Le guide des AWS CDK couches

AWS Conseils prescriptifs

AWS Conseils prescriptifs: Le guide des AWS CDK couches

Table of Contents

Le guide des AWS CDK couches

Steven Guggenheimer, Amazon Web Services ()AWS

Décembre 2023 (historique du document)

L'un des principaux concepts sous-jacents Cloud Development Kit AWS (AWS CDK) ressemble beaucoup à celui qui sous-tend le fait de rester au chaud par temps froid. Ce concept s'appelle le layering. Par temps froid, vous enfilez une chemise, une veste et parfois une veste encore plus grande selon la température. Ensuite, si vous rentrez à l'intérieur et que le chauffage brûle, vous pouvez enlever une ou les deux couches de veste pour ne pas avoir trop chaud. Il AWS CDK utilise la superposition pour fournir différents niveaux d'abstraction pour l'utilisation des composants du cloud. La superposition garantit que vous n'aurez jamais à écrire trop de code ou à accéder trop peu aux propriétés des ressources lorsque vous déployez votre infrastructure sous forme de piles de code (IAC).

Si vous n'utilisez pas le AWS CDK, vous devez écrire vos AWS CloudFormationmodèles à la main, c'est-à-dire que vous n'utilisez qu'une seule couche qui vous oblige à écrire bien plus de code que ce qui est habituellement nécessaire. D'un autre côté, s' AWS CDK ils faisaient abstraction de tout ce CloudFormation que vous n'avez généralement pas besoin d'écrire, vous ne seriez pas en mesure de gérer les cas extrêmes.

Pour résoudre ce problème, le AWS CDK provisionnement des ressources est divisé en trois couches distinctes :

- Couche 1 — La CloudFormation couche : couche la plus élémentaire où la CloudFormation ressource et la AWS CDK ressource sont presque identiques.
- Couche 2 — La couche organisée : couche dans laquelle les CloudFormation ressources sont résumées en classes programmatiques qui rationalisent une grande partie de la syntaxe standard CloudFormation . Cette couche constitue la plupart des AWS CDK.
- Couche 3 — La couche de motif : couche la plus abstraite dans laquelle vous pouvez utiliser les éléments de base fournis par les couches 1 et 2 pour personnaliser le code en fonction de votre cas d'utilisation spécifique.

Chaque élément de chaque couche est une instance d'une AWS CDK classe spéciale appelée `Construct` a. Selon AWS la documentation, les constructions sont « les éléments de base des AWS CDK applications. Une construction représente un « composant cloud » et encapsule tout ce

qui est AWS CloudFormation nécessaire pour créer le composant. » Les constructions au sein de ces couches sont appelées constructions L1, L2 et L3 selon la couche à laquelle elles appartiennent. Dans ce guide, nous allons parcourir chaque AWS CDK couche pour découvrir à quoi elles servent et pourquoi elles sont importantes.

Ce guide est destiné aux responsables techniques, aux responsables et aux développeurs qui souhaitent approfondir les concepts fondamentaux qui sous-tendent le AWS CDK travail. AWS CDK C'est un outil populaire, mais il est très courant que les équipes passent à côté d'une grande partie de ce qu'il a à offrir. Lorsque vous commencez à comprendre les concepts décrits dans ce guide, vous pouvez accéder à un tout nouveau monde de possibilités et optimiser les processus de provisionnement des ressources de vos équipes.

Dans ce guide :

Constructions de couche 1

Les constructions L1 sont les éléments constitutifs du AWS CDK et se distinguent facilement des autres constructions par le préfixe. `Cfn` Par exemple, le package Amazon DynamoDB contenu dans AWS CDK le contient `Table` une construction, qui est une construction L2. La construction L1 correspondante est appelée`CfnTable`, et elle représente directement un CloudFormation `Table` DynamoDB. Il est impossible d'utiliser le AWS CDK sans accéder à cette première couche, bien qu'une AWS CDK application n'utilise généralement jamais directement une construction L1. Cependant, dans la majorité des cas, les constructions L2 et L3 que les développeurs ont l'habitude d'utiliser reposent largement sur les constructions L1. Vous pouvez donc considérer les constructions L1 comme le pont entre CloudFormation et le. AWS CDK

Le seul but du AWS CDK est de générer des CloudFormation modèles en utilisant des langages de codage standard. Une fois que vous avez exécuté la commande cdk synth CLI et que les CloudFormation modèles résultants ont été générés, le travail AWS CDK est terminé. La commande cdk deploy n'est là que pour des raisons pratiques, mais ce que vous faites lorsque vous exécutez cette commande se produit entièrement à l'intérieur CloudFormation. La pièce du puzzle qui traduit le AWS CDK code dans un format CloudFormation compréhensible est la construction L1.

Le AWS CDK— CloudFormation cycle de vie des constructions L1

Le processus de création et d'utilisation des constructions L1 comprend les étapes suivantes :

1. Le processus de AWS CDK construction convertit les CloudFormation spécifications en code programmatique sous la forme de constructions L1.
2. Les développeurs écrivent du code qui fait directement ou indirectement référence à des constructions L1 dans le cadre d'une AWS CDK application.
3. Les développeurs exécutent la commande cdk synth pour reconvertir le code programmatique dans le format dicté par les CloudFormation spécifications (modèles).
4. Les développeurs exécutent la commande cdk deploy pour déployer les CloudFormation piles de ces modèles dans des environnements de AWS comptes.

Faisons un petit exercice. Accédez au référentiel AWS CDK open source activé GitHub, sélectionnez un AWS service aléatoire, puis accédez au AWS CDK package correspondant à ce service (situé dans le dossier`packages`,`aws-cdk-lib`,`aws-<servicename>`,`lib`). Pour cet exemple,

choisissons Amazon S3, mais cela fonctionne pour n'importe quel service. Si vous regardez le fichier index.ts principal de ce package, vous verrez une ligne qui se lit comme suit :

```
export * from './s3.generated';
```

Cependant, vous ne verrez le `s3.generated` fichier nulle part dans le répertoire correspondant. Cela est dû au fait que les constructions L1 sont générées automatiquement à partir de la spécification de la CloudFormation ressource pendant le AWS CDK processus de construction. Vous ne le verrez donc `s3.generated` dans le package qu'après avoir exécuté la commande de AWS CDK construction pour le package.

La spécification AWS CloudFormation de la ressource

La spécification AWS CloudFormation des ressources définit l'infrastructure en tant que code (IAC) AWS et détermine comment le code contenu dans les CloudFormation modèles est converti en ressources dans un AWS compte. Cette spécification définit les AWS ressources au format JSON au niveau de chaque région. Chaque ressource reçoit un nom de type de ressource unique qui suit le format`provider::service::resource`. Par exemple, le nom du type de ressource pour un compartiment Amazon S3 serait`AWS::S3::Bucket`, et le nom du type de ressource pour un point d'accès Amazon S3 serait`AWS::S3::AccessPoint`. Ces types de ressources peuvent être affichés dans un CloudFormation modèle en utilisant la syntaxe définie dans la spécification de la AWS CloudFormation ressource. Lorsque le processus de AWS CDK génération s'exécute, chaque type de ressource devient également une construction L1.

Par conséquent, chaque construction L1 est une image miroir programmatique de sa ressource correspondante CloudFormation . Chaque propriété que vous appliqueriez dans un CloudFormation modèle est disponible lorsque vous instanciez une construction L1, et chaque CloudFormation propriété requise est également requise comme argument lorsque vous instanciez la construction L1 correspondante. Le tableau suivant compare un compartiment S3 tel que représenté dans un CloudFormation modèle avec le même compartiment S3 défini comme une construction AWS CDK L1.

CloudFormation modèle

```
"amzns3demobucket": {
    "Type": "AWS::S3::Bucket",
    "Properties": {
```

Construction L1

```
new CfnBucket(this, "amzns3de
mobucket", {
  bucketName: "amzn-s3-demo-bucket",
```

```
      "BucketName": "amzn-s3-demo-
bucket",
      "BucketEncryption": {
        "ServerSideEncryptionConfig
uration": [
          {
            "ServerSideEncrypt
ionByDefault": {
              "SSEAlgorithm": "AES256"
            }
          }
        ]
      },
      "MetricsConfigurations": [
        {
          "Id": "myConfig"
        }
      ],
      "OwnershipControls": {
        "Rules": [
          {
            "ObjectOwnership":
 "BucketOwnerPreferred"
          }
        ]
      },
      "PublicAccessBlockConfigura
tion": {
        "BlockPublicAcls": true,
        "BlockPublicPolicy": true,
        "IgnorePublicAcls": true,
        "RestrictPublicBuckets": true
      },
      "VersioningConfiguration": {
        "Status": "Enabled"
      }
    }
  }
```

```
  bucketEncryption: {
    serverSideEncryptionConfigu
ration: [
      {
        serverSideEncryptionByDefau
lt: {
          sseAlgorithm: "AES256"
        }
      }
    ]
  },
  metricsConfigurations: [
    {
      id: "myConfig"
    }
  ],
  ownershipControls: {
    rules: [
      {
        objectOwnership: "BucketOw
nerPreferred"
      }
    ]
  },
  publicAccessBlockConfiguration: {
    blockPublicAcls: true,
    blockPublicPolicy: true,
    ignorePublicAcls: true,
    restrictPublicBuckets: true
  },
  versioningConfiguration: {
    status: "Enabled"
  }
});
```

Comme vous pouvez le constater, la construction L1 est la manifestation exacte dans le code de la CloudFormation ressource. Il n'y a pas de raccourcis ni de simplifications, de sorte que la quantité de texte standard à écrire est à peu près la même. Cependant, l'un des grands avantages de l'utilisation

du AWS CDK est censé être qu'il permet d'éliminer une grande partie de cette syntaxe standard. CloudFormation Alors, comment cela se produit-il ? C'est là qu'intervient la construction L2.

Constructions de couche 2

Le référentiel AWS CDK open source est écrit principalement en utilisant le langage TypeScriptde programmation et se compose de nombreux packages et modules. La bibliothèque de packages principale, appeléeaws-cdk-lib, est grossièrement divisée en un package par AWS service, bien que ce ne soit pas toujours le cas. Comme indiqué précédemment, les constructions L1 sont automatiquement générées pendant le processus de construction, alors quel est tout ce code que vous voyez lorsque vous regardez dans le référentiel ? Ce sont des constructions L2, qui sont des abstractions des constructions L1.

Les packages contiennent également une collection de TypeScript types, d'énumérations et d'interfaces ainsi que des classes d'assistance qui ajoutent plus de fonctionnalités, mais ces éléments servent tous à des constructions L2. Toutes les constructions L2 appellent leurs constructions L1 correspondantes dans leurs constructeurs lors de l'instanciation, et la construction L1 résultante qui est créée est accessible depuis la couche 2 comme suit :

```
const role = new Bucket(this, "amzn-s3-demo-bucket", {/*...BucketProps*/});
const cfnBucket = role.node.defaultChild;
```

La construction L2 prend les propriétés par défaut, les méthodes pratiques et les autres sucres syntaxiques et les applique à la construction L1. Cela élimine une grande partie de la répétition et de la verbosité nécessaires pour fournir des ressources directement. CloudFormation

Toutes les constructions L2 construisent leurs constructions L1 correspondantes sous le capot. Cependant, les constructions L2 n'étendent pas réellement les constructions L1. Les constructions L1 et L2 héritent toutes deux d'une classe spéciale appelée Construct. Dans la version 1 de AWS CDK la `Construct` classe était intégrée au kit de développement, mais dans la version 2, il s'agit d'un package autonome distinct. C'est ainsi que d'autres packages tels que le Cloud Development Kit for Terraform (CDKTF) peuvent l'inclure en tant que dépendance. Toute classe qui hérite de la `Construct` classe est une construction L1, L2 ou L3. Les constructions L2 étendent directement cette classe alors que les constructions L1 étendent une classe appeléeCfnResource, comme indiqué dans le tableau suivant.

Arbre d'héritage L1	Arbre d'héritage L2
Construction L1	Construction L2
→ classe CfnResource	→ classe Construct

→ classe abstraite CfnRefElement

→→→ classe abstraite CfnElement

→→→ classe Construct

Si les constructions L1 et L2 héritent de la `Construct` classe, pourquoi les constructions L2 n'étendent-elles pas simplement L1 ? Eh bien, les classes situées entre la `Construct` classe et la couche 1 verrouillent la construction L1 en place en tant qu'image miroir de la CloudFormation ressource. Ils contiennent des méthodes abstraites (méthodes que les classes en aval doivent inclure)`_toCloudFormation`, comme celles qui obligent la construction à générer directement CloudFormation la syntaxe. Les constructions L2 ignorent ces classes et étendent directement la `Construct` classe. Cela leur donne la flexibilité d'abstraire une grande partie du code nécessaire aux constructions L1 en les construisant séparément dans leurs constructeurs.

La section précédente présentait une side-by-side comparaison entre un compartiment S3 issu d'un CloudFormation modèle et ce même compartiment S3 rendu sous forme de construction L1. Cette comparaison a montré que les propriétés et la syntaxe sont presque identiques et que la construction L1 n'enregistre que trois ou quatre lignes par rapport à la CloudFormation construction. Comparons maintenant la construction L1 avec la construction L2 pour le même compartiment S3 :

Construction L1 pour le compartiment S3

```
new CfnBucket(this, "amzns3de
mobucket", {
  bucketName: "amzn-s3-demo-bucket",
  bucketEncryption: {
    serverSideEncryptionConfigu
ration: [
      {
        serverSideEncryptionByDefau
lt: {
          sseAlgorithm: "AES256"
        }
      }
    ]
  },
  metricsConfigurations: [
    {
```

Construction L2 pour le compartiment S3

```
new Bucket(this, "amzns3demobucket",
 {
  bucketName: "amzn-s3-demo-bucket",
  encryption: BucketEncryption.S
3_MANAGED,
  metrics: [
    {
      id: "myConfig"
    },
  ],
  objectOwnership: ObjectOwnership.BU
CKET_OWNER_PREFERRED,
  blockPublicAccess: BlockPubl
icAccess.BLOCK_ALL,
  versioned: true
});
```

```
      id: "myConfig"
    }
  ],
  ownershipControls: {
    rules: [
      {
        objectOwnership: "BucketOw
nerPreferred"
      }
    ]
  },
  publicAccessBlockConfiguration: {
    blockPublicAcls: true,
    blockPublicPolicy: true,
    ignorePublicAcls: true,
    restrictPublicBuckets: true
  },
  versioningConfiguration: {
    status: "Enabled"
  }
});
```

Comme vous pouvez le constater, la construction L2 est inférieure à la moitié de la taille de la construction L1. Les constructions L2 utilisent de nombreuses techniques pour réaliser cette consolidation. Certaines de ces techniques s'appliquent à une seule construction L2, mais d'autres peuvent être réutilisées dans plusieurs constructions afin qu'elles soient séparées dans leur propre classe pour être réutilisables. Les constructions L2 consolident CloudFormation la syntaxe de plusieurs manières, comme indiqué dans les sections suivantes.

Propriétés par défaut

Le moyen le plus simple de consolider le code de provisionnement d'une ressource consiste à transformer les paramètres de propriété les plus courants en paramètres par défaut. AWS CDK Il a accès à de puissants langages de programmation mais CloudFormation ne le fait pas. Ces valeurs par défaut sont donc souvent de nature conditionnelle. Il est parfois possible d'éliminer plusieurs lignes de CloudFormation configuration du AWS CDK code, car ces paramètres peuvent être déduits des valeurs d'autres propriétés transmises à la construction.

Structures, types et interfaces

Bien qu'il AWS CDK soit disponible dans plusieurs langages de programmation, il est écrit nativement TypeScript, de sorte que le système de types du langage est utilisé pour définir les types qui composent les constructions L2. L'étude approfondie de ce type de système dépasse le cadre de ce guide ; consultez la TypeScriptdocumentation pour plus de détails. En résumé, a TypeScript `type` décrit le type de données que contient une variable particulière. Il peut s'agir de données de base telles que `a`string, ou de données plus complexes telles que`object`. A TypeScript `interface` est une autre façon d'exprimer le type d' TypeScriptobjet, et a `struct` est un autre nom pour une interface.

TypeScript n'utilise pas le terme structure, mais si vous regardez dans la référence de l'AWS CDK API, vous verrez qu'une structure n'est en fait qu'une autre TypeScript interface dans le code. La référence d'API fait également référence à certaines interfaces en tant qu'interfaces. Si les structures et les interfaces sont identiques, pourquoi la AWS CDK documentation fait-elle une distinction entre elles ?

Ce que l'on AWS CDK appelle des structures sont des interfaces qui représentent n'importe quel objet utilisé par une construction L2. Cela inclut les types d'objets pour les arguments de propriété transmis à la construction L2 lors de l'instanciation, tels que `BucketProps` pour la construction S3 Bucket et pour `TableProps` la construction DynamoDB Table, ainsi que les autres TypeScript interfaces utilisées dans le. AWS CDK En bref, s'il s'agit d'une TypeScript interface dans le AWS CDK et que son nom n'est pas préfixé par la lettre`I`, il l' AWS CDK appelle une structure.

Inversement, il AWS CDK utilise le terme interface pour représenter les éléments de base. Un objet simple devrait être considéré comme une représentation appropriée d'une construction ou d'une classe d'assistance particulière. C'est-à-dire qu'une interface décrit ce que doivent être les propriétés publiques d'une construction L2. Tous les noms AWS CDK d'interface sont les noms de structures existantes ou de classes auxiliaires préfixés par la lettre. `I` Toutes les constructions L2 étendent la `Construct` classe, mais elles implémentent également l'interface correspondante. La construction L2 `Bucket` implémente donc l'`IBucket`interface.

Méthodes statiques

Chaque instance d'une construction L2 est également une instance de son interface correspondante, mais l'inverse n'est pas vrai. Cela est important lorsque vous parcourez une structure pour voir quels types de données sont requis. Si une structure possède une propriété appeléebucket`, qui nécessite le type de données`IBucket, vous pouvez transmettre soit un objet contenant

les propriétés répertoriées dans l'`IBucket`interface, soit une instance d'un `Bucket` L2. L'un ou l'autre fonctionnerait. Toutefois, si cette `bucket` propriété appelait un L2`Bucket`, vous ne pouviez transmettre qu'une `Bucket` instance dans ce champ.

Cette distinction devient très importante lorsque vous importez des ressources préexistantes dans votre pile. Vous pouvez créer une construction L2 pour n'importe quelle ressource native de votre pile, mais si vous devez référencer une ressource créée en dehors de la pile, vous devez utiliser l'interface de cette construction L2. En effet, la création d'une construction L2 crée une nouvelle ressource s'il n'en existe pas déjà une dans cette pile. Les références aux ressources existantes doivent être des objets simples conformes à l'interface de cette construction L2.

Pour faciliter cela dans la pratique, la plupart des constructions L2 sont associées à un ensemble de méthodes statiques qui renvoient l'interface de cette construction L2. Ces méthodes statiques commencent généralement par le mot`from`. Les deux premiers arguments transmis à ces méthodes sont les mêmes `scope` et les `id` arguments sont requis pour une construction L2 standard. Cependant, le troisième argument `props` n'est qu'un petit sous-ensemble de propriétés (ou parfois une seule propriété) qui définit une interface. Pour cette raison, lorsque vous transmettez une construction L2, dans la plupart des cas, seuls les éléments de l'interface sont requis. Ainsi, vous pouvez également utiliser les ressources importées, dans la mesure du possible.

```
// Example of referencing an external S3 bucket
const preExistingBucket = Bucket.fromBucketName(this, "external-bucket", "name-of-
bucket-that-already-exists");
```

Cependant, vous ne devez pas trop vous fier aux interfaces. Vous ne devez importer des ressources et utiliser les interfaces directement que lorsque cela est absolument nécessaire, car les interfaces ne fournissent pas la plupart des propriétés, telles que les méthodes d'assistance, qui rendent une construction L2 si puissante.

Méthodes auxiliaires

Une construction L2 est une classe programmatique plutôt qu'un simple objet. Elle peut donc exposer des méthodes de classe qui vous permettent de manipuler la configuration de vos ressources après l'instanciation. La construction Gestion des identités et des accès AWS (IAM) L2 Role en est un bon exemple. Les extraits suivants montrent deux manières de créer le même rôle IAM à l'aide de la construction L2. `Role`

Sans méthode auxiliaire :

```
const role = new Role(this, "my-iam-role", {
   assumedBy: new FederatedPrincipal('my-identity-provider.com'),
   managedPolicies: [
      ManagedPolicy.fromAwsManagedPolicyName("ReadOnlyAccess")
   ],
   inlinePolicies: {
      lambdaPolicy: new PolicyDocument({
         statements: [
            new PolicyStatement({
               effect: Effect.ALLOW,
               actions: [ 'lambda:UpdateFunctionCode' ],
               resources: [ 'arn:aws:lambda:us-east-1:123456789012:function:my-
function' ]
            })
         ]
      })
   }
});
```

Avec une méthode auxiliaire :

```
const role = new Role(this, "my-iam-role", {
   assumedBy: new FederatedPrincipal('my-identity-provider.com')
});

role.addManagedPolicy(ManagedPolicy.fromAwsManagedPolicyName("ReadOnlyAccess"));
role.attachInlinePolicy(new Policy(this, "lambda-policy", {
   policyName: "lambdaPolicy",
   statements: [
      new PolicyStatement({
         effect: Effect.ALLOW,
         actions: [ 'lambda:UpdateFunctionCode' ],
         resources: [ 'arn:aws:lambda:us-east-1:123456789012:function:my-function' ]
      })
   ]
 }));
```

La possibilité d'utiliser des méthodes d'instance pour manipuler la configuration des ressources après l'instanciation donne aux constructions L2 une flexibilité supplémentaire par rapport à la couche précédente. Les constructions L1 héritent également de certaines méthodes de ressources (telles que`addPropertyOverride`), mais ce n'est qu'à partir de la deuxième couche que vous obtenez des méthodes spécifiquement conçues pour cette ressource et ses propriétés.

Enums

CloudFormation la syntaxe vous oblige souvent à spécifier de nombreux détails afin de provisionner correctement une ressource. Cependant, la majorité des cas d'utilisation ne sont souvent couverts que par une poignée de configurations. La représentation de ces configurations à l'aide d'une série de valeurs énumérées peut réduire considérablement la quantité de code nécessaire.

Par exemple, dans l'exemple de code L2 du compartiment S3 décrit plus haut dans cette section, vous devez utiliser la `bucketEncryption` propriété du CloudFormation modèle pour fournir tous les détails, y compris le nom de l'algorithme de chiffrement à utiliser. AWS CDK Il fournit plutôt l'`BucketEncryption`énumération, qui reprend les cinq formes les plus courantes de chiffrement des compartiments et vous permet d'exprimer chacune d'elles en utilisant des noms de variables uniques.

Qu'en est-il des cas extrêmes qui ne sont pas couverts par les énumérations ? L'un des objectifs d'une construction L2 est de simplifier la tâche de provisionnement d'une ressource de couche 1, de sorte que certains cas extrêmes moins courants risquent de ne pas être pris en charge dans la couche 2. Pour prendre en charge ces cas extrêmes, vous AWS CDK pouvez manipuler les propriétés CloudFormation des ressources sous-jacentes directement à l'aide de la addPropertyOverrideméthode. Pour en savoir plus sur les remplacements de propriétés, consultez la section Meilleures pratiques de ce guide et la section Abstractions et trappes d'échappement de la documentation. AWS CDK

Cours d'assistance

Parfois, une énumération ne peut pas accomplir la logique de programmation nécessaire pour configurer une ressource pour un cas d'utilisation donné. Dans ces situations, il propose AWS CDK souvent un cours d'assistance à la place. Une énumération est un objet simple qui propose une série de paires clé-valeur, tandis qu'une classe auxiliaire offre toutes les fonctionnalités d'une classe. TypeScript Une classe auxiliaire peut toujours agir comme une énumération en exposant des propriétés statiques, mais ces propriétés peuvent alors avoir leurs valeurs définies en interne avec une logique conditionnelle dans le constructeur de classe auxiliaire ou dans une méthode d'assistance.

Ainsi, bien que l'`BucketEncryption`énumération puisse réduire la quantité de code nécessaire pour définir un algorithme de chiffrement sur un compartiment S3, cette même stratégie ne fonctionnerait pas pour définir des durées car les valeurs possibles sont tout simplement trop nombreuses parmi lesquelles choisir. La création d'une énumération pour chaque valeur serait bien

plus difficile qu'elle n'en vaut la peine. Pour cette raison, une classe d'assistance est utilisée pour les paramètres de configuration par défaut d'un compartiment S3 Object Lock, tels que représentés par la ObjectLockRetentionclasse. `ObjectLockRetention`contient deux méthodes statiques : l'une pour le maintien de la conformité et l'autre pour le maintien de la gouvernance. Les deux méthodes utilisent une instance de la classe d'assistance Duration comme argument pour exprimer la durée pendant laquelle le verrou doit être configuré.

La classe d' AWS Lambda assistance Runtime est un autre exemple. À première vue, il peut sembler que les propriétés statiques associées à cette classe puissent être gérées par une énumération. Cependant, chaque valeur de propriété représente une instance de la `Runtime` classe elle-même, de sorte que la logique exécutée dans le constructeur de la classe n'a pas pu être atteinte dans une énumération.

Constructions de la couche 3

Si les constructions L1 effectuent une traduction littérale des CloudFormation ressources en code programmatique et que les constructions L2 remplacent une grande partie de la CloudFormation syntaxe détaillée par des méthodes auxiliaires et une logique personnalisée, à quoi servent les constructions L3 ? La réponse à cette question n'est limitée que par votre imagination. Vous pouvez créer une couche 3 adaptée à n'importe quel cas d'utilisation spécifique. Si votre projet a besoin d'une ressource dotée d'un sous-ensemble spécifique de propriétés, vous pouvez créer une construction L3 réutilisable pour répondre à ce besoin.

Les constructions L3 sont appelées modèles au sein du. AWS CDK Un modèle est un objet qui étend la `Construct` classe dans le AWS CDK (ou étend une classe qui étend la `Construct` classe) pour exécuter une logique abstraite au-delà de la couche 2. Lorsque vous utilisez la AWS CDK CLI pour exécuter cdk init afin de démarrer un nouveau AWS CDK projet, vous devez choisir entre trois types AWS CDK d'applications : `applib`, et. `sample-app`

```
Available templates:
* app: Template for a CDK Application
   └─ cdk init app --language=[csharp|fsharp|go|java|javascript|python|typescript]
* lib: Template for a CDK Construct Library
   └─ cdk init lib --language=typescript
* sample-app: Example CDK Application with some constructs
   └─ cdk init sample-app --language=[csharp|fsharp|go|java|javascript|python|typescript]
```

appet `sample-app` les deux représentent AWS CDK des applications classiques dans lesquelles vous créez et déployez des CloudFormation stacks dans des environnements AWS. Lorsque vous choisissez`lib`, vous choisissez de construire une toute nouvelle construction L3. appet vous `sample-app` permettent de choisir n'importe quelle langue prise AWS CDK en charge, mais vous ne pouvez choisir qu' TypeScript avec`lib`. Cela est dû au fait qu' AWS CDK il est écrit nativement TypeScript et utilise un système open source appelé JSii pour traduire le code original dans les autres langues prises en charge. Lorsque vous choisissez `lib` de lancer votre projet, vous choisissez de créer une extension du AWS CDK.

Toute classe qui étend la `Construct` classe peut être une construction L3, mais les cas d'utilisation les plus courants de la couche 3 sont les interactions avec les ressources, les extensions de ressources et les ressources personnalisées. La plupart des constructions L3 utilisent un ou plusieurs de ces trois cas afin d'étendre AWS CDK les fonctionnalités.

Interactions avec les ressources

Une solution utilise généralement plusieurs services AWS qui fonctionnent ensemble. Par exemple, une CloudFront distribution Amazon utilise souvent un compartiment S3 comme origine et AWS WAF pour se protéger contre les exploits courants. AWS AppSync et Amazon API Gateway utilisent souvent des tables Amazon DynamoDB comme sources de données pour leurs API. Un pipeline utilise AWS CodePipeline souvent Amazon S3 comme source et AWS CodeBuild pour ses étapes de construction. Dans ces cas, il est souvent utile de créer une seule construction L3 qui gère le provisionnement de deux ou plusieurs constructions L2 interconnectées.

Voici un exemple de construction L3 qui fournit une CloudFront distribution avec son origine S3, et AWS WAF à placer devant elle, un enregistrement Amazon Route 53 et un certificat AWS Certificate Manager (ACM) pour ajouter un point de terminaison personnalisé avec chiffrement en transit, le tout dans une construction réutilisable :

```
// Define the properties passed to the L3 construct
export interface CloudFrontWebsiteProps {
     distributionProps: DistributionProps
     bucketProps: BucketProps
     wafProps: CfnWebAclProps
     zone: IHostedZone
}

// Define the L3 construct
export class CloudFrontWebsite extends Construct {
     public distribution: Distribution

     constructor(
        scope: Construct,
        id: string,
        props: CloudFrontWebsiteProps
     ) {
         super(scope, id);

         const certificate = new Certificate(this, "Certificate", {
            domainName: props.zone.zoneName,
            validation: CertificateValidation.fromDns(props.zone)
         });
         const defaultBehavior = {
            origin: new S3Origin(new Bucket(this, "bucket", props.bucketProps))
         }
```

```
          const waf = new CfnWebACL(this, "waf", props.wafProps);
          this.distribution = new Distribution(this, id, {
             ...props.distributionProps,
             defaultBehavior,
             certificate,
             domainNames: [this.domainName],
             webAclId: waf.attrArn,
           });
      }
}
```

Notez qu'Amazon S3 CloudFront, Route 53 et ACM utilisent tous des constructions L2, mais que l'ACL Web (qui définit les règles de gestion des requêtes Web) utilise une construction L1. Cela est dû au fait qu'il s' AWS CDK agit d'un package open source évolutif qui n'est pas complètement complet et qu'il n'existe pas `WebAcl` encore de construction L2. Cependant, n'importe qui peut y contribuer AWS CDK en créant de nouvelles constructions L2. Donc, jusqu'à ce qu'il AWS CDK propose une construction L2 pour`WebAcl`, vous devez utiliser une construction L1. Pour créer un nouveau site Web à l'aide de la construction L3`CloudFrontWebsite`, vous devez utiliser le code suivant :

```
const siteADotCom = new CloudFrontWebsite(stack, "siteA", siteAProps);
const siteBDotCom = new CloudFrontWebsite(stack, "siteB", siteBProps);
const siteCDotCom = new CloudFrontWebsite(stack, "siteC", siteCProps);
```

Dans cet exemple, la construction CloudFront `Distribution` L2 est exposée en tant que propriété publique de la construction L3. Il y aura toujours des cas où vous devrez exposer de telles propriétés L3, si nécessaire. En fait, nous y `Distribution` reviendrons plus tard, dans la section Ressources personnalisées.

AWS CDK Il inclut quelques exemples de modèles d'interaction avec les ressources tels que celui-ci. Outre le `aws-ecs` package qui contient les constructions L2 pour Amazon Elastic Container Service (Amazon ECS), il contient un package appelé aws-ecs-patterns AWS CDK . Ce package contient plusieurs constructions L3 qui combinent Amazon ECS avec des équilibreurs de charge d'application, des équilibreurs de charge réseau et des groupes cibles, tout en proposant différentes versions prédéfinies pour Amazon Elastic Compute Cloud (Amazon EC2) et. AWS FargateÉtant donné que de nombreuses applications sans serveur utilisent Amazon ECS uniquement avec Fargate, ces constructions L3 sont pratiques et permettent aux développeurs d'économiser du temps et de l'argent aux clients.

Extensions de ressources

Certains cas d'utilisation nécessitent que les ressources disposent de paramètres par défaut spécifiques qui ne sont pas natifs de la construction L2. Au niveau de la pile, cela peut être géré en utilisant des aspects, mais un autre moyen pratique de donner de nouvelles valeurs par défaut à une construction L2 consiste à étendre la couche 2. Comme une construction est une classe qui hérite de la `Construct` classe et que les constructions L2 étendent cette classe, vous pouvez également créer une construction L3 en étendant directement une construction L2.

Cela peut être particulièrement utile pour une logique métier personnalisée qui répond aux besoins spécifiques d'un client. Supposons qu'une entreprise dispose d'un référentiel qui stocke l'ensemble de son code de AWS Lambda fonction dans un répertoire unique appelé `src/lambda` et que la plupart des fonctions Lambda réutilisent le même nom d'environnement d'exécution et de gestionnaire à chaque fois. Au lieu de configurer le chemin du code chaque fois que vous configurez une nouvelle fonction Lambda, vous pouvez créer une nouvelle construction L3 :

```
export class MyCompanyLambdaFunction extends Function {
      constructor(
        scope: Construct,
        id: string,
        props: Partial<FunctionProps> = {}
     ) {
         super(scope, id, {
            handler: 'index.handler',
            runtime: Runtime.NODEJS_LATEST,
            code: Code.fromAsset(`src/lambda/${props.functionName || id}`),
            ...props
     });
}
```

Vous pouvez ensuite remplacer la `Function` construction L2 partout dans le référentiel comme suit :

```
new MyCompanyLambdaFunction(this, "MyFunction");
new MyCompanyLambdaFunction(this, "MyOtherFunction");
new MyCompanyLambdaFunction(this, "MyThirdFunction", {
   runtime: Runtime.PYTHON_3_11
});
```

Les valeurs par défaut vous permettent de créer de nouvelles fonctions Lambda sur une seule ligne, et la structure L3 est configurée de telle sorte que vous pouvez toujours remplacer les propriétés par défaut si nécessaire.

L'extension directe des constructions L2 fonctionne mieux lorsque vous souhaitez simplement ajouter de nouvelles valeurs par défaut aux constructions L2 existantes. Si vous avez également besoin d'une autre logique personnalisée, il est préférable d'étendre la `Construct` classe. La raison en est la `super` méthode, qui est appelée dans le constructeur. Dans les classes qui étendent d'autres classes, la `super` méthode est utilisée pour appeler le constructeur de la classe parent, et cela doit être la première chose qui se passe dans votre constructeur. Cela signifie que toute manipulation des arguments transmis ou de toute autre logique personnalisée ne peut avoir lieu qu'après la création de la construction L2 d'origine. Si vous devez exécuter l'une de ces logiques personnalisées avant d'instancier votre construction L2, il est préférable de suivre le modèle décrit précédemment dans la section Interactions avec les ressources.

Ressources personnalisées

Les ressources personnalisées sont une fonctionnalité puissante CloudFormation qui vous permet d'exécuter une logique personnalisée à partir d'une fonction Lambda activée lors du déploiement de la pile. Chaque fois que vous avez besoin de processus de déploiement qui ne sont pas directement pris en charge par CloudFormation, vous pouvez utiliser une ressource personnalisée pour y parvenir. AWS CDK Il propose des classes qui vous permettent également de créer des ressources personnalisées par programmation. En utilisant des ressources personnalisées dans un constructeur L3, vous pouvez créer une construction à partir de presque n'importe quoi.

L'un des avantages d'Amazon CloudFront réside dans ses puissantes capacités de mise en cache globale. Si vous souhaitez réinitialiser manuellement ce cache afin que votre site Web reflète immédiatement les nouvelles modifications apportées à votre origine, vous pouvez utiliser une CloudFront invalidation. Toutefois, les invalidations sont des processus qui s'exécutent sur une CloudFront distribution au lieu d'être des propriétés d'une CloudFront distribution. Ils peuvent être créés et appliqués à une distribution existante à tout moment, de sorte qu'ils ne font pas partie intégrante du processus de provisionnement et de déploiement.

Dans ce scénario, vous souhaiterez peut-être créer et exécuter une invalidation après chaque mise à jour de l'origine d'une distribution. Grâce aux ressources personnalisées, vous pouvez créer une construction L3 qui ressemble à ceci :

```
export interface CloudFrontInvalidationProps {
```

```
     distribution: Distribution
     region?: string
     paths?: string[]
}

export class CloudFrontInvalidation extends Construct {
      constructor(
        scope: Construct,
        id: string,
        props: CloudFrontInvalidationProps
     ) {
          super(scope, id);
          const policy = AwsCustomResourcePolicy.fromSdkCalls({
               resources:AwsCustomResourcePolicy.ANY_RESOURCE
          });
          new AwsCustomResource(scope, `${id}Invalidation`, {
             policy,
             onUpdate: {
                  service: 'CloudFront',
                  action: 'createInvalidation',
                  region: props.region || 'us-east-1',
                  physicalResourceId:
 PhysicalResourceId.fromResponse('Invalidation.Id'),
                  parameters: {
                     DistributionId: props.distribution.distributionId,
                     InvalidationBatch: {
                         Paths: {
                            Quantity: props.paths?.length || 1,
                            Items: props.paths || ['/*']
                         },
                         CallerReference: crypto.randomBytes(5).toString('hex')
                     }
                  }
             }
          }
     }
}
```

En utilisant la distribution que nous avons créée précédemment dans la construction `CloudFrontWebsite` L3, vous pouvez le faire très facilement :

```
new CloudFrontInvalidation(this, 'MyInvalidation', {
       distribution: siteADotCom.distribution
```

```
});
```

Cette construction L3 utilise une construction AWS CDK L3 appelée AwsCustomResourcepour créer une ressource personnalisée qui exécute une logique personnalisée. `AwsCustomResource`est très pratique lorsque vous devez effectuer un seul appel au SDK AWS, car il vous permet de le faire sans avoir à écrire de code Lambda. Si vous avez des exigences plus complexes et souhaitez implémenter votre propre logique, vous pouvez utiliser directement la CustomResourceclasse de base.

Le déploiement de compartiments S3 est un autre bon exemple d' AWS CDK utilisation d'une structure L3 de ressources personnalisées. La fonction Lambda créée par la ressource personnalisée dans le constructeur de cette construction L3 ajoute des fonctionnalités qui CloudFormation ne pourraient pas être gérées autrement : elle ajoute et met à jour des objets dans un compartiment S3. Sans le déploiement du compartiment S3, vous ne pourriez pas placer de contenu dans le compartiment S3 que vous venez de créer dans le cadre de votre stack, ce qui serait très gênant.

Le meilleur exemple de l' AWS CDK élimination de la nécessité d'écrire des tonnes de CloudFormation syntaxe est le suivant : `S3BucketDeployment`

```
new BucketDeployment(this, 'BucketObjects', {
  sources: [Source.asset('./path/to/amzn-s3-demo-bucket')],
  destinationBucket: amzn-s3-demo-bucket
});
```

Comparez cela avec le CloudFormation code que vous devrez écrire pour accomplir la même chose :

```
"lambdapolicyA5E98E09": {
   "Type": "AWS::IAM::Policy",
   "Properties": {
    "PolicyDocument": {
     "Statement": [
      {
       "Action": "lambda:UpdateFunctionCode",
       "Effect": "Allow",
       "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function"
      }
     ],
     "Version": "2012-10-17"
    },
    "PolicyName": "lambdaPolicy",
    "Roles": [
     {
```

```
      "Ref": "myiamroleF09C7974"
     }
    ]
   },
   "Metadata": {
    "aws:cdk:path": "CdkScratchStack/lambda-policy/Resource"
   }
  },
  "BucketObjectsAwsCliLayer8C081206": {
   "Type": "AWS::Lambda::LayerVersion",
   "Properties": {
    "Content": {
     "S3Bucket": {
      "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
     },
     "S3Key": "e2277687077a2abf9ae1af1cc9565e6715e2ebb62f79ec53aa75a1af9298f642.zip"
    },
    "Description": "/opt/awscli/aws"
   },
   "Metadata": {
    "aws:cdk:path": "CdkScratchStack/BucketObjects/AwsCliLayer/Resource",
    "aws:asset:path":
"asset.e2277687077a2abf9ae1af1cc9565e6715e2ebb62f79ec53aa75a1af9298f642.zip",
    "aws:asset:is-bundled": false,
    "aws:asset:property": "Content"
   }
  },
  "BucketObjectsCustomResourceB12E6837": {
   "Type": "Custom::CDKBucketDeployment",
   "Properties": {
    "ServiceToken": {
     "Fn::GetAtt": [
      "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C81C01536",
      "Arn"
     ]
    },
    "SourceBucketNames": [
     {
      "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
     }
    ],
    "SourceObjectKeys": [
     "f888a9d977f0b5bdbc04a1f8f07520ede6e00d4051b9a6a250860a1700924f26.zip"
    ],
```

```
    "DestinationBucketName": {
     "Ref": "amzn-s3-demo-bucket77F80CC0"
    },
    "Prune": true
   },
   "UpdateReplacePolicy": "Delete",
   "DeletionPolicy": "Delete",
   "Metadata": {
    "aws:cdk:path": "CdkScratchStack/BucketObjects/CustomResource/Default"
   }
  },
  "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRole89A01265": {
   "Type": "AWS::IAM::Role",
   "Properties": {
    "AssumeRolePolicyDocument": {
     "Statement": [
      {
       "Action": "sts:AssumeRole",
       "Effect": "Allow",
       "Principal": {
        "Service": "lambda.amazonaws.com"
       }
      }
     ],
     "Version": "2012-10-17"
    },
    "ManagedPolicyArns": [
     {
      "Fn::Join": [
       "",
       [
        "arn:",
        {
         "Ref": "AWS::Partition"
        },
        ":iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
       ]
      ]
     }
    ]
   },
   "Metadata": {
    "aws:cdk:path": "CdkScratchStack/
Custom::CDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C/ServiceRole/Resource"
```

```
   }
  },

"CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRoleDefaultPolicy88902FDF":
{
  "Type": "AWS::IAM::Policy",
  "Properties": {
   "PolicyDocument": {
    "Statement": [
     {
      "Action": [
       "s3:GetBucket*",
       "s3:GetObject*",
       "s3:List*"
      ],
      "Effect": "Allow",
      "Resource": [
       {
        "Fn::Join": [
         "",
         [
          "arn:",
          {
           "Ref": "AWS::Partition"
          },
          ":s3:::",
          {
           "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
          },
          "/*"
         ]
        ]
       },
       {
        "Fn::Join": [
         "",
         [
          "arn:",
          {
           "Ref": "AWS::Partition"
          },
          ":s3:::",
          {
           "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
```

```
          }
         ]
        ]
       }
      ]
     },
     {
      "Action": [
       "s3:Abort*",
       "s3:DeleteObject*",
       "s3:GetBucket*",
       "s3:GetObject*",
       "s3:List*",
       "s3:PutObject",
       "s3:PutObjectLegalHold",
       "s3:PutObjectRetention",
       "s3:PutObjectTagging",
       "s3:PutObjectVersionTagging"
      ],
      "Effect": "Allow",
      "Resource": [
       {
        "Fn::GetAtt": [
         "amzns3demobucket77F80CC0",
         "Arn"
        ]
       },
       {
        "Fn::Join": [
         "",
         [
          {
           "Fn::GetAtt": [
            "amzns3demobucket77F80CC0",
            "Arn"
           ]
          },
          "/*"
         ]
        ]
       }
      ]
     }
    ],
```

```
     "Version": "2012-10-17"
    },
    "PolicyName":
 "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRoleDefaultPolicy88902FDF",
    "Roles": [
     {
      "Ref":
 "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRole89A01265"
     }
    ]
   },
   "Metadata": {
    "aws:cdk:path": "CdkScratchStack/
Custom::CDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C/ServiceRole/DefaultPolicy/
Resource"
   }
  },
  "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C81C01536": {
   "Type": "AWS::Lambda::Function",
   "Properties": {
    "Code": {
     "S3Bucket": {
      "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
     },
     "S3Key": "9eb41a5505d37607ac419321497a4f8c21cf0ee1f9b4a6b29aa04301aea5c7fd.zip"
    },
    "Role": {
     "Fn::GetAtt": [
      "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRole89A01265",
      "Arn"
     ]
    },
    "Environment": {
     "Variables": {
      "AWS_CA_BUNDLE": "/etc/pki/ca-trust/extracted/pem/tls-ca-bundle.pem"
     }
    },
    "Handler": "index.handler",
    "Layers": [
     {
      "Ref": "BucketObjectsAwsCliLayer8C081206"
     }
    ],
    "Runtime": "python3.9",
```

```
    "Timeout": 900
   },
   "DependsOn": [

 "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRoleDefaultPolicy88902FDF",
    "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRole89A01265"
   ],
   "Metadata": {
    "aws:cdk:path": "CdkScratchStack/
Custom::CDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C/Resource",
    "aws:asset:path":
 "asset.9eb41a5505d37607ac419321497a4f8c21cf0ee1f9b4a6b29aa04301aea5c7fd",
    "aws:asset:is-bundled": false,
    "aws:asset:property": "Code"
   }
  }
```

4 lignes contre 241 lignes, c'est une énorme différence ! Et ce n'est qu'un exemple de ce qui est possible lorsque vous utilisez la couche 3 pour personnaliser vos piles.

Bonnes pratiques

Constructions L1

- Vous ne pouvez pas toujours éviter d'utiliser directement les constructions L1, mais vous devez l'éviter autant que possible. Si une construction L2 spécifique ne convient pas à votre scénario limite, vous pouvez explorer ces deux options au lieu d'utiliser directement la construction L1 :
  - Accès `defaultChild` : Si la CloudFormation propriété dont vous avez besoin n'est pas disponible dans une construction L2, vous pouvez accéder à la construction L1 sous-jacente en utilisant. `L2Construct.node.defaultChild` Vous pouvez mettre à jour toutes les propriétés publiques de la construction L1 en y accédant via cette propriété au lieu de vous donner la peine de créer vous-même la construction L1.
  - Utiliser les remplacements de propriétés : et si la propriété que vous souhaitez mettre à jour n'est pas publique ? La trappe d'évacuation ultime qui permet AWS CDK de faire tout ce qu'un CloudFormation modèle peut faire est d'utiliser une méthode disponible dans chaque construction L1 :. addPropertyOverride Vous pouvez manipuler votre pile au niveau du CloudFormation modèle en transmettant le nom et la valeur de la CloudFormation propriété directement à cette méthode.

Constructions L2

- N'oubliez pas de tirer parti des méthodes d'assistance proposées souvent par les constructions L2. Avec la couche 2, vous n'avez pas à transmettre toutes les propriétés lors de l'instanciation. Les méthodes auxiliaires L2 peuvent rendre le provisionnement des ressources exponentiellement plus pratique, en particulier lorsque la logique conditionnelle est requise. L'une des méthodes d'assistance les plus pratiques est dérivée de la classe Grant. Cette classe n'est pas utilisée directement, mais de nombreuses constructions L2 l'utilisent pour fournir des méthodes d'assistance qui facilitent grandement l'implémentation des autorisations. Par exemple, si vous souhaitez autoriser une fonction Lambda L2 à accéder à un compartiment L2 S3, vous pouvez `s3Bucket.grantReadWrite(lambdaFunction)` appeler au lieu de créer un nouveau rôle et une nouvelle politique.

Constructions L3

- Bien que les constructions L3 puissent être très pratiques lorsque vous souhaitez rendre vos piles plus réutilisables et personnalisables, nous vous recommandons de les utiliser avec précaution.

Déterminez le type de construction L3 dont vous avez besoin ou si vous avez vraiment besoin d'une construction L3 :

- Si vous n'interagissez pas directement avec les AWS ressources, il est souvent plus approprié de créer une classe d'assistance plutôt que d'étendre la `Construct` classe. Cela est dû au fait que la `Construct` classe effectue de nombreuses actions par défaut qui ne sont nécessaires que si vous interagissez directement avec AWS des ressources. Donc, si vous n'avez pas besoin que ces actions soient effectuées, il est plus efficace de les éviter.
- Si vous déterminez que la création d'une nouvelle construction L3 est appropriée, dans la plupart des cas, vous souhaiterez étendre directement la `Construct` classe. Étendez les autres constructions L2 uniquement lorsque vous souhaitez mettre à jour les propriétés par défaut de la construction. Si d'autres constructions L2 ou une logique personnalisée sont impliquées, étendez `Construct` directement et instanciez toutes les ressources du constructeur.

FAQ

Ne puis-je pas utiliser les couches AWS CDK sans comprendre les couches ?

Tu peux absolument le faire. Mais comme c'est le cas pour les outils les plus puissants, ils le AWS CDK deviennent au fur et à mesure que vous en savez. L'apprentissage AWS CDK de la façon dont les couches interagissent ouvre un nouveau niveau de compréhension qui permet de simplifier vos déploiements de stack bien au-delà de ce que vous pouvez faire avec de simples connaissances de base AWS CDK .

Puis-je créer des constructions L2 à partir de L1 de la même manière que je crée des constructions L3 à partir de L2 ?

Si une ressource possède déjà une construction L2, nous vous recommandons d'utiliser cette construction et d'effectuer vos personnalisations dans la couche 3. En effet, de nombreuses recherches ont déjà été menées pour déterminer les meilleurs moyens de configurer les constructions L2 existantes pour une ressource particulière. Cependant, il existe plusieurs constructions L1 dont les constructions L2 n'existent pas encore. Dans ces cas, nous vous encourageons à créer vos propres constructions L2 et à les partager avec d'autres en devenant contributeur à la bibliothèque AWS CDK open source. Vous trouverez tout ce dont vous avez besoin pour commencer dans les directives de contribution du AWS CDK.

Quelles AWS ressources n'ont pas encore de constructions L2 officielles ?

Le nombre de AWS ressources qui n'ont pas de construction L2 diminue de jour en jour, mais si vous souhaitez participer à la création d'une construction L2 pour l'une de ces ressources, consultez la AWS CDK référence des API. Consultez la liste des ressources dans le volet de gauche. Les ressources dont le nom est indiqué en exposant 1 n'ont pas de construction L2 officielle.

Puis-je créer une construction L2 ou L3 dans n'importe quelle langue prise en charge ? AWS CDK

Il AWS CDK prend en charge plusieurs langages de programmation TypeScript JavaScript, notamment Python, Java, C# et Go. Vous pouvez créer vos propres constructions L3 en utilisant le AWS CDK code compilé dans le langage approprié. Toutefois, si vous souhaitez contribuer AWS CDK ou créer des AWS CDK constructions natives, vous devez utiliser TypeScript. C' TypeScript est parce que c'est la seule langue native du AWS CDK. Les AWS CDK versions pour les autres langues sont créées à partir du TypeScript code natif à l'aide d'une AWS bibliothèque appelée JSii.

Où puis-je trouver les constructions L3 existantes en dehors du ? AWS CDK

Il y a trop d'emplacements à partager ici, mais vous pouvez trouver la plupart des constructions les plus populaires sur le site Web de AWS Solutions Constructs et dans la AWS CDK section du Construct Hub.

Ressources

- AWS CDK Référence API
- AWS CloudFormation spécification des ressources
- AWS CDK construit de la documentation
- AWS CDK abstractions et trappes d'évacuation
- Tirez parti des structures L2 pour réduire la complexité de votre AWS CDK application (AWS article de blog)
- AWS CloudFormation ressources personnalisées
- AWS Solutions et constructions
- Construire un hub
- AWS CDK Exemples (GitHub référentiel)

Historique du document

Le tableau suivant décrit les modifications importantes apportées à ce guide. Pour être averti des mises à jour à venir, abonnez-vous à un fil RSS.

Modification	Description	Date
Publication initiale	—	4 décembre 2023

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.