caractère qui n'est pas dans l'alphabet base64 est ignoré.

Voici un exemple d'utilisation :

```
"attributeValues" : {
   "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }
}
```

Type booléen **BOOL**

Valeur booléenne. Une valeur booléenne DynamoDB est désignée par :

```
{ "BOOL" : true }
```

Notez que seules les valeurs `true` et `false` sont valides.

Voici un exemple d'utilisation :

```
"attributeValues" : {
   "orderComplete" : { "BOOL" : false }
}
```

Type de liste **L**

Liste de toutes les autres valeurs DynamoDB prises en charge. Une valeur de liste DynamoDB est indiquée par :

```
{ "L" : [ ... ] }
```

Notez que la valeur est une valeur composée, la liste pouvant contenir zéro ou plus de toute valeur DynamoDB prise en charge (y compris les autres listes). La liste peut également contenir une combinaison de différents types.

Voici un exemple d'utilisation :

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

Type de mappage **M**

Représentant une collection non ordonnée de paires clé-valeur d'autres valeurs DynamoDB prises en charge. La valeur d'une carte DynamoDB est indiquée par :

```
{ "M" : { ... } }
```

Notez qu'un mappage peut contenir zéro ou plusieurs paires clé-valeur. La clé doit être une chaîne, et la valeur peut être n'importe quelle valeur DynamoDB prise en charge (y compris d'autres cartes). Le mappage peut également contenir une combinaison de différents types.

Voici un exemple d'utilisation :

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

Type nul **NULL**

Valeur null. Une valeur DynamoDB Null est désignée par :

```
{ "NULL" : null }
```

Voici un exemple d'utilisation :

```
"attributeValues" : {
   "phoneNumbers" : { "NULL" : null }
}
```

Pour plus d'informations sur chaque type, consultez la documentation DynamoDB.

Système de types (mappage des réponses)

Lorsque vous recevez une réponse de DynamoDB AWS AppSync , elle est automatiquement convertie en types primitifs GraphQL et JSON. Chaque attribut de DynamoDB est décodé et renvoyé dans le contexte de mappage des réponses.

Par exemple, si DynamoDB renvoie ce qui suit :

```
{
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

Le résolveur AWS AppSync DynamoDB le convertit ensuite en types GraphQL et JSON sous la forme suivante :

```
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

Cette section explique comment AWS AppSync convertit les types scalaires, de document et d'ensemble DynamoDB suivants :

Type de chaîne **S**

Valeur de chaîne unique. Une valeur DynamoDB String est renvoyée sous forme de chaîne.

Par exemple, si DynamoDB a renvoyé la valeur de chaîne DynamoDB suivante :

```
{ "S" : "some string" }
```

AWS AppSync le convertit en chaîne :

```
"some string"
```

Type d'ensemble de chaîne **SS**

Ensemble de valeurs de chaîne. Une valeur d'ensemble de chaînes DynamoDB est renvoyée sous forme de liste de chaînes.

Par exemple, si DynamoDB a renvoyé la valeur DynamoDB String Set suivante :

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync le convertit en une liste de chaînes :

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

Type de nombre **N**

Valeur numérique unique. La valeur d'un numéro DynamoDB est renvoyée sous forme de nombre.

Par exemple, si DynamoDB a renvoyé la valeur du numéro DynamoDB suivante :

```
{ "N" : 1234 }
```

AWS AppSync le convertit en nombre :

```
1234
```

Type d'ensemble de nombres **NS**

Ensemble de valeurs de nombres. Une valeur d'ensemble de numéros DynamoDB est renvoyée sous forme de liste de nombres.

Par exemple, si DynamoDB a renvoyé la valeur DynamoDB Number Set suivante :

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync le convertit en une liste de nombres :

```
[ 67.8, 12.2, 70 ]
```

Type binaire **B**

Valeur binaire. Une valeur binaire DynamoDB est renvoyée sous forme de chaîne contenant la représentation base64 de cette valeur.

Par exemple, si DynamoDB a renvoyé la valeur binaire DynamoDB suivante :

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync le convertit en une chaîne contenant la représentation base64 de la valeur :

```
"SGVsbG8sIFdvcmxkIQo="
```

Notez que les données binaires sont codées selon le schéma Base64 tel que spécifié dans RFC 4648 et RFC 2045.

Type d'ensemble binaire **BS**

Ensemble de valeurs binaires. Une valeur d'ensemble binaire DynamoDB est renvoyée sous la forme d'une liste de chaînes contenant la représentation base64 des valeurs.

Par exemple, si DynamoDB a renvoyé la valeur d'ensemble binaire DynamoDB suivante :

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync le convertit en une liste de chaînes contenant la représentation en base64 des valeurs :

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

Notez que les données binaires sont codées selon le schéma Base64 tel que spécifié dans RFC 4648 et RFC 2045.

Type booléen **BOOL**

Valeur booléenne. Une valeur booléenne DynamoDB est renvoyée sous forme de booléen.

Par exemple, si DynamoDB a renvoyé la valeur booléenne DynamoDB suivante :

```
{ "BOOL" : true }
```

AWS AppSync le convertit en booléen :

```
true
```

Type de liste **L**

Liste de toutes les autres valeurs DynamoDB prises en charge. Une valeur de liste DynamoDB est renvoyée sous forme de liste de valeurs, chaque valeur interne étant également convertie.

Par exemple, si DynamoDB a renvoyé la valeur de liste DynamoDB suivante :

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

AWS AppSync le convertit en une liste de valeurs converties :

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

Type de mappage **M**

Collection clé/valeur de toute autre valeur DynamoDB prise en charge. Une valeur de carte DynamoDB est renvoyée sous forme d'objet JSON, où chaque clé/valeur est également convertie.

Par exemple, si DynamoDB a renvoyé la valeur de carte DynamoDB suivante :

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

AWS AppSync le convertit en objet JSON :

```
{
   "someString" : "A string value",
   "someNumber" : 1,
```

```
   "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

Type nul **`NULL`**

Valeur null.

Par exemple, si DynamoDB a renvoyé la valeur Null DynamoDB suivante :

```
{ "NULL" : null }
```

AWS AppSync le convertit en valeur nulle :

```
null
```

Filtres

Lorsque vous interrogez des objets dans DynamoDB à `Query` l'aide des opérations `Scan` et, vous pouvez éventuellement spécifier `filter` un qui évalue les résultats et renvoie uniquement les valeurs souhaitées.

La section de mappage de filtre d'un document de mappage `Query` ou `Scan` a la structure suivante :

```
"filter" : {
    "expression" : "filter expression"
    "expressionNames" : {
        "#name" : "name",
    },
    "expressionValues" : {
        ":value" : ... typed value
    },
}
```

Les champs sont définis comme suit :

`expression`

Expression de la requête. Pour plus d'informations sur la façon d'écrire des expressions de filtre, consultez la documentation DynamoDB et QueryFilter ScanFilter DynamoDB. Ce champ doit être spécifié.

`expressionNames`

Substituts des espaces réservés de nom des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé de nom utilisé dans le `expression`. La valeur doit être une chaîne correspondant au nom d'attribut de l'élément dans DynamoDB. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de nom des attributs de l'expression utilisés dans l'`expression`.

`expressionValues`

Substituts des espaces réservés de valeur des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé de valeur utilisé dans l'`expression`, et la valeur doit être typée. Pour de plus amples informations sur la spécification d'une « valeur typée », veuillez consulter Système de types (mappage des requêtes). Cela doit être spécifié. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de valeur des attributs de l'expression utilisés dans l'`expression`.

Exemple

L'exemple suivant est une section de filtre pour un modèle de mappage, dans laquelle les entrées extraites de DynamoDB ne sont renvoyées que si le titre commence par l'argument. `title`

```
"filter" : {
    "expression" : "begins_with(#title, :title)",
    "expressionNames" : {
        "#title" : "title"
    },
    "expressionValues" : {
        ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
    }
}
```

Expressions de condition

Lorsque vous mutez des objets dans DynamoDB à l'aide des opérations`PutItem`,`UpdateItem`, et `DeleteItem` DynamoDB, vous pouvez éventuellement spécifier une expression de condition qui détermine si la demande doit aboutir ou non, en fonction de l'état de l'objet déjà présent dans DynamoDB avant l'exécution de l'opération.

Le résolveur AWS AppSync DynamoDB permet de spécifier une expression de condition `PutItem` dans `DeleteItem` et de demander des documents de mappage`UpdateItem`, ainsi qu'une stratégie à suivre si la condition échoue et que l'objet n'a pas été mis à jour.

Exemple 1

Le document de mappage `PutItem` suivant ne dispose pas d'une expression de condition. Par conséquent, il place un élément dans DynamoDB même s'il existe déjà un élément portant la même clé, remplaçant ainsi l'élément existant.

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   }
}
```

Exemple 2

Le document de `PutItem` mappage suivant contient une expression de condition qui permet à l'opération de réussir uniquement si un élément portant la même clé n'existe pas dans DynamoDB.

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   },
   "condition" : {
      "expression" : "attribute_not_exists(id)"
   }
}
```

Par défaut, si le contrôle de condition échoue, le résolveur AWS AppSync DynamoDB renvoie une erreur concernant la mutation et la valeur actuelle de l'objet dans DynamoDB dans un `data` champ de la section de la réponse GraphQL. `error` Cependant, le résolveur AWS AppSync DynamoDB propose des fonctionnalités supplémentaires pour aider les développeurs à gérer certains cas extrêmes courants :

- Si le résolveur AWS AppSync DynamoDB peut déterminer que la valeur actuelle dans DynamoDB correspond au résultat souhaité, il traite l'opération comme si elle avait réussi de toute façon.
- Au lieu de renvoyer une erreur, vous pouvez configurer le résolveur pour qu'il appelle une fonction Lambda personnalisée afin de décider de la manière dont le résolveur AWS AppSync DynamoDB doit gérer la panne.

Cette opération est décrite plus en détail dans la section Gestion d'un échec de contrôle de condition.

Pour plus d'informations sur les expressions de conditions DynamoDB, consultez la documentation DynamoDB. ConditionExpressions

Spécifier une condition

Les documents de mappage des demandes `PutItem`, `UpdateItem` et `DeleteItem` permettent tous la spécification d'une section de `condition` facultative. Si cette section est omise, aucune vérification de condition n'est effectuée. Si elle est spécifiée, la condition doit être true pour que l'opération réussisse.

Une section `condition` a la structure suivante :

```
"condition" : {
    "expression" : "someExpression"
    "expressionNames" : {
        "#foo" : "foo"
    },
    "expressionValues" : {
        ":bar" : ... typed value
    },
    "equalsIgnore" : [ "version" ],
    "consistentRead" : true,
    "conditionalCheckFailedHandler" : {
        "strategy" : "Custom",
        "lambdaArn" : "arn:..."
    }
}
```

Les champs suivants spécifient la condition :

`expression`

Expression de mise à jour elle-même. Pour plus d'informations sur la façon d'écrire des expressions de condition, consultez la documentation ConditionExpressions DynamoDB. Ce champ doit être spécifié.

`expressionNames`

Substituts des espaces réservés de nom des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé utilisé dans l'expression, et la valeur doit être une chaîne correspondant au nom d'attribut de l'élément dans DynamoDB. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de nom des attributs de l'expression utilisés dans l'expression.

`expressionValues`

Substituts des espaces réservés de valeur des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé de valeur utilisé dans l'expression, et la valeur doit être typée. Pour de plus amples informations sur la spécification d'une « valeur typée », veuillez consulter Système de types (mappage des requêtes). Cela doit être spécifié. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de valeur des attributs de l'expression utilisés dans l'expression.

Les champs restants indiquent au résolveur AWS AppSync DynamoDB comment gérer un échec de vérification de condition :

`equalsIgnore`

Lorsqu'une vérification de condition échoue lors de l'utilisation de l'`PutItem`opération, le résolveur AWS AppSync DynamoDB compare l'élément actuellement dans DynamoDB à l'élément qu'il a essayé d'écrire. S'ils sont identiques, il traite l'opération comme si elle avait réussi. Vous pouvez utiliser le champ `equalsIgnore` afin de spécifier une liste d'attributs qu'AWS AppSync doit ignorer lors de l'exécution de cette comparaison. Par exemple, si la seule différence était un `version` attribut, il traite l'opération comme si elle avait réussi. Ce champ est facultatif.

`consistentRead`

Lorsqu'une vérification de condition échoue, AWS AppSync obtient la valeur actuelle de l'élément auprès de DynamoDB à l'aide d'une lecture hautement cohérente. Vous pouvez utiliser ce champ pour indiquer au résolveur AWS AppSync DynamoDB d'utiliser une lecture éventuellement cohérente à la place. Ce champ est facultatif et contient `true` par défaut.

`conditionalCheckFailedHandler`

Cette section vous permet de spécifier comment le résolveur AWS AppSync DynamoDB traite un échec de vérification de condition après avoir comparé la valeur actuelle dans DynamoDB au résultat attendu. Cette section est facultative. Si elle n'est pas spécifiée, la valeur par défaut est une stratégie `Reject`.

`strategy`

Stratégie adoptée par le résolveur AWS AppSync DynamoDB après avoir comparé la valeur actuelle dans DynamoDB au résultat attendu. Ce champ est obligatoire et les valeurs suivantes sont possibles :

`Reject`

La mutation échoue et une erreur se produit concernant la mutation et la valeur actuelle de l'objet dans DynamoDB dans `data` un champ de la section de `error` la réponse GraphQL.

`Custom`

Le résolveur AWS AppSync DynamoDB invoque une fonction Lambda personnalisée pour décider de la manière de gérer l'échec du contrôle de condition. Lorsque la `strategy` est définie sur `Custom`, le champ `lambdaArn` doit contenir l'ARN de la fonction Lambda à appeler.

`lambdaArn`

L'ARN de la fonction Lambda à invoquer qui détermine la manière dont le résolveur DynamoDB doit gérer l'AWS AppSync échec de la vérification des conditions. Ce champ doit être spécifié uniquement lorsque `strategy` est défini sur `Custom`. Pour plus d'informations sur l'utilisation de cette fonction, consultez Gestion de l'échec d'une vérification de condition.

Gestion d'un échec de vérification d'état

Par défaut, lorsqu'une vérification de condition échoue, le résolveur AWS AppSync DynamoDB renvoie une erreur concernant la mutation et la valeur actuelle de l'objet dans DynamoDB dans un `data` champ de la section de la réponse GraphQL. `error` Cependant, le résolveur AWS AppSync DynamoDB propose des fonctionnalités supplémentaires pour aider les développeurs à gérer certains cas extrêmes courants :

- Si le résolveur AWS AppSync DynamoDB peut déterminer que la valeur actuelle dans DynamoDB correspond au résultat souhaité, il traite l'opération comme si elle avait réussi de toute façon.

- Au lieu de renvoyer une erreur, vous pouvez configurer le résolveur pour qu'il appelle une fonction Lambda personnalisée afin de décider de la manière dont le résolveur AWS AppSync DynamoDB doit gérer la panne.

Le diagramme de ce processus est le suivant :

Vérification du résultat souhaité

Lorsque le contrôle de condition échoue, le résolveur AWS AppSync DynamoDB exécute `GetItem` une requête DynamoDB pour obtenir la valeur actuelle de l'élément auprès de DynamoDB. Par défaut, il utilise une lecture à cohérence forte, mais cela peut être configuré à l'aide du champ `consistentRead` dans le bloc `condition` et comparé aux résultats prévus :

- Pour l'`PutItem`opération, le résolveur AWS AppSync DynamoDB compare la valeur actuelle à celle qu'il a tenté d'écrire, en excluant les attributs répertoriés dans `equalsIgnore` la comparaison. Si les éléments sont identiques, il considère l'opération comme réussie et renvoie l'élément extrait de DynamoDB. Dans le cas contraire, il suit la stratégie configurée.

  Par exemple, si le document de mappage des requêtes `PutItem` ressemble à ce qui suit :

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   },
   "attributeValues" : {
      "name" : { "S" : "Steve" },
      "version" : { "N" : 2 }
   },
   "condition" : {
      "expression" : "version = :expectedVersion",
      "expressionValues" : {
          ":expectedVersion" : { "N" : 1 }
      },
      "equalsIgnore": [ "version" ]
   }
}
```

  Et si l'élément actuellement dans DynamoDB ressemble à ce qui suit :

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

Le résolveur AWS AppSync DynamoDB comparait l'élément qu'il a essayé d'écrire à la valeur actuelle, constatant que la seule différence réside dans `version` le champ, mais comme il est configuré pour ignorer `version` le champ, il considère l'opération comme réussie et renvoie l'élément extrait de DynamoDB.

- Pour l'`DeleteItem`opération, le résolveur AWS AppSync DynamoDB vérifie qu'un élément a été renvoyé par DynamoDB. Si aucun élément n'a été renvoyé, il traite l'opération comme réussie. Dans le cas contraire, il suit la stratégie configurée.
- Pour l'`UpdateItem`opération, le résolveur AWS AppSync DynamoDB ne dispose pas de suffisamment d'informations pour déterminer si l'élément actuellement dans DynamoDB correspond au résultat attendu et suit donc la stratégie configurée.

Si l'état actuel de l'objet dans DynamoDB est différent du résultat attendu, le résolveur AWS AppSync DynamoDB suit la stratégie configurée, soit pour rejeter la mutation, soit pour invoquer une fonction Lambda pour déterminer la marche à suivre.

Suivre la stratégie du « rejet »

Lorsque vous suivez `Reject` cette stratégie, le résolveur AWS AppSync DynamoDB renvoie une erreur pour la mutation, et la valeur actuelle de l'objet dans DynamoDB est également renvoyée dans un `data` champ de la section de la réponse GraphQL. `error` L'élément renvoyé par DynamoDB est soumis au modèle de mappage des réponses pour le traduire dans le format attendu par le client, puis il est filtré par le jeu de sélection.

Par exemple, si nous avons la demande de mutation suivante :

```
mutation {
    updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
        Name
        theVersion
    }
}
```

Si l'élément renvoyé par DynamoDB ressemble à ce qui suit :

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

Et si le modèle de mappage des réponses ressemble à ce qui suit :

```
{
   "id" : $util.toJson($context.result.id),
   "Name" : $util.toJson($context.result.name),
   "theVersion" : $util.toJson($context.result.version)
}
```

La réponse GraphQL se présente comme suit :

```
{
  "data": null,
  "errors": [
    {
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "data": {
        "Name": "Steve",
        "theVersion": 8
      },
      ...
    }
  ]
}
```

Notez également que si des champs de l'objet renvoyé sont remplis par d'autres résolveurs et que la mutation a réussi, ils ne seront pas résolus lorsque l'objet sera renvoyé dans la section `error`.

Suivre la stratégie « personnalisée »

Lorsqu'il suit la `Custom` stratégie, le résolveur AWS AppSync DynamoDB invoque une fonction Lambda pour décider de la marche à suivre. La fonction Lambda choisit l'une des options suivantes :

- `reject` la mutation. Cela indique au résolveur AWS AppSync DynamoDB de se comporter comme si la stratégie configurée l'`Reject`était, renvoyant une erreur pour la mutation et la valeur actuelle de l'objet dans DynamoDB, comme décrit dans la section précédente.
- `discard` la mutation. Cela indique au résolveur AWS AppSync DynamoDB d'ignorer silencieusement l'échec de la vérification des conditions et renvoie la valeur dans DynamoDB.
- `retry` la mutation. Cela indique au résolveur AWS AppSync DynamoDB de réessayer la mutation avec un nouveau document de mappage de requêtes.

Requête d'appel Lambda

Le résolveur AWS AppSync DynamoDB invoque la fonction Lambda spécifiée dans le. `lambdaArn` Il utilise le même `service-role-arn` que celui configuré sur la source de données. La charge utile de l'appel a la structure suivante :

```
{
    "arguments": { ... },
    "requestMapping": {... },
    "currentValue": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

Les champs sont définis comme suit :

`arguments`

Arguments de la mutation GraphQL. Ce sont les mêmes arguments que ceux disponibles pour le document de mappage des demandes dans `$context.arguments`.

`requestMapping`

Document de mappage des demandes pour cette opération.

`currentValue`

Valeur actuelle de l'objet dans DynamoDB.

`resolver`

Informations sur le résolveur AWS AppSync .

`identity`

Informations sur l'appelant. Ce sont les mêmes informations que celles disponibles pour le document de mappage des demandes dans `$context.identity`.

Exemple complet de la charge utile :

```
{
    "arguments": {
        "id": "1",
        "name": "Steve",
        "expectedVersion": 1
    },
    "requestMapping": {
        "version" : "2017-02-28",
        "operation" : "PutItem",
        "key" : {
           "id" : { "S" : "1" }
        },
        "attributeValues" : {
           "name" : { "S" : "Steve" },
           "version" : { "N" : 2 }
        },
        "condition" : {
           "expression" : "version = :expectedVersion",
           "expressionValues" : {
               ":expectedVersion" : { "N" : 1 }
           },
           "equalsIgnore": [ "version" ]
        }
    },
    "currentValue": {
        "id" : { "S" : "1" },
        "name" : { "S" : "Steve" },
        "version" : { "N" : 8 }
    },
    "resolver": {
        "tableName": "People",
        "awsRegion": "us-west-2",
        "parentType": "Mutation",
        "field": "updatePerson",
        "outputType": "Person"
    },
```

```
    "identity": {
        "accountId": "123456789012",
        "sourceIp": "x.x.x.x",
        "user": "AIDAAAAAAAAAAAAAAAAA",
        "userArn": "arn:aws:iam::123456789012:user/appsync"
    }
}
```

Réponse à l'appel de Lambda

La fonction Lambda peut inspecter la charge utile d'appel et appliquer n'importe quelle logique métier pour décider de la manière dont le résolveur DynamoDB doit gérer AWS AppSync la panne. Il existe trois options pour gérer l'échec de vérification de la condition :

- `reject` la mutation. La charge utile de la réponse pour cette option doit avoir cette structure :

  ```
  {
      "action": "reject"
  }
  ```

  Cela indique au résolveur AWS AppSync DynamoDB de se comporter comme si la stratégie configurée l'`Reject`était, renvoyant une erreur pour la mutation et la valeur actuelle de l'objet dans DynamoDB, comme décrit dans la section ci-dessus.
- `discard` la mutation. La charge utile de la réponse pour cette option doit avoir cette structure :

  ```
  {
      "action": "discard"
  }
  ```

  Cela indique au résolveur AWS AppSync DynamoDB d'ignorer silencieusement l'échec de la vérification des conditions et renvoie la valeur dans DynamoDB.
- `retry` la mutation. La charge utile de la réponse pour cette option doit avoir cette structure :

  ```
  {
      "action": "retry",
      "retryMapping": { ... }
  }
  ```

Cela indique au résolveur AWS AppSync DynamoDB de réessayer la mutation avec un nouveau document de mappage de requêtes. La structure de `retryMapping` cette section dépend de l'opération DynamoDB et constitue un sous-ensemble du document complet de mappage des demandes pour cette opération.

Pour `PutItem`, la section `retryMapping` a la structure suivante. Pour une description du `attributeValues` champ, voir PutItem.

```
{
    "attributeValues": { ... },
    "condition": {
        "equalsIgnore" = [ ... ],
        "consistentRead" = true
    }
}
```

Pour `UpdateItem`, la section `retryMapping` a la structure suivante. Pour une description de `update` cette section, voir UpdateItem.

```
{
    "update" : {
        "expression" : "someExpression"
        "expressionNames" : {
            "#foo" : "foo"
        },
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "condition": {
        "consistentRead" = true
    }
}
```

Pour `DeleteItem`, la section `retryMapping` a la structure suivante.

```
{
    "condition": {
        "consistentRead" = true
    }
```

```
}
```

Il n'y a aucun moyen de spécifier une autre opération ou une autre clé sur laquelle travailler. Le résolveur AWS AppSync DynamoDB autorise uniquement les tentatives de la même opération sur le même objet. D'autre part, la section `condition` ne permet pas de spécifier un `conditionalCheckFailedHandler`. Si la nouvelle tentative échoue, le résolveur AWS AppSync DynamoDB suit la stratégie. `Reject`

Voici un exemple de fonction Lambda pour traiter une demande `PutItem` qui a échoué. La logique métier s'adresse à celui qui effectue l'appel. S'il a été créé par`jeffTheAdmin`, il réessaie la demande en mettant à jour le `version` et à `expectedVersion` partir de l'élément actuellement présent dans DynamoDB. Dans le cas contraire, il rejette la mutation.

```
exports.handler = (event, context, callback) => {
    console.log("Event: "+ JSON.stringify(event));

    // Business logic goes here.

    var response;
    if ( event.identity.user == "jeffTheAdmin" ) {
        response = {
            "action" : "retry",
            "retryMapping" : {
                "attributeValues" : event.requestMapping.attributeValues,
                "condition" : {
                    "expression" : event.requestMapping.condition.expression,
                    "expressionValues" :
 event.requestMapping.condition.expressionValues
                }
            }
        }
        response.retryMapping.attributeValues.version = { "N" :
 event.currentValue.version.N + 1 }
        response.retryMapping.condition.expressionValues[':expectedVersion'] =
 event.currentValue.version

    } else {
        response = { "action" : "reject" }
    }

    console.log("Response: "+ JSON.stringify(response))
```

```
    callback(null, response)
};
```

Expressions des conditions de transaction

Les expressions de condition de transaction sont disponibles dans les modèles de mappage de requêtes des quatre types d'opérations dans `TransactWriteItems`, à savoir `PutItem`, `DeleteItem`, `UpdateItem` et `ConditionCheck`.

Pour`PutItem`, et `DeleteItem``UpdateItem`, l'expression de condition de transaction est facultative. En effet`ConditionCheck`, l'expression de la condition de transaction est obligatoire.

Exemple 1

Le document de mappage transactionnel `DeleteItem` suivant n'a pas d'expression de condition. Par conséquent, il supprime l'élément dans DynamoDB.

```
{
   "version": "2018-05-29",
   "operation": "TransactWriteItems",
   "transactItems": [
      {
         "table": "posts",
         "operation": "DeleteItem",
         "key": {
            "id": { "S" : "1" }
         }
      }
   ]
}
```

Exemple 2

Le document de `DeleteItem` mappage transactionnel suivant contient une expression de condition de transaction qui permet à l'opération de réussir uniquement si l'auteur de cette publication porte un certain nom.

```
{
   "version": "2018-05-29",
   "operation": "TransactWriteItems",
   "transactItems": [
      {
```

```
         "table": "posts",
         "operation": "DeleteItem",
         "key": {
            "id": { "S" : "1" }
         }
         "condition": {
            "expression": "author = :author",
            "expressionValues": {
               ":author": { "S" : "Chunyan" }
            }
         }
      }
   ]
}
```

Si la vérification de condition échoue, elle provoque `TransactionCanceledException` et les détails de l'erreur sont renvoyés dans `$ctx.result.cancellationReasons`. Notez que par défaut, l'ancien élément de DynamoDB à l'origine de l'échec de la vérification des conditions sera renvoyé. `$ctx.result.cancellationReasons`

Spécifier une condition

Les documents de mappage des demandes `PutItem`, `UpdateItem` et `DeleteItem` permettent tous la spécification d'une section de `condition` facultative. Si cette section est omise, aucune vérification de condition n'est effectuée. Si elle est spécifiée, la condition doit être true pour que l'opération réussisse. Le `ConditionCheck` doit avoir une section `condition` à spécifier. La condition doit être vraie pour que l'ensemble de la transaction réussisse.

Une section `condition` a la structure suivante :

```
"condition": {
    "expression": "someExpression",
    "expressionNames": {
        "#foo": "foo"
    },
    "expressionValues": {
        ":bar": ... typed value
    },
    "returnValuesOnConditionCheckFailure": false
}
```

Les champs suivants spécifient la condition :

`expression`

Expression de mise à jour elle-même. Pour plus d'informations sur la façon d'écrire des expressions de condition, consultez la documentation ConditionExpressions DynamoDB. Ce champ doit être spécifié.

`expressionNames`

Substituts des espaces réservés de nom des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé utilisé dans l'expression, et la valeur doit être une chaîne correspondant au nom d'attribut de l'élément dans DynamoDB. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de nom des attributs de l'expression utilisés dans l'expression.

`expressionValues`

Substituts des espaces réservés de valeur des attributs de l'expression, sous la forme de paires clé-valeur. La clé correspond à un espace réservé de valeur utilisé dans l'expression, et la valeur doit être typée. Pour de plus amples informations sur la spécification d'une « valeur typée », veuillez consulter Système de types (mappage des requêtes). Cela doit être spécifié. Ce champ est facultatif et doit être renseigné uniquement avec des substituts des espaces réservés de valeur des attributs de l'expression utilisés dans l'expression.

`returnValuesOnConditionCheckFailure`

Spécifiez s'il faut récupérer l'élément dans DynamoDB en cas d'échec d'une vérification de condition. L'élément récupéré est dans `$ctx.result.cancellationReasons[$index].item`, où `$index` est l'index de l'élément de demande qui a échoué à la vérification de condition. Cette valeur est définie par défaut sur true.

Projections

Lorsque vous lisez des objets dans DynamoDB à `GetItem` l'aide des opérations `ScanQuery`,`BatchGetItem`,, `TransactGetItems` et, vous pouvez éventuellement spécifier une projection qui identifie les attributs souhaités. La structure de la projection est la suivante, similaire à celle des filtres :

```
"projection" : {
    "expression" : "projection expression"
    "expressionNames" : {
```

```
        "#name" : "name",
    }
}
```

Les champs sont définis comme suit :

`expression`

L'expression de projection, qui est une chaîne. Pour récupérer un seul attribut, spécifiez son nom. Pour les attributs multiples, les noms doivent être des valeurs séparées par des virgules. Pour plus d'informations sur l'écriture d'expressions de projection, consultez la documentation sur les expressions de projection DynamoDB. Ce champ est obligatoire.

`expressionNames`

Les substitutions aux espaces réservés aux noms d'attributs d'expression sous forme de paires clé-valeur. La clé correspond à un espace réservé de nom utilisé dans le `expression`. La valeur doit être une chaîne correspondant au nom d'attribut de l'élément dans DynamoDB. Ce champ est facultatif et ne doit être rempli qu'avec des substitutions pour les espaces réservés aux noms d'attributs d'expression utilisés dans le. `expression` Pour plus d'informations`expressionNames`, consultez la documentation DynamoDB.

Exemple 1

L'exemple suivant est une section de projection pour un modèle de mappage VTL dans lequel seuls les attributs `author` et B `id` sont renvoyés par DynamoDB :

```
"projection" : {
    "expression" : "#author, id",
    "expressionNames" : {
        "#author" : "author"
    }
}
```

> **Tip**
>
> Vous pouvez accéder à votre ensemble de sélection de requêtes GraphQL à l'aide de $context.info. selectionSetList. Ce champ vous permet de cadrer votre expression de projection de manière dynamique en fonction de vos besoins.

> **Note**
>
> Lorsque vous utilisez des expressions de projection avec les `Scan` opérations `Query` et, la valeur de `select` doit êtreSPECIFIC_ATTRIBUTES. Pour plus d'informations, consultez la documentation DynamoDB.

Référence du modèle de mappage du résolveur pour RDS

Les modèles de mappage du résolveur AWS AppSync RDS permettent aux développeurs d'envoyer des requêtes SQL à une API de données pour Amazon Aurora Serverless et de récupérer le résultat de ces requêtes.

Modèle de mappage des demandes

Le modèle de mappage de demande RDS est relativement simple :

```
{
    "version": "2018-05-29",
    "statements": [],
    "variableMap": {},
    "variableTypeHintMap": {}
}
```

Voici la représentation du schéma JSON du modèle de mappage de demande RDS, une fois qu'il est résolu :

```
{
    "definitions": {},
    "$schema": "https://json-schema.org/draft-07/schema#",
    "$id": "https://example.com/root.json",
    "type": "object",
    "title": "The Root Schema",
    "required": [
        "version",
        "statements",
        "variableMap"
    ],
    "properties": {
        "version": {
```

```
            "$id": "#/properties/version",
            "type": "string",
            "title": "The Version Schema",
            "default": "",
            "examples": [
                "2018-05-29"
            ],
            "enum": [
                "2018-05-29"
            ],
            "pattern": "^(.*)$"
        },
        "statements": {
            "$id": "#/properties/statements",
            "type": "array",
            "title": "The Statements Schema",
            "items": {
                "$id": "#/properties/statements/items",
                "type": "string",
                "title": "The Items Schema",
                "default": "",
                "examples": [
                    "SELECT * from BOOKS"
                ],
                "pattern": "^(.*)$"
            }
        },
        "variableMap": {
            "$id": "#/properties/variableMap",
            "type": "object",
            "title": "The Variablemap Schema"
        },
        "variableTypeHintMap": {
            "$id": "#/properties/variableTypeHintMap",
            "type": "object",
            "title": "The variableTypeHintMap Schema"
        }
    }
}
```

Voici un exemple de modèle de mappage de demandes avec une requête statique :

```
{
```

```
    "version": "2018-05-29",
    "statements": [
        "select title, isbn13 from BOOKS where author = 'Mark Twain'"
    ]
}
```

Version

Commun à tous les modèles de mappage de demandes, le champ de version définit la version utilisée par le modèle. Le champ de version est obligatoire. La valeur « 2018-05-29 » est la seule version prise en charge pour les modèles de mappage Amazon RDS.

```
"version": "2018-05-29"
```

Déclarations et VariableMap

Le tableau des instructions est un espace réservé pour les requêtes fournies par le développeur. Actuellement, jusqu'à deux requêtes par modèle de mappage de demandes sont prises en charge. `variableMap`Il s'agit d'un champ facultatif qui contient des alias qui peuvent être utilisés pour raccourcir les instructions SQL et les rendre plus lisibles. Par exemple, les solutions suivantes sont possibles :

```
{
"version": "2018-05-29",
    "statements": [
        "insert into BOOKS VALUES (:AUTHOR, :TITLE, :ISBN13)",
        "select * from BOOKS WHERE isbn13 = :ISBN13"
    ],
    "variableMap": {
        ":AUTHOR": $util.toJson($ctx.args.newBook.author),
        ":TITLE": $util.toJson($ctx.args.newBook.title),
        ":ISBN13": $util.toJson($ctx.args.newBook.isbn13)
    }
}
```

AWS AppSyncutilisera les valeurs cartographiques variables pour créer les SqlParameterizedrequêtes qui seront envoyées à l'API de données sans serveur Amazon Aurora. Les instructions SQL sont exécutées avec les paramètres fournis dans la carte des variables, ce qui élimine le risque d'injection de code SQL.

VariableTypeHintMap

`variableTypeHintMap`Il s'agit d'un champ facultatif contenant des types aliasés qui peuvent être utilisés pour envoyer des indications sur le type de paramètre SQL. Ces indications de type évitent le transtypage explicite des instructions SQL, ce qui les raccourcit. Par exemple, les solutions suivantes sont possibles :

```
{
    "version": "2018-05-29",
    "statements": [
        "insert into LOGINDATA VALUES (:ID, :TIME)",
        "select * from LOGINDATA WHERE id = :ID"
     ],
     "variableMap": {
        ":ID": $util.toJson($ctx.args.id),
        ":TIME": $util.toJson($ctx.args.time)
     },
     "variableTypeHintMap": {
        ":id": "UUID",
        ":time": "TIME"
     }
}
```

AWS AppSyncutilisera la valeur cartographique variable pour créer les requêtes envoyées à l'API Amazon Aurora Serverless Data. Il utilise également les `variableTypeHintMap` données et envoie les informations du type à RDS. Le support RDS `typeHints` peut être trouvé ici.

Référence du modèle de mappage Resolver pour OpenSearch

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Le AWS AppSync résolveur pour Amazon OpenSearch Service vous permet d'utiliser GraphQL pour stocker et récupérer des données dans les domaines de service OpenSearch existants de votre compte. Ce résolveur fonctionne en vous permettant de mapper une requête GraphQL entrante en OpenSearch une demande de service, puis de mapper la réponse OpenSearch du service à

GraphQL. Cette section décrit les modèles de mappage pour les opérations de OpenSearch service prises en charge.

Modèle de mappage de demande

La plupart des modèles de mappage des demandes de OpenSearch service ont une structure commune dans laquelle seuls quelques éléments changent. L'exemple suivant exécute une recherche dans un domaine OpenSearch de service, où les documents sont organisés sous un index appelé`post`. Les paramètres de recherche sont définis dans la section `body`, avec un grand nombre de clauses de requête courants définies dans le champ `query`. Cet exemple recherche des documents contenant `"Nadia"` ou `"Bailey"`, ou les deux, dans le champ `author` d'un document :

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/post/_search",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "from":0,
            "size":50,
            "query" : {
                "bool" : {
                    "should" : [
                        {"match" : { "author" : "Nadia" }},
                        {"match" : { "author" : "Bailey" }}
                    ]
                }
            }
        }
    }
}
```

Modèle de mappage de réponse

Comme pour les autres sources de données, OpenSearch Service envoie une réponse AWS AppSync qui doit être convertie en GraphQL.

La plupart des requêtes GraphQL recherchent le `_source` champ à partir d'une réponse de OpenSearch service. Comme vous pouvez effectuer des recherches pour renvoyer un document

individuel ou une liste de documents, deux modèles de mappage des réponses sont couramment utilisés dans OpenSearch Service :

Liste des résultats

```
[
    #foreach($entry in $context.result.hits.hits)
      #if( $velocityCount > 1 ) , #end
        $utils.toJson($entry.get("_source"))
    #end
]
```

Élément individuel

```
$utils.toJson($context.result.get("_source"))
```

operation field

(Modèle de mappage des DEMANDES uniquement)

Méthode ou verbe HTTP (GET, POST, PUT, HEAD ou DELETE) qui AWS AppSync envoie au domaine OpenSearch de service. La clé et la valeur doivent être une chaîne.

```
"operation" : "PUT"
```

path field

(Modèle de mappage des DEMANDES uniquement)

Le chemin de recherche d'une demande OpenSearch de service auprès de AWS AppSync. Cela forme une URL pour le verbe HTTP de l'opération. La clé et la valeur doivent être des chaînes.

```
"path" : "/<indexname>/_doc/<_id>"
"path" : "/<indexname>/_doc"
"path" : "/<indexname>/_search"
"path" : "/<indexname>/_update/<_id>
```

Lorsque le modèle de mappage est évalué, ce chemin est envoyé dans le cadre de la requête HTTP, y compris le domaine OpenSearch de service. Ainsi, l'exemple précédent pourrait donner :

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

`params` field

(Modèle de mappage des DEMANDES uniquement)

Permet de spécifier l'action exécutée par votre recherche, plus couramment en définissant la valeur de requête à l'intérieur du corps. Toutefois, il existe plusieurs autres capacités qui peuvent être configurées, comme la mise en forme des réponses.

- headers

  Informations d'en-tête, sous forme de paires clé-valeur. La clé et la valeur doivent être des chaînes. Par exemple :

  ```
  "headers" : {
      "Content-Type" : "application/json"
  }
  ```

  > **Note**
  >
  > AWS AppSync ne prend actuellement en charge que le format JSON en tant que fichier`Content-Type`.

- queryString

  Paires clé-valeur qui spécifient les options courantes, telles que la mise en forme du code pour les réponses JSON. La clé et la valeur doivent être une chaîne. Par exemple, si vous souhaitez obtenir un JSON mis en forme correctement, vous devez utiliser :

  ```
  "queryString" : {
      "pretty" : "true"
  }
  ```

- body

  Il s'agit de la partie principale de votre demande, qui permet AWS AppSync de créer une demande de recherche bien formée pour votre domaine OpenSearch de service. La clé doit être une chaîne composée d'un objet. Voici quelques démonstrations.

Exemple 1

Renvoyer tous les documents avec une ville correspondant à « seattle » :

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "match" : {
            "city" : "seattle"
        }
    }
}
```

Exemple 2

Renvoyer tous les documents correspondant à « washington » comme ville ou État :

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "multi_match" : {
            "query" : "washington",
            "fields" : ["city", "state"]
        }
    }
}
```

Transmission de variables

(Modèle de mappage des DEMANDES uniquement)

Vous pouvez également transmettre des variables dans le cadre d'une évaluation dans l'instruction VTL. Par exemple, supposons que vous ayez une requête GraphQL similaire à la suivante :

```
query {
    searchForState(state: "washington"){
        ...
    }
}
```

Le modèle de mappage pourrait avoir l'état comme argument :

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "multi_match" : {
            "query" : "$context.arguments.state",
            "fields" : ["city", "state"]
        }
    }
}
```

Pour obtenir la liste des utilitaires que vous pouvez inclure dans la VTL, consultez En-têtes de requête d'accès.

Référence du modèle de mappage Resolver pour Lambda

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Vous pouvez utiliser des AWS AppSync fonctions et des résolveurs pour appeler les fonctions Lambda situées dans votre compte. Vous pouvez façonner les charges utiles de vos demandes et la réponse de vos fonctions Lambda avant de les renvoyer à vos clients. Vous pouvez également utiliser des modèles de mappage pour donner des indications AWS AppSync sur la nature de l'opération à invoquer. Cette section décrit les différents modèles de mappage pour les opérations Lambda prises en charge.

Modèle de mappage des demandes

Le modèle de mappage de requêtes Lambda gère les champs liés à votre fonction Lambda :

```
{
  "version": string,
  "operation": Invoke|BatchInvoke,
  "payload": any type,
```

```
  "invocationType": RequestResponse|Event
}
```

Voici la représentation du schéma JSON du modèle de mappage de demandes Lambda une fois résolu :

```
{
  "definitions": {},
  "$schema": "https://json-schema.org/draft-06/schema#",
  "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
  "type": "object",
  "properties": {
    "version": {
      "$id": "/properties/version",
      "type": "string",
      "enum": [
        "2018-05-29"
      ],
      "title": "The Mapping template version.",
      "default": "2018-05-29"
    },
    "operation": {
      "$id": "/properties/operation",
      "type": "string",
      "enum": [
        "Invoke",
        "BatchInvoke"
      ],
      "title": "The Mapping template operation.",
      "description": "What operation to execute.",
      "default": "Invoke"
    },
    "payload": {},
    "invocationType": {
      "$id": "/properties/invocationType",
      "type": "string",
      "enum": [
        "RequestResponse",
        "Event"
      ],
      "title": "The Mapping template invocation type.",
      "description": "What invocation type to execute.",
      "default": "RequestResponse"
```

```
    }
  },
  "required": [
    "version",
    "operation"
  ],
  "additionalProperties": false
}
```

Voici un exemple qui utilise une `invoke` opération dont les données de charge utile sont le `getPost` champ d'un schéma GraphQL avec ses arguments issus du contexte :

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $util.toJson($context.arguments)
  }
}
```

L'intégralité du document de mappage est transmise en entrée à votre fonction Lambda, de sorte que l'exemple précédent ressemble désormais à ceci :

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "id": "postId1"
    }
  }
}
```

Version

Commun à tous les modèles de mappage de demandes, il `version` définit la version utilisée par le modèle. Le `version` est obligatoire et est une valeur statique :

```
"version": "2018-05-29"
```

Opération

La source de données Lambda vous permet de définir deux opérations `operation` sur le terrain : `Invoke` et. `BatchInvoke` L'`Invoke`opération permet de AWS AppSync savoir d'appeler votre fonction Lambda pour chaque résolveur de champs GraphQL. `BatchInvoke`indique de regrouper AWS AppSync les requêtes pour le champ GraphQL actuel. Le champ `operation` est obligatoire.

En effet`Invoke`, le modèle de mappage des demandes résolues correspond à la charge utile d'entrée de la fonction Lambda. Modifions l'exemple ci-dessus :

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
    "payload": {
      "arguments": $util.toJson($context.arguments)
    }
}
```

Ceci est résolu et transmis à la fonction Lambda, qui pourrait ressembler à ceci :

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
    "payload": {
      "arguments": {
        "id": "postId1"
      }
    }
}
```

En `BatchInvoke` effet, le modèle de mappage est appliqué à chaque résolveur de champs du lot. Par souci de concision, AWS AppSync fusionne toutes les `payload` valeurs du modèle de mappage résolues dans une liste sous un seul objet correspondant au modèle de mappage. Le modèle d'exemple suivant affiche la fusion :

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": $util.toJson($context)
}
```

Ce modèle est résolu dans le document de mappage suivant :

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

Chaque élément de la `payload` liste correspond à un seul article du lot. La fonction Lambda devrait également renvoyer une réponse sous forme de liste correspondant à l'ordre des éléments envoyés dans la demande :

```
[
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
 1
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
 2
  { "data": {...}, "errorMessage": null, "errorType": null }  // result for batch item
 3
]
```

Charge utile

Le `payload` champ est un conteneur utilisé pour transmettre n'importe quel JSON bien formé à la fonction Lambda. Si le `operation` champ est défini sur`BatchInvoke`, AWS AppSync regroupe les `payload` valeurs existantes dans une liste. Le champ `payload` est facultatif.

Type d'invocation

La source de données Lambda vous permet de définir deux types d'invocation : et. `RequestResponse` `Event` Les types d'invocation sont synonymes des types d'invocation définis dans l'API Lambda. Le type `RequestResponse` d'invocation permet d' AWS AppSync appeler votre fonction Lambda de manière synchrone pour attendre une réponse. L'`Event`invocation vous permet d'appeler votre fonction Lambda de manière asynchrone. Pour plus d'informations sur la façon dont Lambda gère les demandes de type `Event` invocation, consultez Invocation asynchrone.

Le champ `invocationType` est facultatif. Si ce champ n'est pas inclus dans la demande, le type d'`RequestResponse`appel AWS AppSync sera défini par défaut.

Quel que soit `invocationType` le champ, la demande résolue correspond à la charge utile d'entrée de la fonction Lambda. Modifions l'exemple ci-dessus :

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event"
  "payload": {
    "arguments": $util.toJson($context.arguments)
  }
}
```

Ceci est résolu et transmis à la fonction Lambda, qui pourrait ressembler à ceci :

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

Lorsque l'`BatchInvoke`opération est utilisée conjointement avec le champ de type d'`Event`appel, elle AWS AppSync fusionne le résolveur de champ de la même manière que celle mentionnée ci-dessus, et la demande est transmise à votre fonction Lambda sous forme d'événement asynchrone sous la forme d'une liste de valeurs. `payload` Nous vous recommandons de désactiver la mise en cache des résolveurs pour les résolveurs de type `Event` invocation, car ceux-ci ne seraient pas envoyés à Lambda en cas d'accès au cache.

Modèle de mappage des réponses

Comme pour les autres sources de données, votre fonction Lambda envoie une réponse AWS AppSync qui doit être convertie en un type GraphQL.

Le résultat de la fonction Lambda est défini sur l'`context`objet disponible via la propriété Velocity Template Language (VTL). `$context.result`

Si la forme de la réponse de la fonction Lambda correspond exactement à celle du type GraphQL, vous pouvez transmettre la réponse à l'aide du modèle de mappage de réponse suivant :

```
$util.toJson($context.result)
```

Il n'y a pas de champs obligatoires ni de restrictions des formes qui s'appliquent au modèle de mappage de la réponse. Toutefois, dans la mesure où GraphQL est fortement typé, le modèle de mappage résolu doit correspondre au type GraphQL prévu.

Réponse par lots de la fonction Lambda

Si le `operation` champ est défini sur`BatchInvoke`, AWS AppSync attend une liste d'éléments en retour de la fonction Lambda. Afin de AWS AppSync faire correspondre chaque résultat à l'élément de demande d'origine, la liste de réponses doit correspondre en taille et en ordre. Il est valide d'avoir des `null` éléments dans la liste de réponses ; elle `$ctx.result` est définie sur null en conséquence.

Résolveurs Lambda directs

Si vous souhaitez éviter complètement l'utilisation de modèles de mappage, vous AWS AppSync pouvez fournir une charge utile par défaut à votre fonction Lambda et une réponse de fonction Lambda par défaut à un type GraphQL. Vous pouvez choisir de fournir un modèle de demande, un modèle de réponse ou aucun des deux, et AWS AppSync le gérer en conséquence.

Modèle de mappage de requêtes Lambda direct

Lorsque le modèle de mappage de demandes n'est pas fourni, l'`Context`objet AWS AppSync sera envoyé directement à votre fonction Lambda sous forme d'`Invoke`opération. Pour plus d'informations sur la structure de l'objet `Context`, consultez Référence contextuelle du modèle de mappage Resolver.

Modèle de mappage des réponses Lambda directes

Lorsque le modèle de mappage des réponses n'est pas fourni, AWS AppSync effectuez l'une des deux opérations suivantes après avoir reçu la réponse de votre fonction Lambda. Si vous n'avez

pas fourni de modèle de mappage de demandes ou si vous avez fourni un modèle de mappage de demandes avec la version`2018-05-29`, la réponse sera équivalente au modèle de mappage de réponse suivant :

```
#if($ctx.error)
     $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
 #end
 $util.toJson($ctx.result)
```

Si vous avez fourni un modèle avec la version`2017-02-28`, la logique de réponse fonctionne de la même manière que le modèle de mappage des réponses suivant :

```
$util.toJson($ctx.result)
```

À première vue, le contournement du modèle de mappage fonctionne de la même manière que l'utilisation de certains modèles de mappage, comme indiqué dans les exemples précédents. Cependant, dans les coulisses, l'évaluation des modèles de mappage est totalement contournée. Dans la mesure où l'étape d'évaluation du modèle est contournée, les applications peuvent être confrontées à moins de surcharge et de latence pendant la réponse dans certains scénarios par rapport à une fonction Lambda avec un modèle de mappage des réponses qui doit être évalué.

Gestion personnalisée des erreurs dans les réponses Direct Lambda Resolver

Vous pouvez personnaliser les réponses aux erreurs issues des fonctions Lambda invoquées par les résolveurs Lambda directs en déclenchant une exception personnalisée. L'exemple suivant montre comment créer une exception personnalisée en utilisant JavaScript :

```
class CustomException extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomException";
  }
}

throw new CustomException("Custom message");
```

Lorsque des exceptions sont déclenchées, les `errorType` `name` et `errorMessage` `message` sont respectivement les et et de l'erreur personnalisée générée.

Dans `errorType` l'affirmative`UnauthorizedException`, AWS AppSync renvoie le message par défaut (`"You are not authorized to make this call."`) au lieu d'un message personnalisé.

L'extrait suivant est un exemple de réponse GraphQL illustrant une personnalisation : `errorType`

```
{
  "data": {
    "query": null
  },
  "errors": [
    {
      "path": [
        "query"
      ],
      "data": null,
      "errorType": "CustomException",
      "errorInfo": null,
      "locations": [
        {
          "line": 5,
          "column": 10,
          "sourceName": null
        }
      ],
      "message": "Custom Message"
    }
  ]
}
```

Résolveurs Lambda directs : le traitement par lots est activé

Vous pouvez activer le traitement par lots pour votre résolveur Lambda direct en le `maxBatchSize` configurant sur votre résolveur. Lorsqu'il `maxBatchSize` est défini sur une valeur supérieure à celle `0` d'un résolveur Lambda direct, il AWS AppSync envoie des requêtes par lots à votre fonction Lambda dans des tailles allant jusqu'à. `maxBatchSize`

Le réglage `maxBatchSize` `0` sur sur un résolveur Direct Lambda désactive le traitement par lots.

Pour plus d'informations sur le fonctionnement du traitement par lots avec des résolveurs Lambda, consultez. Cas d'utilisation avancé : traitement par lots

Modèle de mappage des demandes

Lorsque le traitement par lots est activé et que le modèle de mappage des demandes n'est pas fourni, AWS AppSync envoie une liste d'`Context`objets sous forme d'`BatchInvoke`opération directement à votre fonction Lambda.

Modèle de mappage des réponses

Lorsque le traitement par lots est activé et que le modèle de mappage des réponses n'est pas fourni, la logique de réponse est équivalente au modèle de mappage des réponses suivant :

```
#if( $context.result && $context.result.errorMessage )
      $utils.error($context.result.errorMessage, $context.result.errorType,
      $context.result.data)
#else
      $utils.toJson($context.result.data)
#end
```

La fonction Lambda doit renvoyer une liste de résultats dans le même ordre que la liste des `Context` objets envoyés. Vous pouvez renvoyer des erreurs individuelles en fournissant un `errorMessage` et `errorType` pour un résultat spécifique. Chaque résultat de la liste a le format suivant :

```
{
   "data" : { ... }, // your data
   "errorMessage" : { ... }, // optional, if included an error entry is added to the
 "errors" object in the AppSync response
   "errorType" : { ... } // optional, the error type
}
```

> **Note**
>
> Les autres champs de l'objet de résultat sont actuellement ignorés.

Gestion des erreurs liées à Lambda

Vous pouvez renvoyer une erreur pour tous les résultats en lançant une exception ou une erreur dans votre fonction Lambda. Si la taille de la demande de charge utile ou de réponse pour votre demande de lot est trop importante, Lambda renvoie une erreur. Dans ce cas, vous devez envisager de réduire votre charge utile de réponse `maxBatchSize` ou de réduire sa taille.

Pour plus d'informations sur la gestion des erreurs individuelles, voirRenvoi d'erreurs individuelles.

Exemples de fonctions Lambda

À l'aide du schéma ci-dessous, vous pouvez créer un résolveur Lambda direct pour le résolveur de `Post.relatedPosts` champs et activer le traitement par lots en définissant les paramètres ci-dessus : `maxBatchSize` `0`

```
schema {
    query: Query
    mutation: Mutation
}

type Query {
    getPost(id:ID!): Post
    allPosts: [Post]
}

type Mutation {
    addPost(id: ID!, author: String!, title: String, content: String, url: String):
 Post!
}

type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    relatedPosts: [Post]
}
```

Dans la requête suivante, la fonction Lambda sera appelée avec des lots de demandes à résoudre : `relatedPosts`

```
query getAllPosts {
  allPosts {
    id
    relatedPosts {
      id
    }
```

```
  }
}
```

Une implémentation simple d'une fonction Lambda est fournie ci-dessous :

```
const posts = {
  1: {
    id: '1',
    title: 'First book',
    author: 'Author1',
    url: 'https://amazon.com/',
    content:
      'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
 AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1',
    ups: '100',
    downs: '10',
  },
  2: {
    id: '2',
    title: 'Second book',
    author: 'Author2',
    url: 'https://amazon.com',
    content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT',
    ups: '100',
    downs: '10',
  },
  3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null, ups:
 null, downs: null },
  4: {
    id: '4',
    title: 'Fourth book',
    author: 'Author4',
    url: 'https://www.amazon.com/',
    content:
      'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4',
    ups: '1000',
    downs: '0',
  },
  5: {
    id: '5',
    title: 'Fifth book',
```

```
    author: 'Author5',
    url: 'https://www.amazon.com/',
    content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE
 TEXT AUTHOR 5 SAMPLE TEXT',
    ups: '50',
    downs: '0',
  },
}

const relatedPosts = {
  1: [posts['4']],
  2: [posts['3'], posts['5']],
  3: [posts['2'], posts['1']],
  4: [posts['2'], posts['1']],
  5: [],
}
exports.handler = async (event) => {
  console.log('event ->', event)
  // retrieve the ID of each post
  const ids = event.map((context) => context.source.id)
  // fetch the related posts for each post id
  const related = ids.map((id) => relatedPosts[id])

  // return the related posts; or an error if none were found
  return related.map((r) => {
    if (r.length > 0) {
      return { data: r }
    } else {
      return { data: null, errorMessage: 'Not found', errorType: 'ERROR' }
    }
  })
}
```

Référence du modèle de mappage Resolver pour EventBridge

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Le modèle de mappage du AWS AppSync résolveur utilisé avec la source de EventBridge données vous permet d'envoyer des événements personnalisés au EventBridge bus Amazon.

Modèle de mappage des demandes

Le modèle de mappage des `PutEvents` demandes vous permet d'envoyer plusieurs événements personnalisés à un bus d' EventBridgeévénements. Le document de mappage a la structure suivante :

```
{
    "version" : "2018-05-29",
    "operation" : "PutEvents",
    "events" : [{}]
}
```

Voici un exemple de modèle de mappage de demandes pour EventBridge :

```
{
    "version": "2018-05-29",
    "operation": "PutEvents",
    "events": [{
        "source": "com.mycompany.myapp",
        "detail": {
            "key1" : "value1",
            "key2" : "value2"
        },
        "detailType": "myDetailType1"
    },
    {
        "source": "com.mycompany.myapp",
        "detail": {
            "key3" : "value3",
            "key4" : "value4"
        },
        "detailType": "myDetailType2",
        "resources" : ["Resource1", "Resource2"],
        "time" : "2023-01-01T00:30:00.000Z"
    }

    ]
}
```

Modèle de mappage des réponses

Si l'`PutEvents`opération est réussie, la réponse du formulaire EventBridge est incluse dans le `$ctx.result` :

```
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end
  $util.toJson($ctx.result)
```

Les erreurs qui se produisent lors de l'exécution d'`PutEvents`opérations telles que `InternalExceptions` ou `Timeouts` apparaîtront dans`$ctx.error`. Pour obtenir la liste EventBridge des erreurs courantes, consultez la référence des erreurs EventBridge courantes.

Ce `result` sera dans le format suivant :

```
{
    "Entries" [
        {
            "ErrorCode" : String,
            "ErrorMessage" : String,
            "EventId" : String
        }
    ],
    "FailedEntryCount" : number
}
```

- Entrées

  L'événement ingéré entraîne à la fois un succès et un échec. Si l'ingestion a réussi, l'entrée `EventID` en contient. Sinon, vous pouvez utiliser le `ErrorCode` et `ErrorMessage` pour identifier le problème lié à l'entrée.

  Pour chaque enregistrement, l'index de l'élément de réponse est le même que celui du tableau de requêtes.
- FailedEntryCount

  Le nombre d'entrées ayant échoué. Cette valeur est représentée sous la forme d'un entier.

Pour plus d'informations sur la réponse de`PutEvents`, voir PutEvents.

Exemple de réponse 1

L'exemple suivant est une `PutEvents` opération avec deux événements réussis :

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ],
    "FailedEntryCount" : 0
}
```

Exemple de réponse 2

L'exemple suivant est une `PutEvents` opération comportant trois événements, deux réussites et un échec :

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        },
        {
            "ErrorCode" : "SampleErrorCode",
            "ErrorMessage" : "Sample Error Message"
        }
    ],
    "FailedEntryCount" : 1
}
```

`PutEvents` field

- Version

Commun à tous les modèles de mappage de demandes, le `version` champ définit la version utilisée par le modèle. Ce champ est obligatoire. La valeur `2018-05-29` est la seule version prise en charge pour les modèles de EventBridge mappage.

- Opération

  La seule opération prise en charge est`PutEvents`. Cette opération vous permet d'ajouter des événements personnalisés à votre bus d'événements.

- Evénements

  Un ensemble d'événements qui seront ajoutés au bus d'événements. Ce tableau doit avoir une allocation de 1 à 10 éléments.

  L'`Event`objet est un objet JSON valide qui comporte les champs suivants :

  - `"source"`: chaîne qui définit la source de l'événement.
  - `"detail"`: objet JSON que vous pouvez utiliser pour joindre des informations sur l'événement. Ce champ peut être une carte vide (`{ }`).
  - `"detailType`: chaîne identifiant le type d'événement.
  - `"resources"`: tableau JSON de chaînes identifiant les ressources impliquées dans l'événement. Ce champ peut être un tableau vide.
  - `"time"`: l'horodatage de l'événement fourni sous forme de chaîne. Cela doit suivre le format d'horodatage de la RFC3339.

Les extraits ci-dessous sont des exemples d'objets valides `Event` :

Exemple 1

```
{
    "source" : "source1",
    "detail" : {
        "key1" : [1,2,3,4],
        "key2" : "strval"
    },
    "detailType" : "sampleDetailType",
    "resources" : ["Resouce1", "Resource2"],
    "time" : "2022-01-10T05:00:10Z"
}
```

Exemple 2

```
{
    "source" : "source1",
    "detail" : {},
    "detailType" : "sampleDetailType"
}
```

Exemple 3

```
{
    "source" : "source1",
    "detail" : {
        "key1" : 1200
    },
    "detailType" : "sampleDetailType",
    "resources" : []
}
```

Référence du modèle de mappage Resolver pour la source de données None

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Le modèle de mappage du AWS AppSync résolveur utilisé avec la source de données de type None vous permet de façonner les demandes pour les opérations AWS AppSync locales.

Modèle de mappage des demandes

Le modèle de mappage est simple et vous permet de transmettre autant d'informations contextuelles que possible via le champ `payload`.

```
{
   "version": string,
```

```
   "payload": any type
}
```

Voici la représentation du schéma JSON du modèle de mappage de demande, une fois qu'il est résolu :

```
{
    "definitions": {},
    "$schema": "https://json-schema.org/draft-06/schema#",
    "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
    "type": "object",
    "properties": {
        "version": {
            "$id": "/properties/version",
            "type": "string",
            "enum": [
                "2018-05-29"
            ],
            "title": "The Mapping template version.",
            "default": "2018-05-29"
        },
        "payload": {}
    },
    "required": [
        "version"
    ],
    "additionalProperties": false
}
```

Voici un exemple où les arguments du champ sont transmis via la propriété `$context.arguments` de contexte VTL :

```
{
    "version": "2018-05-29",
    "payload": $util.toJson($context.arguments)
}
```

La valeur du champ `payload` est transmise au modèle de mappage de réponse et est alors disponible dans la propriété de contexte VTL (`$context.result`).

Voici un exemple représentant la valeur interpolée du champ `payload` :

```
{
    "id": "postId1"
}
```

Version

Commun à tous les modèles de mappage de demandes, le `version` champ définit la version utilisée par le modèle.

Le champ `version` est obligatoire.

Exemple :

```
"version": "2018-05-29"
```

Charge utile

Le champ `payload` est un conteneur qui peut être utilisé pour transmettre tout élément JSON de format correct au modèle de mappage de réponse.

Le champ `payload` est facultatif.

Modèle de mappage des réponses

Étant donné qu'il n'y a pas de source de données, la valeur du champ `payload` est transmise au modèle de mappage de réponse et définie sur l'objet `context` qui est disponible via la propriété VTL `$context.result`.

Si la forme de la valeur du champ `payload` correspond exactement à celle du type GraphQL, vous pouvez transmettre la réponse à l'aide du modèle de mappage de réponse suivant :

```
$util.toJson($context.result)
```

Il n'y a pas de champs obligatoires ni de restrictions des formes qui s'appliquent au modèle de mappage de la réponse. Toutefois, dans la mesure où GraphQL est fortement typé, le modèle de mappage résolu doit correspondre au type GraphQL prévu.

Référence du modèle de mappage du résolveur pour HTTP

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Les modèle de mappage des résolveurs HTTP d'AWS AppSync vous permettent d'envoyer des demandes d'AWS AppSync vers n'importe quel point de terminaison HTTP, et des réponses de votre point de terminaison HTTP vers AWS AppSync. En utilisant les modèles de mappage, vous pouvez donner des conseils à AWS AppSync sur la nature de l'opération à appeler. Cette section décrit les différents modèles de mappage pour le résolveur HTTP pris en charge.

Modèle de mappage de demande

```
{
    "version": "2018-05-29",
    "method": "PUT|POST|GET|DELETE|PATCH",
    "params": {
        "query": Map,
        "headers": Map,
        "body": any
    },
    "resourcePath": string
}
```

Une fois le modèle de mappage de demande HTTP résolu, la représentation du schéma JSON du modèle de mappage de demande ressemble à ce qui suit :

```
{
    "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
    "type": "object",
    "properties": {
        "version": {
        "$id": "/properties/version",
        "type": "string",
        "title": "The Version Schema ",
        "default": "",
        "examples": [
```

```
            "2018-05-29"
        ],
        "enum": [
            "2018-05-29"
        ]
        },
        "method": {
        "$id": "/properties/method",
        "type": "string",
        "title": "The Method Schema ",
        "default": "",
        "examples": [
            "PUT|POST|GET|DELETE|PATCH"
        ],
        "enum": [
            "PUT",
            "PATCH",
            "POST",
            "DELETE",
            "GET"
        ]
        },
        "params": {
        "$id": "/properties/params",
        "type": "object",
        "properties": {
            "query": {
            "$id": "/properties/params/properties/query",
            "type": "object"
            },
            "headers": {
            "$id": "/properties/params/properties/headers",
            "type": "object"
            },
            "body": {
            "$id": "/properties/params/properties/body",
            "type": "string",
            "title": "The Body Schema ",
            "default": "",
            "examples": [
                ""
            ]
            }
        }
```

```
        },
        "resourcePath": {
        "$id": "/properties/resourcePath",
        "type": "string",
        "title": "The Resourcepath Schema ",
        "default": "",
        "examples": [
            ""
        ]
        }
    },
    "required": [
        "version",
        "method",
        "resourcePath"
    ]
}
```

Voici un exemple de demande HTTP POST, avec un corps `text/plain` :

```
{
    "version": "2018-05-29",
    "method": "POST",
    "params": {
        "headers":{
        "Content-Type":"text/plain"
        },
        "body":"this is an example of text body"
    },
    "resourcePath": "/"
}
```

Version

Modèle de mappage de demande uniquement

Définit la version utilisée par le modèle. `version` est commun à tous les modèles de mappage de demande et obligatoire.

```
"version": "2018-05-29"
```

Méthode

Modèle de mappage de demande uniquement

Méthode ou verbe HTTP (GET, POST, PUT, PATCH ou DELETE) envoyé par AWS AppSync au point de terminaison HTTP.

```
"method": "PUT"
```

ResourcePath

Modèle de mappage de demande uniquement

Chemin de la ressource à laquelle vous souhaitez accéder. Avec le point de terminaison dans la source de données HTTP, le chemin de la ressource constitue l'URL à laquelle le service AWS AppSync adresse une demande.

```
"resourcePath": "/v1/users"
```

Lorsque le modèle de mappage est évalué, ce chemin est envoyé dans le cadre de la demande HTTP, y compris le point de terminaison HTTP. Ainsi, l'exemple précédent pourrait donner :

```
PUT <endpoint>/v1/users
```

Champ Params

Modèle de mappage de demande uniquement

Permet de spécifier l'action exécutée par votre recherche, plus couramment en définissant la valeur de query à l'intérieur de la section body. Toutefois, il existe plusieurs autres capacités qui peuvent être configurées, comme la mise en forme des réponses.

headers

Informations d'en-tête, sous forme de paires clé-valeur. La clé et la valeur doivent être des chaînes.

Par exemple :

```
"headers" : {
```

```
    "Content-Type" : "application/json"
}
```

Les en-têtes `Content-Type` actuellement pris en charge sont les suivants :

```
text/*
application/xml
application/json
application/soap+xml
application/x-amz-json-1.0
application/x-amz-json-1.1
application/vnd.api+json
application/x-ndjson
```

Remarque : vous ne pouvez pas définir les en-têtes HTTP suivants :

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

Paires clé-valeur qui spécifient les options courantes, telles que la mise en forme du code pour les réponses JSON. La clé et la valeur doivent être une chaîne. L'exemple suivant montre comment envoyer une chaîne de requête comme `?type=json` :

```
"query" : {
    "type" : "json"
}
```

body

Le corps contient le corps de la demande HTTP que vous choisissez de définir. Le corps de la demande est toujours une chaîne encodée en UTF-8, sauf si le type de contenu spécifie le jeu de caractères.

```
"body":"body string"
```

Autorités de certification (CA) reconnues parAWS AppSyncpour les points de terminaison HTTPS

> **Note**
>
> Let's Encrypt est accepté via leidentrustetisr grootx1certificats. Aucune action de votre part n'est requise si vous utilisez Let's Encrypt.

Pour le moment, les certificats auto-signés ne sont pas pris en charge par les résolveurs HTTP lors de l'utilisation de HTTPS.AWS AppSync reconnaît les autorités de certification suivantes lors de la résolution des certificats SSL/TLS pour HTTPS :

Certificats racines connus dansAWS AppSync

Nom	Date	Empreintes digitales SHA1
`digicertassuredidrootca`	21 avril 2018	`05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43`
`trustcenterclass2caii`	21 avril 2018	`AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E`
`thawtepremiumserverca`	21 avril 2018	`E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:66:66`
`cia-crt-g3-02-ca`	23 novembre 2016	`96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09`
`swisssignplatinumg2ca`	21 avril 2018	`56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66`
`swisssignsilverg2ca`	21 avril 2018	`9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB`
`thawteserverca`	21 avril 2018	`9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79`

Nom	Date	Empreintes digitales SHA1
equifaxsecureebusinessca1	21 avril 2018	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
securetrustca	21 avril 2018	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
utnuserfirstclientauthemailca	21 avril 2018	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
thawtepersonalfreemailca	21 avril 2018	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
affirmtrustnetworkingca	21 avril 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
entrustevca	21 avril 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
utnuserfirsthardwareca	21 avril 2018	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
certumca	21 avril 2018	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
addtrustclass1ca	21 avril 2018	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
entrustrootcag2	21 avril 2018	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
equifaxsecureca	21 avril 2018	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
quovadisrootca3	21 avril 2018	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	21 avril 2018	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7

Nom	Date	Empreintes digitales SHA1
digicertglobalroot g2	21 avril 2018	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73: FE:06:D1:CC:8D:4F:82:A4
digicerthighassura nceevrootca	21 avril 2018	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A: E6:D3:8F:1A:61:C7:DC:25
secomvalicertclass 1ca	21 avril 2018	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB: 8C:E8:6A:81:10:9F:E4:8E
equifaxsecureglobal ebusinessca1	21 avril 2018	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35: 98:64:B8:2D:82:BD:1A:36
geotrustuniversalc a	21 avril 2018	E6:21:F3:35:43:79:05:9A:4B:68:30:9D: 8A:2F:74:22:15:87:EC:79
deprecateditsecca	27 janvier 2012	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5: DE:13:6E:83:5A:29:72:9D
verisignclass3ca	21 avril 2018	A1:DB:63:93:91:6F:17:E4:18:55:09:40: 04:15:C7:02:40:B0:AE:6B
thawteprimaryrootc ag3	21 avril 2018	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43: 5B:17:15:89:CA:F3:6B:F2
thawteprimaryrootc ag2	21 avril 2018	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38: DD:F4:1D:DB:08:9E:F0:12
deutschetelekomroo tca2	21 avril 2018	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD: D6:13:30:FD:8C:DE:37:BF
buypassclass3ca	21 avril 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD: C7:C2:81:A5:BC:A9:64:57
utnuserfirstobject ca	21 avril 2018	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B: AC:1D:81:D8:38:5E:2D:46
geotrustprimaryca	21 avril 2018	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2: 10:0D:D6:02:90:37:F0:96

Nom	Date	Empreintes digitales SHA1
buypassclass2ca	21 avril 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
baltimorecodesigningca	21 avril 2018	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D
verisignclass1ca	21 avril 2018	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1
baltimorecybertrustca	21 avril 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
starfieldclass2ca	21 avril 2018	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
camerfirmachamberscommerceca	21 avril 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
ttelesecglobalrootclass3ca	21 avril 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
verisignclass3g5ca	21 avril 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
ttelesecglobalrootclass2ca	21 avril 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
trustcenteruniversalcai	21 avril 2018	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
verisignclass3g4ca	21 avril 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3g3ca	21 avril 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
xrampglobalca	21 avril 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6

Nom	Date	Empreintes digitales SHA1
amzninternalrootca	12 décembre 2008	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06
certplusclass3pprimaryca	21 avril 2018	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
certumtrustednetworkca	21 avril 2018	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
verisignclass3g2ca	21 avril 2018	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
globalsignr3ca	21 avril 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
utndatacorpsgcca	21 avril 2018	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
secomscrootca2	21 avril 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
gtecybertrustgloballca	21 avril 2018	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
secomscrootca1	21 avril 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
affirmtrustcommercialca	21 avril 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
trustcenterclass4caii	21 avril 2018	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
verisignuniversalrootca	21 avril 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54

Nom	Date	Empreintes digitales SHA1
globalsignr2ca	21 avril 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
certplusclass2primaryca	21 avril 2018	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
digicertglobalrootca	21 avril 2018	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
globalsignca	21 avril 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
thawteprimaryrootca	21 avril 2018	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
starfieldrootg2ca	21 avril 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
geotrustglobalca	21 avril 2018	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
soneraclass2ca	21 avril 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
verisigntsaca	21 avril 2018	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
soneraclass1ca	21 avril 2018	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
quovadisrootca	21 avril 2018	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
affirmtrustpremiumeccca	21 avril 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
starfieldservicesrootg2ca	21 avril 2018	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F

Nom	Date	Empreintes digitales SHA1
valicertclass2ca	21 avril 2018	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
comodoaaaca	21 avril 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
aolrootca2	21 avril 2018	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
keynectisrootca	21 avril 2018	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
addtrustqualifiedca	21 avril 2018	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
aolrootca1	21 avril 2018	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignclass2g3ca	21 avril 2018	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
addtrustexternalca	21 avril 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
verisignclass2g2ca	21 avril 2018	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
geotrustprimarycag3	21 avril 2018	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycag2	21 avril 2018	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
swisssigngoldg2ca	21 avril 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
entrust2048ca	21 avril 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31

Nom	Date	Empreintes digitales SHA1
chunghwaepkirootca	21 avril 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
camerfirmachambersignca	21 avril 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
camerfirmachambersca	21 avril 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
godaddyclass2ca	21 avril 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
affirmtrustpremiumca	21 avril 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
verisignclass1g3ca	21 avril 2018	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
secomevrootca1	21 avril 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
verisignclass1g2ca	21 avril 2018	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
amzninternalinfosecag3	27 février 2015	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
cia-crt-g3-01-ca	23 novembre 2016	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
godaddyrootg2ca	21 avril 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
digicertassuredidrootca	21 avril 2018	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43

Nom	Date	Empreintes digitales SHA1
microseceszignorootca2009	21 avril 2018	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
affirmtrustcommercial	21 avril 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
comodoecccertificationauthority	21 avril 2018	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
cadisigrootr2	21 avril 2018	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
swisssignsilvercag2	21 avril 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
securetrustca	21 avril 2018	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
cadisigrootr1	21 avril 2018	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
accvraiz1	21 avril 2018	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
entrustrootcertificationauthority	21 avril 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
camerfirmaglobalchambersignroot	21 avril 2018	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
dstacescax6	21 avril 2018	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
identrustpublicsectorrootca1	21 avril 2018	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
starfieldrootcertificateauthorityg2	21 avril 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E

Nom	Date	Empreintes digitales SHA1
secureglobalca	21 avril 2018	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
eecertificationcentrerootca	21 avril 2018	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
opentrustrootcag3	21 avril 2018	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8A:A6
teliasonerarootcav1	21 avril 2018	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
autoridaddecertificacionfirmaprofesionalcifa62634068	21 avril 2018	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
opentrustrootcag2	21 avril 2018	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	21 avril 2018	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:D5:7E
globalsigneccrootcar5	21 avril 2018	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
globalsigneccrootcar4	21 avril 2018	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
izenpecom	21 avril 2018	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
turktrustelektroniksertifikahizmetsaglayicisih5	21 avril 2018	C4:18:F6:4D:46:D1:DF:00:3D:27:30:13:72:43:A9:12:11:C6:75:FB

Nom	Date	Empreintes digitales SHA1
gdcatrustauthr5root	21 avril 2018	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
dtrustrootclass3ca22009	21 avril 2018	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
quovadisrootca3	21 avril 2018	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	21 avril 2018	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
geotrustprimarycertificationauthorityg3	21 avril 2018	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycertificationauthorityg2	21 avril 2018	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
oistewisekeyglobalrootgbca	21 avril 2018	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
addtrustexternalroot	21 avril 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
chambersofcommerceroot2008	21 avril 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
digicertglobalrootg3	21 avril 2018	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E
comodoaaaservicesroot	21 avril 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
digicertglobalrootg2	21 avril 2018	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4

Nom	Date	Empreintes digitales SHA1
certinomisrootca	21 avril 2018	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:88:A8
oistewisekeyglobalrootgaca	21 avril 2018	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
dstrootcax3	21 avril 2018	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
certigna	21 avril 2018	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
digicerthighassurancеevrootca	21 avril 2018	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
soneraclass2rootca	21 avril 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
trustcorrootcertca2	21 avril 2018	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
usertrustrsacertificationauthority	21 avril 2018	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
trustcorrootcertca1	21 avril 2018	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
geotrustuniversalca	21 avril 2018	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
certsignrootca	21 avril 2018	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
amazonrootca4	21 avril 2018	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
amazonrootca3	21 avril 2018	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E

Nom	Date	Empreintes digitales SHA1
amazonrootca2	21 avril 2018	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
verisignuniversalrootcertificationauthority	21 avril 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
amazonrootca1	21 avril 2018	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
networksolutionscertificate authority	21 avril 2018	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
thawteprimaryrootcag3	21 avril 2018	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
affirmtrustnetworking	21 avril 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
thawteprimaryrootcag2	21 avril 2018	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
trustcoreca1	21 avril 2018	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
deutschetelekomrootca2	21 avril 2018	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
godaddyrootcertificateauthorityg2	21 avril 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
entrustrootcertificationauthorityec1	21 avril 2018	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47

Nom	Date	Empreintes digitales SHA1
szafirrootca2	21 avril 2018	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE
tubitakkamusmsslkoksertifikasisurum1	21 avril 2018	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
buypassclass3rootca	21 avril 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
comodorsacertificationauthority	21 avril 2018	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
netlockaranyclassgoldfotanusitvany	21 avril 2018	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
securitycommunicationrootca2	21 avril 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
dtrustrootclass3ca2ev2009	21 avril 2018	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
starfieldclass2ca	21 avril 2018	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
pscprocert	21 avril 2018	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
actalisauthenticationrootca	21 avril 2018	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
staatdernederlandenrootcag3	21 avril 2018	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
cfcaevroot	21 avril 2018	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83

Nom	Date	Empreintes digitales SHA1
digicerttrustedroo tg4	21 avril 2018	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F: C8:3A:4D:7D:77:5D:05:E4
staatdernederlande nrootcag2	21 avril 2018	59:AF:82:79:91:86:C7:B4:75:07:CB:CF: 03:57:46:EB:04:DD:B7:16
securitycommunicat ionevrootca1	21 avril 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8: 90:8F:FD:28:86:65:64:7D
globalsignrootcar3	21 avril 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1: 09:26:DF:5B:85:69:76:AD
globalsignrootcar2	21 avril 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F: DD:DE:38:E4:B7:24:2E:FE
certumtrustednetwo rkca2	21 avril 2018	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5: FA:76:26:CF:D3:DC:30:92
acraizfnmtrcm	21 avril 2018	EC:50:35:07:B2:15:C4:95:62:19:E2:A8: 9A:5B:42:99:2C:4C:2C:20
hellenicacademican dresearch instituti onseccrootca2015	21 avril 2018	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4: BC:6F:84:68:0B:BA:B6:66
certplusrootcag2	21 avril 2018	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47: 41:C9:54:25:5D:69:CC:1A
twcarootcertificat ionauthority	21 avril 2018	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5: A3:7A:A0:76:A9:06:23:48
twcaglobalrootca	21 avril 2018	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32: 52:55:60:13:F5:AD:AF:65
certplusrootcag1	21 avril 2018	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0: AC:A6:7B:6A:1F:E3:F7:66

Nom	Date	Empreintes digitales SHA1
geotrustuniversalca2	21 avril 2018	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
baltimorecybertrustroot	21 avril 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
buypassclass2rootca	21 avril 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
certumtrustednetworkca	21 avril 2018	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
digicertassuredidrootg3	21 avril 2018	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
digicertassuredidrootg2	21 avril 2018	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F
isrgrootx1	21 avril 2018	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
entrustnetpremium2048secureserverca	21 avril 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
certplusclass2primaryca	21 avril 2018	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
digicertglobalrootca	21 avril 2018	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
entrustrootcertificationauthorityg2	21 avril 2018	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
starfieldservicesrootcertificateauthorityg2	21 avril 2018	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F

Nom	Date	Empreintes digitales SHA1
thawteprimaryrootca	21 avril 2018	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
atostrustedroot2011	21 avril 2018	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
geotrustglobalca	21 avril 2018	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
luxtrustglobalroot2	21 avril 2018	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5F:3F
etugracertificationauthority	21 avril 2018	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
visaecommerceroot	21 avril 2018	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
quovadisrootca	21 avril 2018	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
identrustcommercialrootca1	21 avril 2018	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
staatdernederlandenevrootca	21 avril 2018	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
ttelesecglobalrootclass3	21 avril 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
ttelesecglobalrootclass2	21 avril 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
comodocertificationauthority	21 avril 2018	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
securitycommunicationrootca	21 avril 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7

Nom	Date	Empreintes digitales SHA1
quovadisrootca3g3	21 avril 2018	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D
xrampglobalcaroot	21 avril 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
securesignrootca11	21 avril 2018	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
affirmtrustpremium	21 avril 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
globalsignrootca	21 avril 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
swisssigngoldcag2	21 avril 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
quovadisrootca2g3	21 avril 2018	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
affirmtrustpremium ecc	21 avril 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
geotrustprimarycer tificatio nauthority	21 avril 2018	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
quovadisrootca1g3	21 avril 2018	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67
hongkongpostrootca 1	21 avril 2018	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
usertrustecccertif icationauthority	21 avril 2018	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0

Nom	Date	Empreintes digitales SHA1
cybertrustglobalroot	21 avril 2018	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
godaddyclass2ca	21 avril 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
hellenicacademicandresearchinstitutionsrootca2015	21 avril 2018	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
ecacc	21 avril 2018	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:F2:E8
hellenicacademicandresearchinstitutionsrootca2011	21 avril 2018	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
verisignclass3publicprimarycertificationauthorityg5	21 avril 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
verisignclass3publicprimarycertificationauthorityg4	21 avril 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3publicprimarycertificationauthorityg3	21 avril 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
trustisfpsrootca	21 avril 2018	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04

Nom	Date	Empreintes digitales SHA1
epkirootcertificationauthority	21 avril 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
globalchambersignroot2008	21 avril 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
camerfirmachambersofcommerceroot	21 avril 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert81.pem	13 mars 2014	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
mozillacert99.pem	13 mars 2014	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
mozillacert145.pem	13 mars 2014	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:3A:04
mozillacert37.pem	13 mars 2014	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert4.pem	13 mars 2014	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B
mozillacert70.pem	13 mars 2014	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
mozillacert88.pem	13 mars 2014	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
mozillacert134.pem	13 mars 2014	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
mozillacert26.pem	13 mars 2014	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
mozillacert77.pem	13 mars 2014	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6

Nom	Date	Empreintes digitales SHA1
mozillacert123.pem	13 mars 2014	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10:DD:6B:DF:99:72:2C:96:E5
mozillacert15.pem	13 mars 2014	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
mozillacert66.pem	13 mars 2014	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34
mozillacert112.pem	13 mars 2014	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
mozillacert55.pem	13 mars 2014	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
mozillacert101.pem	13 mars 2014	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
mozillacert119.pem	13 mars 2014	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
mozillacert44.pem	13 mars 2014	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
mozillacert108.pem	13 mars 2014	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
mozillacert95.pem	13 mars 2014	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert141.pem	13 mars 2014	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
mozillacert33.pem	13 mars 2014	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
mozillacert0.pem	13 mars 2014	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74

Nom	Date	Empreintes digitales SHA1
mozillacert84.pem	13 mars 2014	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2
mozillacert130.pem	13 mars 2014	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert148.pem	13 mars 2014	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
mozillacert22.pem	13 mars 2014	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
mozillacert7.pem	13 mars 2014	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
mozillacert73.pem	13 mars 2014	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
mozillacert137.pem	13 mars 2014	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A1:D1
mozillacert11.pem	13 mars 2014	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
mozillacert29.pem	13 mars 2014	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
mozillacert62.pem	13 mars 2014	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert126.pem	13 mars 2014	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
mozillacert18.pem	13 mars 2014	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
mozillacert51.pem	13 mars 2014	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B

Nom	Date	Empreintes digitales SHA1
mozillacert69.pem	13 mars 2014	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
mozillacert115.pem	13 mars 2014	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert40.pem	13 mars 2014	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
mozillacert58.pem	13 mars 2014	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
mozillacert104.pem	13 mars 2014	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E
mozillacert91.pem	13 mars 2014	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
mozillacert47.pem	13 mars 2014	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
mozillacert80.pem	13 mars 2014	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert98.pem	13 mars 2014	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
mozillacert144.pem	13 mars 2014	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
mozillacert36.pem	13 mars 2014	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D
mozillacert3.pem	13 mars 2014	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
mozillacert87.pem	13 mars 2014	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74

Nom	Date	Empreintes digitales SHA1
mozillacert133.pem	13 mars 2014	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
mozillacert25.pem	13 mars 2014	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
mozillacert76.pem	13 mars 2014	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert122.pem	13 mars 2014	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
mozillacert14.pem	13 mars 2014	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
mozillacert65.pem	13 mars 2014	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:35:FB
mozillacert111.pem	13 mars 2014	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
mozillacert129.pem	13 mars 2014	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
mozillacert54.pem	13 mars 2014	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
mozillacert100.pem	13 mars 2014	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
mozillacert118.pem	13 mars 2014	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45
mozillacert151.pem	13 mars 2014	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
mozillacert43.pem	13 mars 2014	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A

Nom	Date	Empreintes digitales SHA1
mozillacert107.pem	13 mars 2014	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
mozillacert94.pem	13 mars 2014	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
mozillacert140.pem	13 mars 2014	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert32.pem	13 mars 2014	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C
mozillacert83.pem	13 mars 2014	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:EA:46
mozillacert147.pem	13 mars 2014	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
mozillacert21.pem	13 mars 2014	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
mozillacert39.pem	13 mars 2014	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
mozillacert6.pem	13 mars 2014	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
mozillacert72.pem	13 mars 2014	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
mozillacert136.pem	13 mars 2014	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert10.pem	13 mars 2014	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert28.pem	13 mars 2014	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B

Nom	Date	Empreintes digitales SHA1
mozillacert61.pem	13 mars 2014	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
mozillacert79.pem	13 mars 2014	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert125.pem	13 mars 2014	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert17.pem	13 mars 2014	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert50.pem	13 mars 2014	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
mozillacert68.pem	13 mars 2014	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
mozillacert114.pem	13 mars 2014	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert57.pem	13 mars 2014	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
mozillacert103.pem	13 mars 2014	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert90.pem	13 mars 2014	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
mozillacert46.pem	13 mars 2014	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
mozillacert97.pem	13 mars 2014	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
mozillacert143.pem	13 mars 2014	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7

Nom	Date	Empreintes digitales SHA1
mozillacert35.pem	13 mars 2014	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
mozillacert2.pem	13 mars 2014	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
mozillacert86.pem	13 mars 2014	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
mozillacert132.pem	13 mars 2014	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert24.pem	13 mars 2014	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
mozillacert9.pem	13 mars 2014	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
mozillacert75.pem	13 mars 2014	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
mozillacert121.pem	13 mars 2014	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
mozillacert139.pem	13 mars 2014	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert13.pem	13 mars 2014	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
mozillacert64.pem	13 mars 2014	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
mozillacert110.pem	13 mars 2014	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert128.pem	13 mars 2014	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D

Nom	Date	Empreintes digitales SHA1
mozillacert53.pem	13 mars 2014	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
mozillacert117.pem	13 mars 2014	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
mozillacert150.pem	13 mars 2014	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
mozillacert42.pem	13 mars 2014	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
mozillacert106.pem	13 mars 2014	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
mozillacert93.pem	13 mars 2014	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17
mozillacert31.pem	13 mars 2014	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
mozillacert49.pem	13 mars 2014	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71
mozillacert82.pem	13 mars 2014	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
mozillacert146.pem	13 mars 2014	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A
mozillacert20.pem	13 mars 2014	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert38.pem	13 mars 2014	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36
mozillacert5.pem	13 mars 2014	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6

Nom	Date	Empreintes digitales SHA1
mozillacert71.pem	13 mars 2014	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert89.pem	13 mars 2014	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
mozillacert135.pem	13 mars 2014	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert27.pem	13 mars 2014	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
mozillacert60.pem	13 mars 2014	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert78.pem	13 mars 2014	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert124.pem	13 mars 2014	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert16.pem	13 mars 2014	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
mozillacert67.pem	13 mars 2014	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
mozillacert113.pem	13 mars 2014	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
mozillacert56.pem	13 mars 2014	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
mozillacert102.pem	13 mars 2014	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
mozillacert45.pem	13 mars 2014	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0

Nom	Date	Empreintes digitales SHA1
mozillacert109.pem	13 mars 2014	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
mozillacert96.pem	13 mars 2014	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
mozillacert142.pem	13 mars 2014	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
mozillacert34.pem	13 mars 2014	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
mozillacert1.pem	13 mars 2014	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
mozillacert85.pem	13 mars 2014	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
mozillacert131.pem	13 mars 2014	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
mozillacert149.pem	13 mars 2014	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert23.pem	13 mars 2014	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
mozillacert8.pem	13 mars 2014	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:6A:0F
mozillacert74.pem	13 mars 2014	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert120.pem	13 mars 2014	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
mozillacert138.pem	13 mars 2014	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD

Nom	Date	Empreintes digitales SHA1
mozillacert12.pem	13 mars 2014	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
mozillacert63.pem	13 mars 2014	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
mozillacert127.pem	13 mars 2014	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert19.pem	13 mars 2014	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
mozillacert52.pem	13 mars 2014	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F
mozillacert116.pem	13 mars 2014	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert41.pem	13 mars 2014	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
mozillacert59.pem	13 mars 2014	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
mozillacert105.pem	13 mars 2014	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC
mozillacert92.pem	13 mars 2014	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0
mozillacert30.pem	13 mars 2014	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
mozillacert48.pem	13 mars 2014	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC
verisignc4g2.pem	20 mars 2014	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52

Nom	Date	Empreintes digitales SHA1
verisignc2g3.pem	20 mars 2014	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
verisignc1g6.pem	31 décembre 2014	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
verisignc2g2.pem	20 mars 2014	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
verisignroot.pem	20 mars 2014	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
verisignc2g1.pem	20 mars 2014	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:14:2A
verisignc3g5.pem	20 mars 2014	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
verisignc1g3.pem	20 mars 2014	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
verisignc3g4.pem	20 mars 2014	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignc1g2.pem	20 mars 2014	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
verisignc3g3.pem	20 mars 2014	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
verisignc1g1.pem	20 mars 2014	90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:55:6F
verisignc3g2.pem	20 mars 2014	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F

Nom	Date	Empreintes digitales SHA1
verisignc3g1.pem	20 mars 2014	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
verisignc2g6.pem	31 décembre 2014	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F
verisignc4g3.pem	20 mars 2014	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
gdroot-g2.pem	31 décembre 2014	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
gd-class2-root.pem	31 décembre 2014	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
gd_bundle-g2.pem	31 décembre 2014	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8
dstacescax6	18 juin 2018	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
gd_bundle-g2.pem	18 juin 2018	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8
verisignc4g3.pem	18 juin 2018	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
swisssignplatinumg2ca	21 avril 2018	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66
geotrustprimarycertificationauthorityg3	18 juin 2018	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD

Nom	Date	Empreintes digitales SHA1
geotrustprimarycertificationauthorityg2	18 juin 2018	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
buypassclass2rootca	18 juin 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
camerfirmachambersofcommerceroot	18 juin 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert20.pem	18 juin 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert12.pem	18 juin 2018	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
mozillacert90.pem	18 juin 2018	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
mozillacert82.pem	18 juin 2018	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
mozillacert140.pem	18 juin 2018	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert74.pem	18 juin 2018	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert132.pem	18 juin 2018	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert66.pem	18 juin 2018	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34
mozillacert124.pem	18 juin 2018	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF

Nom	Date	Empreintes digitales SHA1
mozillacert58.pem	18 juin 2018	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
securitycommunicationrootca2	18 juin 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert116.pem	18 juin 2018	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert108.pem	18 juin 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
certigna	18 juin 2018	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert3.pem	18 juin 2018	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
verisignc1g1.pem	18 juin 2018	90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:55:6F
verisignc4g2.pem	18 juin 2018	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52
deutschetelekomrootca2	18 juin 2018	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
starfieldrootg2ca	21 avril 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
comodoecccertificationauthority	18 juin 2018	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
digicertglobalrootg3	18 juin 2018	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E
digicertglobalrootg2	18 juin 2018	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4

Nom	Date	Empreintes digitales SHA1
mozillacert11.pem	18 juin 2018	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
mozillacert81.pem	18 juin 2018	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
mozillacert73.pem	18 juin 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
szafirrootca2	18 juin 2018	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE
mozillacert131.pem	18 juin 2018	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
ecacc	18 juin 2018	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:F2:E8
mozillacert65.pem	18 juin 2018	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:35:FB
turktrustelektroniksertifikahizmetsaglayicisih5	18 juin 2018	C4:18:F6:4D:46:D1:DF:00:3D:27:30:13:72:43:A9:12:11:C6:75:FB
mozillacert123.pem	18 juin 2018	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10:DD:6B:DF:99:72:2C:96:E5
mozillacert57.pem	18 juin 2018	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
mozillacert115.pem	18 juin 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert49.pem	18 juin 2018	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71

Nom	Date	Empreintes digitales SHA1
mozillacert107.pem	18 juin 2018	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
verisignclass3g4ca	21 avril 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
securetrustca	18 juin 2018	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
mozillacert2.pem	18 juin 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
buypassclass2ca	21 avril 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
secomscrootca2	21 avril 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
secomscrootca1	21 avril 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
trustisfpsrootca	18 juin 2018	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
hongkongpostrootca1	18 juin 2018	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
certsignrootca	18 juin 2018	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
geotrustprimaryca	21 avril 2018	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
twcaglobalrootca	18 juin 2018	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
camerfirmachambersca	21 avril 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C

Nom	Date	Empreintes digitales SHA1
mozillacert10.pem	18 juin 2018	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert80.pem	18 juin 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert72.pem	18 juin 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
comodoaaaca	21 avril 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert130.pem	18 juin 2018	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert64.pem	18 juin 2018	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
mozillacert122.pem	18 juin 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
mozillacert56.pem	18 juin 2018	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
equifaxsecureebusinessca1	21 avril 2018	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
camerfirmachambersignca	21 avril 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert114.pem	18 juin 2018	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert48.pem	18 juin 2018	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC
pscprocert	18 juin 2018	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74

Nom	Date	Empreintes digitales SHA1
mozillacert106.pem	18 juin 2018	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
mozillacert1.pem	18 juin 2018	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
eecertificationcentrerootca	18 juin 2018	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
digicertglobalrootca	18 juin 2018	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
thawteprimaryrootcag3	18 juin 2018	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
thawteprimaryrootcag2	18 juin 2018	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
entrustrootcertificationauthorityec1	18 juin 2018	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
valicertclass2ca	21 avril 2018	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
globalchambersignroot2008	18 juin 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
amazonrootca4	18 juin 2018	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
gd-class2-root.pem	18 juin 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
amazonrootca3	18 juin 2018	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E

Nom	Date	Empreintes digitales SHA1
amazonrootca2	18 juin 2018	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
securitycommunicationrootca	18 juin 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
amazonrootca1	18 juin 2018	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
acraizfnmtrcm	18 juin 2018	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
quovadisrootca3g3	18 juin 2018	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D
certplusrootcag2	18 juin 2018	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:CC:1A
certplusrootcag1	18 juin 2018	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:F7:66
mozillacert71.pem	18 juin 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert63.pem	18 juin 2018	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
mozillacert121.pem	18 juin 2018	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
ttelesecglobalrootclass3ca	21 avril 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
mozillacert55.pem	18 juin 2018	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
mozillacert113.pem	18 juin 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31

Nom	Date	Empreintes digitales SHA1
baltimorecybertrustca	21 avril 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
mozillacert47.pem	18 juin 2018	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
mozillacert105.pem	18 juin 2018	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC
mozillacert39.pem	18 juin 2018	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
usertrustecccertificationauthority	18 juin 2018	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0
mozillacert0.pem	18 juin 2018	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
securitycommunicationevrootca1	18 juin 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
verisignc3g5.pem	18 juin 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
globalsignr3ca	21 avril 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
trustcoreca1	18 juin 2018	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
equifaxsecureglobalebusinessca1	21 avril 2018	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
geotrustuniversalca	18 juin 2018	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
affirmtrustpremiumca	21 avril 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27

Nom	Date	Empreintes digitales SHA1
staatdernederlandenrootcag3	18 juin 2018	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
staatdernederlandenrootcag2	18 juin 2018	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
mozillacert70.pem	18 juin 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
secomevrootca1	21 avril 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
geotrustglobalca	18 juin 2018	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert62.pem	18 juin 2018	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert120.pem	18 juin 2018	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
mozillacert54.pem	18 juin 2018	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
mozillacert112.pem	18 juin 2018	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
mozillacert46.pem	18 juin 2018	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
swisssigngoldcag2	18 juin 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert104.pem	18 juin 2018	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E
mozillacert38.pem	18 juin 2018	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36

Nom	Date	Empreintes digitales SHA1
certplusclass3pprimaryca	21 avril 2018	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
entrustrootcertificationauthorityg2	18 juin 2018	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
godaddyrootg2ca	21 avril 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
cfcaevroot	18 juin 2018	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
verisignc3g4.pem	18 juin 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
geotrustuniversalca2	18 juin 2018	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
starfieldservicesrootg2ca	21 avril 2018	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
digicerthighassuranceevrootca	18 juin 2018	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
entrustnetpremium2048secureserverca	18 juin 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
camerfirmaglobalchambersignroot	18 juin 2018	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
verisignclass3g3ca	21 avril 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
godaddyclass2ca	18 juin 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
mozillacert61.pem	18 juin 2018	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84

Nom	Date	Empreintes digitales SHA1
mozillacert53.pem	18 juin 2018	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
atostrustedroot2011	18 juin 2018	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert111.pem	18 juin 2018	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
staatdernederlandenevrootca	18 juin 2018	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
mozillacert45.pem	18 juin 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert103.pem	18 juin 2018	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert37.pem	18 juin 2018	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert29.pem	18 juin 2018	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
izenpecom	18 juin 2018	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
comodorsacertificationauthority	18 juin 2018	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
mozillacert99.pem	18 juin 2018	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
mozillacert149.pem	18 juin 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
utnuserfirstobjectca	21 avril 2018	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46

Nom	Date	Empreintes digitales SHA1
verisignc3g3.pem	18 juin 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
dstrootcax3	18 juin 2018	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
addtrustexternalroot	18 juin 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
certumtrustednetworkca	18 juin 2018	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
affirmtrustpremiumecc	18 juin 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
starfieldclass2ca	18 juin 2018	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
actalisauthenticationrootca	18 juin 2018	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
verisignclass2g3ca	21 avril 2018	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
isrgrootx1	18 juin 2018	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
godaddyrootcertificateauthorityg2	18 juin 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
mozillacert60.pem	18 juin 2018	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
chunghwaepkirootca	21 avril 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert52.pem	18 juin 2018	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F

Nom	Date	Empreintes digitales SHA1
microseceszignorootca2009	18 juin 2018	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
securesignrootca11	18 juin 2018	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert110.pem	18 juin 2018	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert44.pem	18 juin 2018	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
mozillacert102.pem	18 juin 2018	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
mozillacert36.pem	18 juin 2018	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D
mozillacert28.pem	18 juin 2018	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
baltimorecybertrustroot	18 juin 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
amzninternalrootca	12 décembre 2008	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06
mozillacert98.pem	18 juin 2018	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
mozillacert148.pem	18 juin 2018	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
verisignc3g2.pem	18 juin 2018	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F

Nom	Date	Empreintes digitales SHA1
quovadisrootca2g3	18 juin 2018	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
geotrustprimarycertificatio nauthority	18 juin 2018	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
opentrustrootcag3	18 juin 2018	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8A:A6
opentrustrootcag2	18 juin 2018	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	18 juin 2018	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:D5:7E
verisignclass3ca	21 avril 2018	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
globalsignca	21 avril 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
ttelesecglobalrootclass2ca	21 avril 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
verisignclass1g3ca	21 avril 2018	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
verisignuniversalrootca	21 avril 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
soneraclass2ca	21 avril 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
starfieldservicesrootcertificateauthorityg2	18 juin 2018	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F

Nom	Date	Empreintes digitales SHA1
mozillacert51.pem	18 juin 2018	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
mozillacert43.pem	18 juin 2018	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
mozillacert101.pem	18 juin 2018	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
mozillacert35.pem	18 juin 2018	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
globalsignr2ca	21 avril 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
mozillacert27.pem	18 juin 2018	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
affirmtrustpremium	18 juin 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert19.pem	18 juin 2018	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
mozillacert97.pem	18 juin 2018	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
netlockaranyclassgoldfotanusitvany	18 juin 2018	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
mozillacert89.pem	18 juin 2018	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
verisignroot.pem	18 juin 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
mozillacert147.pem	18 juin 2018	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4

Nom	Date	Empreintes digitales SHA1
aolrootca2	21 avril 2018	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
cia-crt-g3-01-ca	23 novembre 2016	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
aolrootca1	21 avril 2018	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignc3g1.pem	18 juin 2018	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert139.pem	18 juin 2018	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
soneraclass2rootca	18 juin 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
swisssignsilverg2ca	21 avril 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
thawteprimaryrootca	18 juin 2018	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
gdcatrustauthr5root	18 juin 2018	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
trustcenterclass4caii	21 avril 2018	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
usertrustrsacertificationauthority	18 juin 2018	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
digicertassuredidrootg3	18 juin 2018	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89

Nom	Date	Empreintes digitales SHA1
digicertassuredidrootg2	18 juin 2018	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F
mozillacert50.pem	18 juin 2018	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
mozillacert42.pem	18 juin 2018	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
mozillacert100.pem	18 juin 2018	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
mozillacert34.pem	18 juin 2018	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
affirmtrustcommercialca	21 avril 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert26.pem	18 juin 2018	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
globalsigneccrootcar5	18 juin 2018	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
globalsigneccrootcar4	18 juin 2018	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
buypassclass3rootca	18 juin 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert18.pem	18 juin 2018	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
mozillacert96.pem	18 juin 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
verisignc2g6.pem	18 juin 2018	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F

Nom	Date	Empreintes digitales SHA1
secomvalicertclass1ca	21 avril 2018	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert88.pem	18 juin 2018	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
accvraiz1	18 juin 2018	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert146.pem	18 juin 2018	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A
mozillacert138.pem	18 juin 2018	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
verisignclass3g2ca	21 avril 2018	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
dtrustrootclass3ca2ev2009	18 juin 2018	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
xrampglobalca	21 avril 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
mozillacert9.pem	18 juin 2018	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
verisignuniversalrootcertificationauthority	18 juin 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
tubitakkamusmsslkoksertifikasisurum1	18 juin 2018	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
mozillacert41.pem	18 juin 2018	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3

Nom	Date	Empreintes digitales SHA1
mozillacert33.pem	18 juin 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
mozillacert25.pem	18 juin 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
mozillacert17.pem	18 juin 2018	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert95.pem	18 juin 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
affirmtrustpremiumeccca	21 avril 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert87.pem	18 juin 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert145.pem	18 juin 2018	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:3A:04
mozillacert79.pem	18 juin 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert137.pem	18 juin 2018	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A1:D1
digicertassuredidrootca	18 juin 2018	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
addtrustqualifiedca	21 avril 2018	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert129.pem	18 juin 2018	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
verisignclass2g2ca	21 avril 2018	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D

Nom	Date	Empreintes digitales SHA1
baltimorecodesigningca	21 avril 2018	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D
luxtrustglobalroot2	18 juin 2018	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5F:3F
visaecommerceroot	18 juin 2018	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
oistewisekeyglobalrootgbca	18 juin 2018	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
mozillacert8.pem	18 juin 2018	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:6A:0F
comodocertificationauthority	18 juin 2018	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
cia-crt-g3-02-ca	23 novembre 2016	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
verisignc1g6.pem	18 juin 2018	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
trustcenterclass2caii	21 avril 2018	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
quovadisrootca1g3	18 juin 2018	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67
mozillacert40.pem	18 juin 2018	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
cadisigrootr2	18 juin 2018	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71

Nom	Date	Empreintes digitales SHA1
cadisigrootr1	18 juin 2018	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
mozillacert32.pem	18 juin 2018	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C
utndatacorpsgcca	21 avril 2018	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
mozillacert24.pem	18 juin 2018	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
addtrustclass1ca	21 avril 2018	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
mozillacert16.pem	18 juin 2018	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
affirmtrustnetworkingca	21 avril 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert94.pem	18 juin 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
mozillacert86.pem	18 juin 2018	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
mozillacert144.pem	18 juin 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
mozillacert78.pem	18 juin 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert136.pem	18 juin 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert128.pem	18 juin 2018	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D

Nom	Date	Empreintes digitales SHA1
verisignclass1g2ca	21 avril 2018	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
hellenicacademicandresearchinstitutionsrootca2015	18 juin 2018	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
soneraclass1ca	21 avril 2018	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
hellenicacademicandresearchinstitutionsrootca2011	18 juin 2018	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
certumtrustednetworkca2	18 juin 2018	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
equifaxsecureca	21 avril 2018	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
thawteserverca	21 avril 2018	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
mozillacert7.pem	18 juin 2018	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
affirmtrustnetworking	18 juin 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
deprecateditsecca	27 janvier 2012	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
globalsignrootcar3	18 juin 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD

Nom	Date	Empreintes digitales SHA1
globalsignrootcar2	18 juin 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
quovadisrootca	18 juin 2018	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert31.pem	18 juin 2018	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
entrustrootcertificationauthority	18 juin 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert23.pem	18 juin 2018	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
mozillacert15.pem	18 juin 2018	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
verisignc2g3.pem	18 juin 2018	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
mozillacert93.pem	18 juin 2018	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17
mozillacert151.pem	18 juin 2018	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
mozillacert85.pem	18 juin 2018	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
certplusclass2primaryca	18 juin 2018	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
mozillacert143.pem	18 juin 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
mozillacert77.pem	18 juin 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6

Nom	Date	Empreintes digitales SHA1
mozillacert135.pem	18 juin 2018	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert69.pem	18 juin 2018	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
mozillacert127.pem	18 juin 2018	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert119.pem	18 juin 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
geotrustprimarycag3	21 avril 2018	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
identrustpublicsectorrootca1	18 juin 2018	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
geotrustprimarycag2	21 avril 2018	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
trustcorrootcertca2	18 juin 2018	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
mozillacert6.pem	18 juin 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
trustcorrootcertca1	18 juin 2018	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
networksolutionscertificate authority	18 juin 2018	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
twcarootcertificationauthority	18 juin 2018	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48

Nom	Date	Empreintes digitales SHA1
addtrustexternalca	21 avril 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
verisignclass3g5ca	21 avril 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
autoridaddecertificacionfirmaprofesionalcifa62634068	18 juin 2018	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
hellenicacademicandresearchinstitutionseccrootca2015	18 juin 2018	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
verisigntsaca	21 avril 2018	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
utnuserfirsthardwareca	21 avril 2018	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
identrustcommercialrootca1	18 juin 2018	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
dtrustrootclass3ca22009	18 juin 2018	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
epkirootcertificationauthority	18 juin 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert30.pem	18 juin 2018	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
teliasonerarootcav1	18 juin 2018	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37

Nom	Date	Empreintes digitales SHA1
buypassclass3ca	21 avril 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert22.pem	18 juin 2018	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
mozillacert14.pem	18 juin 2018	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
verisignc2g2.pem	18 juin 2018	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
certumca	21 avril 2018	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert92.pem	18 juin 2018	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0
mozillacert150.pem	18 juin 2018	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
mozillacert84.pem	18 juin 2018	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2
ttelesecglobalrootclass3	18 juin 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
globalsignrootca	18 juin 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
ttelesecglobalrootclass2	18 juin 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert142.pem	18 juin 2018	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
mozillacert76.pem	18 juin 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7

Nom	Date	Empreintes digitales SHA1
mozillacert134.pem	18 juin 2018	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
mozillacert68.pem	18 juin 2018	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
etugracertificationauthority	18 juin 2018	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert126.pem	18 juin 2018	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
keynectisrootca	21 avril 2018	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
mozillacert118.pem	18 juin 2018	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45
quovadisrootca3	18 juin 2018	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	18 juin 2018	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert5.pem	18 juin 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
verisignc1g3.pem	18 juin 2018	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
cybertrustglobalroot	18 juin 2018	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
amzninternalinfosecag3	27 février 2015	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
starfieldrootcertificateauthorityg2	18 juin 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E

Nom	Date	Empreintes digitales SHA1
entrust2048ca	21 avril 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
swisssignsilvercag2	18 juin 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
affirmtrustcommercial	18 juin 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
certinomisrootca	18 juin 2018	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:88:A8
xrampglobalcaroot	18 juin 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
secureglobalca	18 juin 2018	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
swisssigngoldg2ca	21 avril 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert21.pem	18 juin 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
mozillacert13.pem	18 juin 2018	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
verisignc2g1.pem	18 juin 2018	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:14:2A
mozillacert91.pem	18 juin 2018	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
oistewisekeyglobalrootgaca	18 juin 2018	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
mozillacert83.pem	18 juin 2018	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:EA:46

Nom	Date	Empreintes digitales SHA1
entrustevca	21 avril 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert141.pem	18 juin 2018	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
mozillacert75.pem	18 juin 2018	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
mozillacert133.pem	18 juin 2018	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
mozillacert67.pem	18 juin 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
mozillacert125.pem	18 juin 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert59.pem	18 juin 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
thawtepremiumserverca	21 avril 2018	E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:66:66
mozillacert117.pem	18 juin 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
utnuserfirstclientauthemailca	21 avril 2018	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
entrustrootcag2	21 avril 2018	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
mozillacert109.pem	18 juin 2018	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
digicerttrustedrootg4	18 juin 2018	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4

Nom	Date	Empreintes digitales SHA1
gdroot-g2.pem	18 juin 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
comodoaaaservicesroot	18 juin 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert4.pem	18 juin 2018	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B
verisignclass3publicprimary certificationauthorityg5	18 juin 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
chambersofcommerceroot2008	18 juin 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
verisignclass3publicprimary certificationauthorityg4	18 juin 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3publicprimary certificationauthorityg3	18 juin 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
thawtepersonalfreemailca	21 avril 2018	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
verisignc1g2.pem	18 juin 2018	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
gtecybertrustglobalca	21 avril 2018	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74

Nom	Date	Empreintes digitales SHA1
`trustcenteruniversalcai`	21 avril 2018	`6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3`
`camerfirmachamberscommerceca`	21 avril 2018	`6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1`
`verisignclass1ca`	21 avril 2018	`CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1`

Journal des modifications du modèle de mappage Resolver

> **Note**
>
> Nous prenons désormais principalement en charge le runtime APPSYNC_JS et sa documentation. Pensez à utiliser le runtime APPSYNC_JS et ses guides ici.

Les modèles de mappage des résolveurs et des fonctions sont versionnés. La version du modèle de mappage (telle que`2018-05-29`) dicte ce qui suit :* La forme attendue de la configuration de demande de source de données fournie par le modèle de demande * Le comportement d'exécution du modèle de mappage de demande et du modèle de mappage de réponse

Les versions sont représentés à l'aide du format AAAA/MM/JJ, une date ultérieure correspondant à une version plus récente. Cette page répertorie les différences entre les versions de modèles de mappage actuellement prises en charge dansAWS AppSync.

Rubriques

- Disponibilité des opérations de sources de données par matrice de version
- Modification de la version sur un modèle de mappage de résolveur d'unité
- Modification de la version sur une fonction
- 2018-05-29
- 2017-02-28

Disponibilité des opérations de sources de données par matrice de version

Opération/Version prise en charge	2017-02-28	2018-05-29
AWS LambdaInvoquer	Oui	Oui
AWS Lambda BatchInvoke	Oui	Oui
Aucune source de données	Oui	Oui
Amazon OpenSearch GET	Oui	Oui
Amazon OpenSearch POST	Oui	Oui
Amazon OpenSearch PUT	Oui	Oui
Amazon OpenSearch SUPPRIMER	Oui	Oui
Amazon OpenSearch GET	Oui	Oui
DynamoDB GetItem	Oui	Oui
DynamoDB Scan	Oui	Oui
DynamoDB Query	Oui	Oui
DynamoDB DeleteItem	Oui	Oui
DynamoDB PutItem	Oui	Oui
DynamoDB BatchGetItem	Non	Oui
DynamoDB BatchPutItem	Non	Oui
DynamoDB BatchDeleteItem	Non	Oui
HTTP	Non	Oui
Amazon RDS	Non	Oui

Remarque : Seule la version 2018-05-29 est actuellement prise en charge dans les fonctions.

Modification de la version sur un modèle de mappage de résolveur d'unité

Pour les résolveurs d'unité, la version est spécifiée dans le corps du modèle de mappage de la requête. Pour mettre à jour la version, il vous suffit de mettre à jour le champ `version` avec la nouvelle version.

Par exemple, pour mettre à jour la version du AWS Lambda modèle :

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

Vous devez mettre à jour le champ de version de `2017-02-28` en `2018-05-29` comme suit :

```
{
    "version": "2018-05-29", ## Note the version
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

Modification de la version sur une fonction

Pour les fonctions, la version est spécifiée dans le champ `functionVersion` sur l'objet de la fonction. Pur mettre à jour la version, il vous suffit de mettre à jour `functionVersion`. Remarque : Actuellement, seul `2018-05-29` est pris en charge pour la fonction.

Voici un exemple de commande CLI pour mettre à jour la version d'une fonction existante :

```
aws appsync update-function \
--api-id REPLACE_WITH_API_ID \
--function-id REPLACE_WITH_FUNCTION_ID \
```

```
--data-source-name "PostTable" \
--function-version "2018-05-29" \
--request-mapping-template "{...}" \
--response-mapping-template "\$util.toJson(\$ctx.result)"
```

Remarque : Il est recommandé d'omettre le champ de version dans le modèle de mappage de requête de la fonction car il ne sera pas respecté. Si vous spécifiez une version dans le modèle de mappage de requête d'une fonction, la valeur de la version sera remplacée par la valeur du champ `functionVersion`.

2018-05-29

Changement de comportement

- Si le résultat de l'appel de la source de données est `null`, le modèle de mappage de réponse est exécuté.
- Si l'appel de la source de données génère une erreur, c'est à vous de la gérer, le résultat de l'évaluation du modèle de mappage de réponse sera toujours placé dans le bloc de réponse GraphQL`data`.

Raisonnement

- Un résultat d'appel `null` a une signification, et dans certains cas d'utilisation d'application, il est possible que nous voulions gérer des résultats `null` de façon personnalisée. Par exemple, une application peut vérifier si un enregistrement existe dans une table Amazon DynamoDB pour réaliser un contrôle d'autorisation. Dans ce cas, un résultat d'appel `null` signifierait que l'utilisateur n'est probablement pas autorisé. L'exécution du modèle de mappage de réponse offre désormais la possibilité de générer une erreur de non autorisation. Ce comportement procure un meilleur contrôle au concepteur d'API.

Selon le modèle de mappage de réponse suivant :

```
$util.toJson($ctx.result)
```

Auparavant avec `2017-02-28`, si `$ctx.result` revenait nul, le modèle de mappage de réponse n'était pas exécuté. Avec `2018-05-29`, on peut désormais traiter ce scénario. Par exemple, on peut choisir de générer une erreur d'autorisation comme suit :

```
# throw an unauthorized error if the result is null
#if ( $util.isNull($ctx.result) )
    $util.unauthorized()
#end
$util.toJson($ctx.result)
```

Remarque : Les erreurs provenant d'une source de données peuvent ne pas être fatales, voire attendues, c'est pourquoi le modèle de mappage de réponse devrait disposer de la possibilité de traiter l'erreur d'appel et de décider de l'ignorer, de la générer de nouveau ou de générer une autre erreur.

Selon le modèle de mappage de réponse suivant :

```
$util.toJson($ctx.result)
```

Auparavant, avec `2017-02-28`, en cas d'erreur d'appel, le modèle de mappage de réponse était évalué et le résultat était placé automatiquement dans le bloc `errors` de la réponse GraphQL. Avec `2018-05-29`, on peut désormais choisir que faire de l'erreur : la générer de nouveau, générer une erreur différente ou ajouter l'erreur tout en renvoyant des données.

Générer de nouveau une erreur d'appel

Dans le modèle de réponse suivant, on génère la même erreur que celle ayant été renvoyée à partir de la source de données.

```
#if ( $ctx.error )
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

En cas d'erreur d'invocation (par exemple, présence de `$ctx.error`) la réponse ressemble à ceci :

```
{
    "data": {
        "getPost": null
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
```

```
            "errorType": "DynamoDB:ConditionalCheckFailedException",
            "message": "Conditional check failed exception...",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

Générer une autre erreur

Dans le modèle de réponse suivant, on génère notre propre erreur personnalisée après avoir traité l'erreur renvoyée par la source de données.

```
#if ( $ctx.error )
    #if ( $ctx.error.type.equals("ConditionalCheckFailedException") )
        ## we choose here to change the type and message of the error for
 ConditionalCheckFailedExceptions
        $util.error("Error while updating the post, try again. Error:
 $ctx.error.message", "UpdateError")
    #else
        $util.error($ctx.error.message, $ctx.error.type)
    #end
#end
$util.toJson($ctx.result)
```

En cas d'erreur d'invocation (par exemple, présence de `$ctx.error`) la réponse ressemble à ceci :

```
{
    "data": {
        "getPost": null
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "UpdateError",
            "message": "Error while updating the post, try again. Error: Conditional
 check failed exception...",
```

```
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

Ajouter une erreur à des données renvoyées

Dans le modèle de réponse suivant, on ajoute la même erreur renvoyée depuis la source de données tout en renvoyant des données dans la réponse. Ce processus est également connu sous le nom de réponse partielle.

```
#if ( $ctx.error )
    $util.appendError($ctx.error.message, $ctx.error.type)
    #set($defaultPost = {id: "1", title: 'default post'})
    $util.toJson($defaultPost)
#else
    $util.toJson($ctx.result)
#end
```

En cas d'erreur d'invocation (par exemple, présence de `$ctx.error`) la réponse ressemble à ceci :

```
{
    "data": {
        "getPost": {
            "id": "1",
            "title: "A post"
        }
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "ConditionalCheckFailedException",
            "message": "Conditional check failed exception...",
            "locations": [
                {
```

```
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

Migration de 2017-02-28 à 2018-05-29

La migration de 2017-02-28 à 2018-05-29 est simple. Modifiez le champ de version sur le modèle de mappage de requête du résolveur ou sur l'objet de la version de la fonction. Cependant, notez que l'exécution de 2018-05-29 est différente de celle de 2017-02-28. Les modifications sont décrites ici.

Préservation du même comportement d'exécution de 2017-02-28 à 2018-05-29

Dans certains cas, il est possible de conserver le même comportement d'exécution que celui de la version 2017-02-28 lorsqu'on exécute le modèle de version 2018-05-29.

Exemple : DynamoDB PutItem

Compte tenu du modèle de demande DynamoDB 2017-02-28 suivant : PutItem

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "attributeValues" : {
        "baz" : ... typed value
    },
    "condition" : {
       ...
    }
}
```

Et le modèle de réponse suivant :

```
$util.toJson($ctx.result)
```

La migration vers 2018-05-29 modifie ces modèles comme suit :

```
{
    "version" : "2018-05-29", ## Note the new 2018-05-29 version
    "operation" : "PutItem",
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "attributeValues" : {
        "baz" : ... typed value
    },
    "condition" : {
       ...
    }
}
```

Et modifie le modèle de réponse de la façon suivante :

```
## If there is a datasource invocation error, we choose to raise the same error
## the field data will be set to null.
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end

## If the data source invocation is null, we return null.
#if($util.isNull($ctx.result))
  #return
#end

$util.toJson($ctx.result)
```

Il est désormais de votre responsabilité de gérer les erreurs. Nous avons choisi de générer la même erreur en utilisant `$util.error()`, ayant été renvoyé depuis DynamoDB. Vous pouvez adapter cet extrait pour convertir votre modèle de mappage en 2018-05-29. Notez que si votre modèle de réponse est différent, vous devrez tenir compte des changements de comportement d'exécution.

Exemple : DynamoDB GetItem

Compte tenu du modèle de demande DynamoDB 2017-02-28 suivant : GetItem

```
{
```

```
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true
}
```

Et le modèle de réponse suivant :

```
## map table attribute postId to field Post.id
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))

$util.toJson($ctx.result)
```

La migration vers 2018-05-29 modifie ces modèles comme suit :

```
{
    "version" : "2018-05-29", ## Note the new 2018-05-29 version
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true
}
```

Et modifie le modèle de réponse de la façon suivante :

```
## If there is a datasource invocation error, we choose to raise the same error
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type)
#end

## If the data source invocation is null, we return null.
#if($util.isNull($ctx.result))
  #return
#end

## map table attribute postId to field Post.id
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))
```

```
$util.toJson($ctx.result)
```

Dans la version 2017-02-28, si l'appel de source de données est `null`, ce qui signifie qu'aucun élément de la table DynamoDB ne correspond à notre clé, le modèle de mappage de réponse ne s'exécute pas. Cela ne pose aucun problème dans la plupart des cas, mais si vous attendiez que `$ctx.result` ne soit pas `null`, vous devez maintenant gérer ce scénario.

2017-02-28

Caractéristiques

- Si le résultat de l'appel de source de données est `null` le modèle de mappage de réponse n'est pas exécuté.
- Si l'appel de la source de données génère une erreur, le modèle de mappage de réponse est exécuté et le résultat évalué est placé dans le bloc `errors.data` de réponse GraphQL.

Référence de type

Cette section est utilisée comme référence pour les types de schéma.

Types de scalaires dansAWS AppSync

Un type d'objet GraphQL a un nom et des champs, et ces champs peuvent avoir des sous-champs. En fin de compte, les champs d'un type d'objet doivent être résolus enscalairetypes, qui représentent les feuilles de la requête. Pour plus d'informations sur les types d'objets et les scalaires, voirSchémas et typessur le site Web de GraphQL.

En plus de l'ensemble de scalaires GraphQL par défaut,AWS AppSyncvous permet également d'utiliserdéfini par le servicescalaires qui commencent parAWSpréfixe.AWS AppSyncne prend pas en charge la création dedéfini par l'utilisateurscalaires (personnalisés). Vous devez utiliser la valeur par défaut ouAWSscalaires.

Vous ne pouvez pas utiliserAWScomme préfixe pour les types d'objets personnalisés.

La section suivante est une référence pour la saisie de schémas.

Scalaires par défaut

GraphQL définit les scalaires par défaut suivants :

Liste des scalaires par défaut

`ID`

Identifiant unique d'un objet. Ce scalaire est sérialisé comme un`String`mais n'est pas censé être lisible par l'homme.

`String`

Une séquence de caractères UTF-8.

`Int`

Une valeur entière comprise entre - (2)^{31}) et 2^{31}-1.

`Float`

Une valeur à virgule flottante IEEE 754.

`Boolean`

Une valeur booléenne, `true` ou `false`.

AWS AppSyncscalaires

AWS AppSyncdéfinit les scalaires suivants :

AWS AppSyncliste de scalaires

`AWSDate`

Une extensionDate ISO 8601chaîne au format`YYYY-MM-DD`.

`AWSTime`

Une extensionHeure ISO 8601chaîne au format`hh:mm:ss.sss`.

`AWSDateTime`

Une extensionDate et heure ISO 8601chaîne au format`YYYY-MM-DDThh:mm:ss.sssZ`.

> **Note**
>
> Le`AWSDate`,`AWSTime`, et`AWSDateTime`les scalaires peuvent éventuellement inclure undécalage du fuseau horaire. Par exemple, les valeurs`1970-01-01Z`,`1970-01-01-07:00`, et`1970-01-01+05:30`sont tous valables pour`AWSDate`. Le décalage de fuseau horaire doit être soit`Z`(UTC) ou un décalage en heures et minutes (et éventuellement en secondes). Par exemple, `±hh:mm:ss`. Le champ des secondes dans le décalage horaire est considéré comme valide même s'il ne fait pas partie de la norme ISO 8601.

`AWSTimestamp`

Une valeur entière représentant le nombre de secondes avant ou après`1970-01-01-T00:00Z`.

`AWSEmail`

Une adresse e-mail au format`local-part@domain-part`tel que défini parRFC 822.

`AWSJSON`

Une chaîne JSON. Toute construction JSON valide est automatiquement analysée et chargée dans le code du résolveur sous forme de cartes, de listes ou de valeurs scalaires plutôt que sous forme de chaînes d'entrée littérales. Les chaînes sans guillemets ou le format JSON non valide entraînent une erreur de validation GraphQL.

`AWSPhone`

Un numéro de téléphone. Cette valeur est stockée sous forme de chaîne. Les numéros de téléphone peuvent contenir des espaces ou des traits d'union pour séparer les groupes de chiffres. Les numéros de téléphone sans code de pays sont supposés être des numéros américains ou nord-américains conformément à laPlan de numérotage nord-américain (NANP).

`AWSURL`

Une URL telle que définie parRFC 1738. Par exemple, `https://www.amazon.com/dp/B000NZW3KC/` ou `mailto:example@example.com`. Les URL doivent contenir un schéma (`http`,`mailto`) et ne peut pas contenir deux barres obliques (`//`) dans la partie du chemin.

`AWSIPAddress`

Une adresse IPv4 ou IPv6 valide. Les adresses IPv4 sont attendues en notation à quatre points (`123.12.34.56`). Les adresses IPv6 sont attendues dans un format non entre crochets et séparés par des deux-points (`1a2b:3c4b::1234:4567`). Vous pouvez inclure un suffixe CIDR facultatif (`123.45.67.89/16`) pour indiquer le masque de sous-réseau.

Exemple d'utilisation du schéma

L'exemple de schéma GraphQL suivant utilise tous les scalaires personnalisés comme « objet » et montre les modèles de demande et de réponse du résolveur pour les opérations de base de type put, get et list. Enfin, l'exemple montre comment vous pouvez l'utiliser lors de l'exécution de requêtes et de mutations.

```
type Mutation {
    putObject(
        email: AWSEmail,
        json: AWSJSON,
        date: AWSDate,
        time: AWSTime,
        datetime: AWSDateTime,
        timestamp: AWSTimestamp,
```

```
        url: AWSURL,
        phoneno: AWSPhone,
        ip: AWSIPAddress
    ): Object
}

type Object {
    id: ID!
    email: AWSEmail
    json: AWSJSON
    date: AWSDate
    time: AWSTime
    datetime: AWSDateTime
    timestamp: AWSTimestamp
    url: AWSURL
    phoneno: AWSPhone
    ip: AWSIPAddress
}

type Query {
    getObject(id: ID!): Object
    listObjects: [Object]
}

schema {
    query: Query
    mutation: Mutation
}
```

Voici à quoi sert un modèle de demande`putObject`pourrait ressembler à. UN`putObject`utilise un`PutItem`opération pour créer ou mettre à jour un élément dans votre table Amazon DynamoDB. Notez que cet extrait de code ne contient pas de table Amazon DynamoDB configurée comme source de données. Ceci n'est utilisé qu'à titre d'exemple :

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

Le modèle de réponse pour`putObject`renvoie les résultats :

```
$util.toJson($ctx.result)
```

Voici à quoi sert un modèle de demande`getObject`pourrait ressembler à. UN`getObject`utilise un`GetItem`opération pour renvoyer un ensemble d'attributs pour l'élément doté de la clé primaire. Notez que cet extrait de code ne contient pas de table Amazon DynamoDB configurée comme source de données. Ceci n'est utilisé qu'à titre d'exemple :

```
{
    "version": "2017-02-28",
    "operation": "GetItem",
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
    }
}
```

Le modèle de réponse pour`getObject`renvoie les résultats :

```
$util.toJson($ctx.result)
```

Voici à quoi sert un modèle de demande`listObjects`pourrait ressembler à. UN`listObjects`utilise un`Scan`opération pour renvoyer un ou plusieurs éléments et attributs. Notez que cet extrait de code ne contient pas de table Amazon DynamoDB configurée comme source de données. Ceci n'est utilisé qu'à titre d'exemple :

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
}
```

Le modèle de réponse pour`listObjects`renvoie les résultats :

```
$util.toJson($ctx.result.items)
```

Voici quelques exemples d'utilisation de ce schéma avec des requêtes GraphQL :

```
mutation CreateObject {
    putObject(email: "example@example.com"
        json: "{\"a\":1, \"b\":3, \"string\": 234}"
```

```
        date: "1970-01-01Z"
        time: "12:00:34."
        datetime: "1930-01-01T16:00:00-07:00"
        timestamp: -123123
        url:"https://amazon.com"
        phoneno: "+1 555 764 4377"
        ip: "127.0.0.1/8"
    ) {
        id
        email
        json
        date
        time
        datetime
        url
        timestamp
        phoneno
        ip
    }
}

query getObject {
    getObject(id:"0d97daf0-48e6-4ffc-8d48-0537e8a843d2"){
        email
        url
        timestamp
        phoneno
        ip
    }
}

query listObjects {
    listObjects {
        json
        date
        time
        datetime
    }
}
```

Interfaces et unions dans GraphQL

Le système de type GraphQL prend en chargeInterfaces. Une interface expose un certain ensemble de champs qu'un type doit inclure pour implémenter l'interface.

Le système de type GraphQL prend également en chargeSyndicats. Les unions sont identiques aux interfaces, si ce n'est qu'elles ne définissent pas un ensemble commun de champs. Les unions sont généralement préférées aux interfaces lorsque les types possibles ne partagent pas une hiérarchie logique.

La section suivante est une référence pour la saisie de schémas.

Exemples d'interface

Nous pourrions représenter un`Event`interface qui représente tout type d'activité ou de rassemblement de personnes. Certains types d'événements possibles sont`Concert`,`Conference`, et`Festival`. Ces types partagent tous des caractéristiques communes : ils ont tous un nom, un lieu où l'événement se déroule, ainsi qu'une date de début et une date de fin. Ces types présentent également des différences ; a`Conference`propose une liste de conférenciers et d'ateliers, tandis qu'un`Concert`met en vedette un groupe de musique.

Dans le langage de définition de schéma (SDL), le`Event`l'interface est définie comme suit :

```
interface Event {
        id: ID!
        name : String!
        startsAt: String
        endsAt: String
        venue: Venue
        minAgeRestriction: Int
}
```

Et chacun des types implémente le`Event`interface comme suit :

```
type Concert implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
```

```
    performingBand: String
}

type Festival implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performers: [String]
}

type Conference implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    speakers: [String]
    workshops: [String]
}
```

Les interfaces sont utiles pour représenter des éléments qui peuvent être de plusieurs types. Par exemple, nous pourrions rechercher tous les événements se déroulant dans un lieu spécifique. Ajoutons un champ `findEventsByVenue` au schéma :

```
schema {
    query: Query
}

type Query {
    # Retrieve Events at a specific Venue
    findEventsAtVenue(venueId: ID!): [Event]
}

type Venue {
    id: ID!
    name: String
    address: String
    maxOccupancy: Int
}
```

```
type Concert implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performingBand: String
}

interface Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
}

type Festival implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performers: [String]
}

type Conference implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    speakers: [String]
    workshops: [String]
}
```

LefindEventsByVenuerenvoie une liste deEvent. Comme les champs d'interface GraphQL sont communs à tous les types d'implémentation, il est possible de sélectionner tous les champs de

l'interface `Event` (`id`, `name`, `startsAt`, `endsAt`, `venue` et `minAgeRestriction`). En outre, vous pouvez accéder aux champs sur n'importe quel type d'implémentation en utilisant des fragments GraphQL, tant que vous en spécifiez le type.

Examinons un exemple de requête GraphQL utilisant l'interface.

```
query {
  findEventsAtVenue(venueId: "Madison Square Garden") {
    id
    name
    minAgeRestriction
    startsAt

    ... on Festival {
      performers
    }

    ... on Concert {
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

La requête précédente génère une liste unique de résultats et le serveur pourrait, par défaut, trier les événements par date de début.

```
{
  "data": {
    "findEventsAtVenue": [
      {
        "id": "Festival-2",
        "name": "Festival 2",
        "minAgeRestriction": 21,
        "startsAt": "2018-10-05T14:48:00.000Z",
        "performers": [
          "The Singers",
          "The Screamers"
        ]
```

```
      },
      {
        "id": "Concert-3",
        "name": "Concert 3",
        "minAgeRestriction": 18,
        "startsAt": "2018-10-07T14:48:00.000Z",
        "performingBand": "The Jumpers"
      },
      {
        "id": "Conference-4",
        "name": "Conference 4",
        "minAgeRestriction": null,
        "startsAt": "2018-10-09T14:48:00.000Z",
        "speakers": [
          "The Storytellers"
        ],
        "workshops": [
          "Writing",
          "Reading"
        ]
      }
    ]
  }
}
```

Les résultats étant renvoyés sous la forme d'un ensemble unique d'événements, l'utilisation d'interfaces pour représenter des caractéristiques communes est très utile pour trier les résultats.

Exemples syndicaux

Comme indiqué précédemment, les syndicats ne définissent pas d'ensembles de domaines communs. Un résultat de recherche peut représenter de nombreux types différents. À l'aide du schéma `Event`, vous pouvez définir une union `SearchResult` comme suit :

```
type Query {
    # Retrieve Events at a specific Venue
    findEventsAtVenue(venueId: ID!): [Event]
    # Search across all content
    search(query: String!): [SearchResult]
}

union SearchResult = Conference | Festival | Concert | Venue
```

Dans ce cas, pour interroger n'importe quel champ de notre`SearchResult`union, vous devez utiliser des fragments :

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
      id
      name
      performers
    }

    ... on Concert {
      id
      name
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

Tapez la résolution dansAWS AppSync

La résolution de type est le mécanisme par lequel le moteur GraphQL identifie une valeur résolue en tant que type d'objet spécifique.

Pour en revenir à l'exemple de recherche syndicale, à condition que notre requête produise des résultats, chaque élément de la liste des résultats doit se présenter comme l'un des types possibles que`SearchResult`défini par l'union (c'est-à-dire`Conference`,`Festival`,`Concert`, ou`Venue`).

Comme la logique permettant d'identifier un événement `Festival` d'un élément `Venue` ou `Conference` dépend des exigences de l'application, le moteur GraphQL doit se voir attribuer un conseil, pour identifier nos types possibles parmi les résultats bruts.

AvecAWS AppSync, cet indice est représenté par un champ méta nommé`__typename`, dont la valeur correspond au nom du type d'objet identifié.`__typename`est obligatoire pour les types de retour qui sont des interfaces ou des unions.

Exemple de résolution de type

Réutilisons le schéma précédent. Vous pouvez suivre en accédant à la console et en ajoutant les éléments suivants sous la page Schéma :

```
schema {
    query: Query
}

type Query {
    # Retrieve Events at a specific Venue
    findEventsAtVenue(venueId: ID!): [Event]
    # Search across all content
    search(query: String!): [SearchResult]
}

union SearchResult = Conference | Festival | Concert | Venue

type Venue {
    id: ID!
    name: String!
    address: String
    maxOccupancy: Int
}

interface Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
}

type Festival implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
```

```
    venue: Venue
    minAgeRestriction: Int
    performers: [String]
}

type Conference implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    speakers: [String]
    workshops: [String]
}

type Concert implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performingBand: String
}
```

Attachons un résolveur au champ `Query.search`. Dans le`Resolvers`section, choisissezJoindre, créez un nouveauSource de donnéesde typeAUCUNE, puis nommez-leStubDataSource. Dans le cadre de cet exemple, nous allons prétendre que nous avons récupéré des résultats d'une source externe, et coder les résultats récupérés dans notre modèle de mappage de requête.

Dans le volet du modèle de mappage de requête, saisissez :

```
{
    "version" : "2018-05-29",
    "payload":
    ## We are effectively mocking our search results for this example
    [
        {
            "id": "Venue-1",
            "name": "Venue 1",
            "address": "2121 7th Ave, Seattle, WA 98121",
            "maxOccupancy": 1000
```

```
        },
        {
            "id": "Festival-2",
            "name": "Festival 2",
            "performers": ["The Singers", "The Screamers"]
        },
        {
            "id": "Concert-3",
            "name": "Concert 3",
            "performingBand": "The Jumpers"
        },
        {
            "id": "Conference-4",
            "name": "Conference 4",
            "speakers": ["The Storytellers"],
            "workshops": ["Writing", "Reading"]
        }
    ]
}
```

Si l'application renvoie le nom du type dans le cadre du`id`champ, la logique de résolution de type doit analyser le`id`champ pour extraire le nom du type, puis ajouter le`__typename`champ correspondant à chacun des résultats. Vous pouvez exécuter cette logique dans le modèle de mappage de réponse comme suit :

> **Note**
>
> Vous pouvez également effectuer cette tâche dans le cadre de votre fonction Lambda, si vous utilisez la source de données Lambda.

```
#foreach ($result in $context.result)
    ## Extract type name from the id field.
    #set( $typeName = $result.id.split("-")[0] )
    #set( $ignore = $result.put("__typename", $typeName))
#end
$util.toJson($context.result)
```

Exécutez la requête suivante :

```
query {
```

```
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
        id
      name
      performers
    }

    ... on Concert {
      id
      name
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

La requête produit les résultats suivants :

```
{
  "data": {
    "search": [
      {
        "id": "Venue-1",
        "name": "Venue 1",
        "address": "2121 7th Ave, Seattle, WA 98121"
      },
      {
        "id": "Festival-2",
        "name": "Festival 2",
        "performers": [
          "The Singers",
          "The Screamers"
        ]
```

```
      },
      {
        "id": "Concert-3",
        "name": "Concert 3",
        "performingBand": "The Jumpers"
      },
      {
        "speakers": [
          "The Storytellers"
        ],
        "workshops": [
          "Writing",
          "Reading"
        ]
      }
    ]
  }
}
```

La logique de résolution de type varie en fonction de l'application. Par exemple, vous pouvez avoir une logique d'identification différente qui recherche l'existence de certains champs ou même une combinaison de champs. En d'autres termes, vous pourriez détecter la présence du champ `performers` pour identifier un événement `Festival` ou la combinaison des champs `speakers` et `workshops` pour identifier un événement `Conference`. En fin de compte, c'est à vous de définir la logique que vous souhaitez utiliser.

Résolution des problèmes et erreurs courantes

Cette section présente certaines erreurs courantes et la manière de les résoudre.

Mappage de clé DynamoDB incorrect

Si votre opération GraphQL renvoie le message d'erreur suivant, c'est peut-être parce que la structure de votre modèle de mappage de demandes ne correspond pas à la structure clé d'Amazon DynamoDB :

```
The provided key element does not match the schema (Service: AmazonDynamoDBv2; Status
 Code: 400; Error Code
```

Par exemple, si une clé de hachage est `"id"` appelée dans votre table DynamoDB et que votre modèle `"PostID"` indique, comme dans l'exemple suivant, cela entraîne l'erreur précédente, car elle ne correspond pas. `"id"` `"PostID"`

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "PostID" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

Résolveur manquant

Si vous exécutez une opération GraphQL, telle qu'une requête, et que vous obtenez une réponse null, cela peut signifier que vous n'avez pas de résolveur configuré.

Par exemple, si vous importez un schéma qui définit un champ `getCustomer(userId: ID!):` et que vous n'avez pas configuré de résolveur pour ce champ, lorsque vous exécutez une requête telle que `getCustomer(userId:"ID123"){...}`, vous obtenez une réponse semblable à la suivante :

```
{
    "data": {
    "getCustomer": null
```

```
    }
}
```

Erreurs de modèle de mappage

Si votre modèle de mappage n'est pas correctement configuré, vous recevrez une réponse GraphQL dont l'élément `errorType` a pour valeur `MappingTemplate`. Le champ `message` doit indiquer où le problème se situe dans votre modèle de mappage.

Par exemple, si vous n'avez pas de champ `operation` dans votre modèle de mappage de demande ou si le nom du champ `operation` est incorrect, vous obtenez une réponse similaire à la suivante :

```
{
    "data": {
        "searchPosts": null
    },
    "errors": [
        {
        "path": [
            "searchPosts"
        ],
        "errorType": "MappingTemplate",
        "locations": [
            {
            "line": 2,
            "column": 3
            }
        ],
        "message": "Value for field '$[operation]' not found."
        }
    ]
}
```

Types de retour incorrects

Le type de retour renvoyé par votre source de données doit correspondre au type défini d'un objet dans votre schéma, sinon, vous risquez de voir une erreur GraphQL similaire à la suivante :

```
"errors": [
    {
```

```
    "path": [
        "posts"
    ],
    "locations": null,
    "message": "Can't resolve value (/posts) : type mismatch error, expected type LIST,
 got OBJECT"
    }
]
```

Par exemple, cela peut se produire avec la définition de requête suivante :

```
type Query {
    posts: [Post]
}
```

Cette définition attend comme réponse une liste (élément LIST) d'objets `[Posts]`. Par exemple, si vous avez une fonction Lambda dans Node.JS avec quelque chose de similaire à ce qui suit :

```
const result = { data: data.Items.map(item => { return item ; }) };
callback(err, result);
```

Vous obtenez une erreur car `result` est un objet. Vous devez soit modifier le rappel de `result.data`, soit modifier votre schéma afin qu'il ne renvoie pas d'élément LIST.

Traitement des demandes non valides

Lorsqu'il AWS AppSync est impossible de traiter et d'envoyer une demande (en raison de données incorrectes telles qu'une syntaxe non valide) au résolveur de champs, la charge utile de la réponse renvoie les données du champ avec les valeurs définies sur `null` et les erreurs pertinentes.

Les traductions sont fournies par des outils de traduction automatique. En cas de conflit entre le contenu d'une traduction et celui de la version originale en anglais, la version anglaise prévaudra.