개발자 안내서

AWS AppSync

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS AppSync: 개발자 안내서

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 상표 및 트레이드 드레스는 Amazon 외 제품 또는 서비스와 함께 사용되어서는 안되며, 고객에게 혼동을 일으키거나 Amazon 브랜드 이미지를 떨어뜨리고 폄하하는 방식으로 이용할 수 없습니다. Amazon이 소유하지 않은 기타 모든 상표는 Amazon과 제휴 관계이거나 관련이 있거나 후원 관계와 관계없이 해당 소유자의 자산입니다.

Table of Contents

AWS AppSync란 무엇인가요? 1
AWS AppSync 기능 1
AWS AppSync를 처음 사용하십니까? 2
관련 서비스 2
AWS AppSync 요금 2
GraphQL 및 AWS AppSync 아키텍처 3
API란? 4
클라이언트 4
리소스 4
REST란? 4
통합 인터페이스 5
상태 없음 5
계층형 시스템 6
캐싱 가능성 6
RESTful API란? 6
RESTful API는 어떻게 작동하나요? 6
REST 대신 GraphQL을 사용하는 이유는 무엇인가요? 7
GraphQL API의 구성 요소 8
스키마 9
데이터 원본 26
해석기 38
GraphQL의 추가 속성 47
선언적 47
계층적 48
내관적 49
강력한 유형 지정 51
시작하기: 첫 번째 GraphQL API 만들기 52
1단계: 스키마 시작 53
2단계: 콘솔 둘러보기 56
스키마 디자이너 57
데이터 원본 58
쿼리 58
설정 59
3단계: GraphQL 변형을 사용하여 데이터 추가 59

4단계: GraphQL 쿼리를 사용하여 데이터 가져오기 64
보충 섹션 67
통합 67
보충 자료 68
GraphQL API 설계 69
GraphQL API 구조화(빈 API 또는 가져온 API) 69
1단계: 스키마 설계 70
2단계: 데이터 원본 연결 97
3단계: 해석기 구성 108
4단계: API 사용: CDK 예제 160
실시간 데이터 178
GraphQL 스키마 구독 명령 178
구독 인수 사용 181
서버리스 WebSocket으로 구동되는 일반 pub-sub API 생성 184
향상된 구독 필터링 187
연결 구독 취소 197
실시간 웹소켓 클라이언트 빌드 202
병합된 API 217
병합된 API 및 페더레이션 218
병합된 API 충돌 해결 220
스키마 구성 228
권한 부여 모드 구성 228
실행 역할 구성 229
AWS RAM을 사용하여 교차 계정 병합 API 구성 230
병합 232
병합된 API에 대한 추가 지원 233
병합된 API 제한 사항 234
병합된 API 생성 234
RDS 내부 검사 236
내부 검사 기능 사용(콘솔) 237
내부 검사 기능 사용(API) 240
클라이언트 애플리케이션 빌드 244
해석기 튜토리얼(JavaScript) 247
자습서: DynamoDB JavaScript 해석기 247
GraphQL API 생성 248
기본 Post API 정의 248

Amazon DynamoDB 테이블 설정 249
addPost 해석기 설정(Amazon DynamoDB PutItem) 250
getPost 해석기 설정(Amazon DynamoDB GetItem) 253
updatePost 뮤테이션 생성(Amazon DynamoDB UpdateItem) 256
투표 뮤테이션 생성(Amazon DynamoDB UpdateItem) 260
deletePost 해석기 설정(Amazon DynamoDB DeleteItem) 263
allPost 해석기 설정(Amazon DynamoDB 스캔) 269
allPostsByAuthor 해석기 설정(Amazon DynamoDB 쿼리) 273
집합 사용 278
결론 285
자습서: Lambda 해석기 286
Lambda 함수 생성 286
Lambda용 데이터 소스 구성 288
GraphQL 스키마 생성 289
해석기 구성 109
GraphQL API 테스트 290
오류 반환 292
고급 사용 사례: 일괄 처리 295
자습서: 로컬 해석기 304
게시/구독 앱 만들기 304
메시지 전송 및 구독 305
튜토리얼: GraphQL 해석기 결합 306
스키마 예제 306
해석기를 통해 데이터 변경 308
DynamoDB 및 OpenSearch Service 308
자습서: Amazon OpenSearch Service 해석기 311
새 OpenSearch Service 도메인 생성 311
OpenSearch Service 데이터 소스 구성 311
해석기 연결 313
검색 수정 315
OpenSearch Service에 데이터 추가 316
단일 문서 가져오기 317
쿼리 및 뮤테이션 수행 318
모범 사례 319
자습서: DynamoDB 트랜잭션 해석기 319
권한 319

데이터 원본 320
트랜잭션 322
자습서: DynamoDB 배치 해석기 329
단일 테이블 배치 329
다중 테이블 배치 333
오류 처리 341
자습서: HTTP 해석기 347
REST API 만들기 347
GraphQL API 생성 348
GraphQL 스키마 생성 348
HTTP 데이터 소스 구성 349
해석기 구성 139
AWS 서비스 간접 호출 352
자습서: 데이터 API를 사용하는 Aurora PostgreSQL 354
클러스터 생성 354
데이터 API 활성화 355
데이터베이스 및 테이블 생성 356
GraphQL 스키마 생성 356
RDS용 해석기 358
모든 클러스터 삭제 366
해석기 자습서(VTL) 367
자습서: DynamoDB 해석기 368
DynamoDB 테이블 설정 368
GraphQL API 생성 348
기본 Post API 정의 369
DynamoDB 테이블에 대한 데이터 소스 구성 371
addPost 해석기 설정(DynamoDB PutItem) 372
getPost 해석기 설정(DynamoDB GetItem) 376
updatePost 뮤테이션 생성(DynamoDB UpdateItem) 379
updatePost 해석기 수정(DynamoDB UpdateItem) 383
upvotePost 및 downvotePost 뮤테이션 생성(DynamoDB UpdateItem) 389
deletePost 해석기 설정(DynamoDB DeleteItem) 393
allPost 해석기 설정(DynamoDB Scan) 399
allPostsByAuthor 해석기 설정(DynamoDB Query) 404
집합 사용 278
목록 및 맵 사용 417

결론 420
자습서: Lambda 해석기 421
Lambda 함수 생성 421
Lambda용 데이터 소스 구성 423
GraphQL 스키마 생성 348
해석기 구성 139
GraphQL API 테스트 427
오류 반환 429
고급 사용 사례: 일괄 처리 431
자습서: Amazon OpenSearch Service 해석기 441
원클릭 설치 442
새 OpenSearch Service 도메인 생성 442
OpenSearch Service를 위한 데이터 소스 구성 442
해석기 연결 444
검색 수정 446
OpenSearch Service에 데이터 추가 447
단일 문서 가져오기 448
쿼리 및 변형 수행 449
모범 사례 449
자습서: 로컬 해석기 450
페이징 애플리케이션 생성 450
페이지 전송 및 구독 451
튜토리얼: GraphQL 해석기 결합 452
스키마 예제 452
해석기를 통해 데이터 변경 454
DynamoDB 및 OpenSearch Service 454
자습서: DynamoDB 배치 해석기 458
권한 459
데이터 원본 460
단일 테이블 배치 460
다중 테이블 배치 464
오류 처리 471
자습서: DynamoDB 트랜잭션 해석기 477
권한 459
데이터 원본 460
트랜잭션 479

자습서: HTTP 해석기 489
원클릭 설치 442
REST API 만들기 347
GraphQL API 생성 348
GraphQL 스키마 생성 348
HTTP 데이터 원본 구성 349
해석기 구성 139
AWS 서비스 간접 호출 494
자습서: Aurora Serverless 496
클러스터 생성 496
데이터 API 활성화 355
데이터베이스 및 테이블 생성 497
GraphQL 스키마 498
해석기 구성 139
변형 실행 504
쿼리 실행 505
입력 폐기 506
자습서: 파이프라인 해석기 508
원클릭 설치 442
수동 설정 509
GraphQL API 테스트 427
자습서: 델타 동기화 522
원클릭 설치 442
Schema 523
변형 525
동기화 쿼리 526
예 526
구성 및 설정 533
캐싱 및 압축 533
인스턴스 유형 534
캐싱 동작 535
캐시 암호화 536
캐시 제거 536
캐시 항목 제거 536
자격 증명을 기반으로 캐시 항목 제거 538
API 응답 압축 539

사용자 지정 도메인 이름 구성 540
도메인 이름 등록 및 구성 541
AWS AppSync에서 사용자 지정 도메인 생성 541
AWS AppSync의 와일드카드 사용자 지정 도메인 이름 542
충돌 감지 및 동기화 543
버전이 지정된 데이터 원본 543
충돌 감지 및 해결 546
동기화 작업 556
모니터링 및 로깅 556
설정 및 구성 556
CloudWatch 메트릭 558
CloudWatch 로그 567
로그 유형 참조 571
로그 인사이트를 통한 CloudWatch 로그 분석 573
서비스로 OpenSearch 로그를 분석하세요. 575
로그 형식 마이그레이션 575
AWS X-Ray를 사용하여 추적 576
설정 및 구성 556
X-Ray로 API 추적 576
AWS CloudTrail을 사용하여 AWS AppSync API 호출 로깅 579
CloudTrail의 AWS AppSync 정보 579
AWS AppSync 로그 파일 항목 이해 580
AWS AppSync 프라이빗 API 사용 583
AWS AppSync 프라이빗 API 생성 585
AWS AppSync에 대한 인터페이스 엔드포인트 생성 586
고급 예제 587
IAM 정책을 사용하여 퍼블릭 API 생성 제한 591
AWS AppSync를 사용하여 GraphQL 실행 복잡성, 쿼리 깊이 및 내부 검사 구성 592
내부 검사 기능 사용 592
쿼리 깊이 제한 구성 593
해석기 수 제한 구성 595
에서 환경 변수 사용 AWS AppSync 596
환경 변수 구성(콘솔) 597
환경 변수 구성(API) 598
환경 변수 (CFN) 구성 599
환경 변수 및 병합된 API 599

환경 변수 검색 600
인증 및 권한 부여 602
권한 부여 유형 602
API_KEY 권한 부여 603
AWS_LAMBDA 권한 부여 605
SigV4 및 OIDC 토큰 권한 부여 제한 방지 610
AWS_IAM 권한 부여 610
OPENID_CONNECT 권한 부여 612
AMAZON_COGNITO_USER_POOLS 권한 부여 614
추가 권한 부여 모드 사용 615
세분화된 액세스 제어 617
필터링 정보 620
데이터 원본 액세스 621
권한 부여 사용 사례 621
개요 621
데이터 읽기 622
데이터 쓰기 626
퍼블릭 및 프라이빗 레코드 628
실시간 데이터 629
AWS WAF를 사용하여 API 보호 633
AppSync API를 AWS WAF와 통합 634
웹 ACL에 대한 규칙 생성 635
보안 639
데이터 보호 639
작동 중 암호화 640
규정 준수 확인 641
인프라 보안 642
복원력 642
자격 증명 및 액세스 관리 643
고객 643
ID를 통한 인증 644
정책을 사용한 액세스 관리 647
IAM의 AWS AppSync 작동 방식 649
ID 기반 정책 655
문제 해결 667
를 통한 AWS AppSync API 호출 로깅 AWS CloudTrail 669

AWS AppSync 에 대한 정보 CloudTrail ... 669
로그 파일 항목 이해 AWS AppSync ... 670
모범 사례 ... 449
인증 방법 이해 ... 672
HTTP 해석기에 TLS 사용 ... 673
가능한 한 권한이 가장 적은 역할 사용 ... 673
IAM 정책 모범 사례 ... 673
해석기 참조(JavaScript) ... 675
JavaScript 해석기 개요 ... 675
지원되는 런타임 기능 ... 675
유닛 해석기 ... 676
JavaScript 파이프라인 해석기의 구조 ... 676
코드 작성 ... 681
유틸리티 ... 683
번들링, TypeScript, 소스 맵 ... 686
테스트 ... 692
VTL에서 JavaScript로 마이그레이션 ... 694
직접 데이터 소스 액세스와 Lambda 데이터 소스를 통한 프록시 중 선택 ... 697
해석기 컨텍스트 객체 참조 ... 699
`context`에 액세스 ... 699
해석기 및 함수에 대한 JavaScript 런타임 기능 ... 709
지원되는 런타임 기능 ... 710
기본 제공 유틸리티 ... 717
기본 제공 모듈 ... 720
런타임 유틸리티 ... 743
util.time의 시간 도우미 ... 744
util.dynamodb의 DynamoDB 도우미 ... 745
util.http의 HTTP 헬퍼 ... 751
util.transform의 변환 도우미 ... 752
util.str의 문자열 도우미 ... 766
확장 프로그램 ... 766
utils.xml의 XML 도우미 ... 769
DynamoDB용 JavaScript 해석기 함수 참조 ... 771
GetItem ... 771
PutItem ... 773
UpdateItem ... 776

DeleteItem 780
쿼리 783
스캔 787
동기화 791
BatchGetItem 794
BatchDeleteItem 797
BatchPutItem 799
TransactGetItems 801
TransactWriteItems 804
형식 시스템(요청 매핑) 811
형식 시스템(응답 매핑) 815
Filters 819
조건 표현식 821
트랜잭션 조건 표현식 832
프로젝션 835
OpenSearch용 JavaScript 해석기 함수 참조 836
요청 836
응답 837
operation field 837
path field 838
params field 838
변수 전달 840
JavaScript Lambda용 리졸버 함수 참조 841
요청 객체 841
응답 객체 844
Lambda 함수 일괄 처리 응답 845
JavaScript 데이터 소스에 대한 EventBridge 리졸버 함수 참조 845
요청 836
응답 846
PutEvents field 848
None 데이터 소스용 JavaScript 해석기 함수 참조 849
요청 836
Payload 843
응답 846
JavaScript HTTP에 대한 리졸버 함수 참조 850
요청 836

메서드 ... 851
ResourcePath ... 851
Params Field ... 852
응답 ... 846
JavaScript Amazon RDS용 리졸버 함수 레퍼런스 ... 853
SQL 태그가 지정된 템플릿 ... 854
문 생성 ... 855
데이터 불러오기 ... 855
유틸리티 함수 ... 856
캐스팅 ... 864
해석기 매핑 템플릿 참조(VTL) ... 866
해석기 매핑 템플릿 개요 ... 866
유닛 해석기 ... 867
파이프라인 해석기 ... 156
템플릿 예제 ... 871
평가된 맵핑 템플릿 역직렬화 규칙 ... 873
해석기 매핑 템플릿 프로그래밍 안내서 ... 874
설정 ... 875
Variables ... 876
메서드 호출 ... 878
문자열 ... 879
Loops ... 880
배열 ... 881
조건부 확인 ... 882
연산자 ... 883
컨텍스트 ... 885
필터링 ... 885
해석기 매핑 템플릿 컨텍스트 참조 ... 890
$context에 액세스 ... 890
입력 삭제 ... 900
해석기 매핑 템플릿 유틸리티 참조 ... 901
$util의 유틸리티 도우미 ... 902
AWS AppSync 디렉티브 ... 913
$util.time의 시간 도우미 ... 914
$util.list의 목록 도우미 ... 916
$util.map의 맵 도우미 ... 917

$util.dynamodb의 DynamoDB 헬퍼 ... 918
$util.rds의 Amazon RDS 도우미 ... 928
$util.http의 HTTP 도우미 ... 931
$util.xml의 XML 도우미 ... 932
$util.transform의 변환 도우미 ... 934
$util.math의 수학 연산 도우미 ... 948
$util.str의 문자열 도우미 ... 949
확장 프로그램 ... 950
DynamoDB에 대한 해석기 매핑 템플릿 참조 ... 962
GetItem ... 963
PutItem ... 964
UpdateItem ... 968
DeleteItem ... 974
쿼리 ... 976
스캔 ... 981
동기화 ... 985
BatchGetItem ... 988
BatchDeleteItem ... 991
BatchPutItem ... 995
TransactGetItems ... 998
TransactWriteItems ... 1002
형식 시스템(요청 매핑) ... 1010
형식 시스템(응답 매핑) ... 1014
Filters ... 1018
조건 표현식 ... 1020
트랜잭션 조건 표현식 ... 1031
프로젝션 ... 1034
RDS에 대한 해석기 매핑 템플릿 참조 ... 1035
요청 매핑 템플릿 ... 1035
버전 ... 1037
명령문 및 VariableMap ... 1037
VariableTypeHintMap ... 1038
OpenSearch에 대한 해석기 매핑 템플릿 참조입니다. ... 1038
요청 매핑 템플릿 ... 1035
응답 매핑 템플릿 ... 837
operation field ... 837

path field 838
params field 838
변수 전달 840
Lambda용 해석기 매핑 템플릿 참조 1043
요청 매핑 템플릿 1035
응답 매핑 템플릿 837
Lambda 함수 일괄 처리 응답 1048
Direct Lambda 해석기 1049
에 대한 리졸버 매핑 템플릿 참조 EventBridge 1055
요청 매핑 템플릿 1035
응답 매핑 템플릿 837
PutEvents field 848
None 데이터 소스에 대한 해석기 매핑 템플릿 참조 1059
요청 매핑 템플릿 1035
버전 1037
Payload 1047
응답 매핑 템플릿 837
HTTP에 대한 해석기 매핑 템플릿 참조 1061
요청 매핑 템플릿 1035
버전 1037
메서드 1064
ResourcePath 1064
Params Field 838
HTTPS 엔드포인트에 대해 AWS AppSync가 인정하는 인증 기관(CA) 1066
해석기 매핑 템플릿 변경 로그 1129
버전 매트릭스에 따른 데이터 원본 작업 가용성 1129
유닛 해석기 매핑 템플릿에 대한 버전 변경 1131
함수에 대한 버전 변경 1131
2018-05-29 1132
2017-02-28 1139
유형 참조 1140
스칼라 형식 1140
기본 스칼라 1140
AWS AppSync 스칼라 1141
스키마 예제 사용 1142
GraphQL의 인터페이스 및 공용 구조체 1145

인터페이스 예시 1146
공용 구조체 예시 1150
AWS AppSync의 형식 확인 1151
형식 확인의 예 1151
문제 해결 및 일반적인 실수 1157
잘못된 DynamoDB 키 매핑 1157
해석기 누락 1157
매핑 템플릿 오류 1158
잘못된 반환 형식 1158
잘못된 요청 처리 1159
..... mclx

AWS AppSync란 무엇인가요?

AWS AppSync를 사용하면 개발자가 안전한 서버리스 고성능 GraphQL 및 Pub/Sub API를 사용하여 애플리케이션과 서비스를 데이터 및 이벤트에 연결할 수 있습니다. AWS를 사용하여 다음을 할 수 있습니다.

- 단일 GraphQL API 엔드포인트에서 하나 이상의 데이터 소스의 데이터에 액세스합니다.
- 여러 소스 GraphQL API를 하나의 병합된 GraphQL API로 결합합니다.
- 애플리케이션에 실시간 데이터 업데이트를 게시할 수 있습니다.
- 내장된 보안, 모니터링, 로깅 및 추적을 활용하고 지연 시간을 줄이기 위한 선택적 캐싱을 활용합니다.
- API 요청 및 전송된 모든 실시간 메시지에 대해서만 비용을 지불하면 됩니다.

주제

- AWS AppSync 기능
- AWS AppSync를 처음 사용하십니까?
- 관련 서비스
- AWS AppSync 요금

AWS AppSync 기능

- GraphQL로 구동되는 간소화된 데이터 액세스 및 쿼리
- GraphQL 구독 및 게시/구독 채널을 위한 서버리스 WebSocket
- 서버 측 캐싱을 통해 데이터를 고속 인 메모리 캐시에서 사용할 수 있도록 하여 지연 시간을 단축합니다.
- 비즈니스 로직 작성을 위한 JavaScript 및 TypeScript 지원
- API 액세스를 제한하는 비공개 API를 통한 엔터프라이즈 보안 및 AWS WAF 통합
- API 키, IAM, Amazon Cognito, OpenID Connect 공급자, 사용자 지정 로직에 대한 Lambda 권한 부여를 지원하는 권한 부여 제어 기능이 내장되어 있습니다.
- 연동 사용 사례를 지원하기 위한 병합된 API

이러한 각 기능에 대한 자세한 내용은 AWS AppSync 기능을 참조하세요.

AWS AppSync를 처음 사용하십니까?

AWS AppSync를 처음 사용할 경우 먼저 다음 섹션을 읽을 것을 권장합니다.

- GraphQL에 익숙하지 않은 경우 시작하기: 첫 번째 GraphQL API 만들기를 참조하세요.
- GraphQL API를 사용하는 애플리케이션을 빌드하는 경우 클라이언트 애플리케이션 빌드 및 the section called “실시간 데이터”를 참조하세요.
- GraphQL 해석기 정보를 찾고 있는 경우 다음을 참조하세요.

  JavaScript/TypeScript

  - 해석기 튜토리얼(JavaScript)
  - 해석기 참조(JavaScript)

  VTL

  - 해석기 자습서(VTL)
  - 해석기 매핑 템플릿 참조(VTL)
- AWS AppSync 예제 프로젝트, 업데이트 등을 찾고 있다면 AppSync 블로그를 참조하세요.

관련 서비스

웹 또는 모바일 앱을 처음부터 빌드하는 경우 AWS Amplify 사용을 고려해 보세요. Amplify는 AWS AppSync 및 기타 AWS 서비스를 활용하여 더 적은 작업으로 더 강력하고 강력한 웹 및 모바일 앱을 빌드할 수 있도록 지원합니다.

AWS AppSync 요금

AWS AppSync는 수백만 건의 요청 및 업데이트를 기준으로 가격이 책정됩니다. 캐싱에는 추가 요금이 부과됩니다. 자세한 내용은 AWS AppSync 요금을 참조하세요.

다음 목록은 일반 AWS AppSync 요금의 예외 항목입니다.

- AWS AppSync의 API 캐싱은 AWS 프리 티어에서 제공되지 않습니다.
- 요청은 권한 부여 및 인증이 실패한 경우 요금이 청구되지 않습니다.
- API 키를 요구하는 호출 메서드는 API 키 누락 또는 무효 시에 요금이 청구되지 않습니다.

GraphQL 및 AWS AppSync 아키텍처

> **Note**
>
> 이 안내서에서는 사용자가 REST 아키텍처 스타일에 대한 실무 지식이 있다고 가정합니다. GraphQL 및 AWS AppSync로 작업하기 전에 이 주제와 다른 프런트엔드 주제를 검토하는 것이 좋습니다.

GraphQL은 API를 위한 쿼리 및 조작 언어입니다. GraphQL은 데이터 요구 사항 및 상호 작용을 설명하는 유연하고 직관적인 구문을 제공합니다. 이를 통해 개발자는 필요한 것이 무엇인지 정확히 물어보고 예측 가능한 결과를 얻을 수 있습니다. 또한 단일 요청으로 여러 소스에 액세스할 수 있어 네트워크 호출 수와 대역폭 요구 사항이 감소하므로 애플리케이션이 소비하는 배터리 수명과 CPU 주기가 절약됩니다.

변형을 통해 데이터 업데이트가 간단하게 이루어지므로 개발자는 데이터가 변경되어야 하는 방식을 설명할 수 있습니다. 또한 GraphQL은 구독을 통해 실시간 솔루션을 빠르게 설정할 수 있도록 합니다. 이러한 모든 기능이 강력한 개발자 도구와 결합된 GraphQL은 애플리케이션 데이터 관리에 필수입니다.

GraphQL은 REST의 대안입니다. RESTful 아키텍처는 현재 클라이언트-서버 통신에 가장 많이 사용되는 솔루션 중 하나입니다. 이 아키텍처는 URL을 통해 리소스(데이터)가 노출된다는 개념에 중점을 둡니다. 이러한 URL을 사용하여 `GET`, `POST`, `DELETE` 등의 HTTP 메서드 형식으로 CRUD(생성, 읽기, 업데이트, 삭제) 작업을 통해 데이터에 액세스하고 데이터를 조작할 수 있습니다. REST의 장점은 학습과 구현이 비교적 간단하다는 점입니다. RESTful API를 빠르게 설정하여 다양한 서비스를 호출할 수 있습니다.

하지만 기술은 점점 더 복잡해지고 있습니다. 애플리케이션, 도구, 서비스가 전 세계 사용자를 대상으로 확장되기 시작하면서 빠르고 확장 가능한 아키텍처의 필요성이 무엇보다 중요해졌습니다. REST는 확장성이 요구되는 작업을 처리할 때 많은 단점이 있습니다. 이 사용 사례에서 그 예를 확인해 보세요.

다음 섹션에서는 RESTful API와 관련된 몇 가지 개념을 검토합니다. 그런 다음 GraphQL과 작동 방식을 알아봅니다.

GraphQL에 대한 추가 정보와 AWS로 마이그레이션할 때의 이점을 자세히 알아보려면 GraphQL 구현을 위한 결정 안내서를 참조하세요.

주제

- API란?
- REST란?
- REST 대신 GraphQL을 사용하는 이유는 무엇인가요?
- GraphQL API의 구성 요소
- GraphQL의 추가 속성

API란?

애플리케이션 프로그래밍 인터페이스(API)는 다른 소프트웨어 시스템과 통신하기 위해 따라야 하는 규칙을 정의합니다. 개발자는 API를 공개하거나 생성하여 다른 애플리케이션이 프로그래밍 방식으로 애플리케이션과 통신할 수 있도록 합니다. 예를 들어, 근무 시간 기록 애플리케이션은 직원의 성명과 날짜 범위를 요구하는 API를 공개합니다. 이 정보를 수신하면 내부적으로 직원의 근무 시간 기록을 처리하고 해당 날짜 범위에서 근무한 시간을 반환합니다.

웹 API는 웹상의 클라이언트와 리소스 사이의 게이트웨이라고 생각할 수 있습니다.

클라이언트

클라이언트는 웹에서 정보에 액세스하려는 사용자입니다. 클라이언트는 API를 사용하는 사람 또는 소프트웨어 시스템일 수 있습니다. 예를 들어 개발자는 날씨 시스템의 날씨 데이터에 액세스하는 프로그램을 작성할 수 있습니다. 또는 사용자가 날씨 웹 사이트를 직접 방문할 때 브라우저에서 동일한 데이터에 액세스할 수 있습니다.

리소스

리소스는 다양한 애플리케이션이 클라이언트에게 제공하는 정보입니다. 리소스는 이미지, 동영상, 텍스트, 숫자 또는 모든 유형의 데이터일 수 있습니다. 클라이언트에 리소스를 제공하는 시스템을 서버라고도 합니다. 조직은 API를 사용하여 리소스를 공유하고 웹 서비스를 제공하는 동시에 보안, 제어 및 인증을 유지 관리합니다. 또한 API는 특정 내부 리소스에 액세스할 수 있는 클라이언트를 결정하는 데 도움이 됩니다.

REST란?

개괄적으로 보자면 REST(표현형 상태 전송)는 API의 작동 방식에 대한 조건을 강제하는 소프트웨어 아키텍처입니다. REST는 원래 인터넷과 같은 복잡한 네트워크에서의 통신을 관리하기 위한 지침으로 만들어졌습니다. REST 기반 아키텍처를 사용하면 높은 성능의 신뢰할 수 있는 통신을 대규모로 지원

할 수 있습니다. 손쉽게 구현하고 수정할 수 있어 모든 API 시스템에 가시성과 플랫폼 간 이동성을 제공합니다.

API 개발자는 여러 아키텍처를 사용하여 API를 설계할 수 있습니다. REST 아키텍처 스타일을 따르는 API를 REST API라고 합니다. REST 아키텍처를 구현하는 웹 서비스는 RESTful 웹 서비스라고 합니다. RESTful API라는 용어는 일반적으로 RESTful 웹 API를 나타냅니다. 그러나 REST API와 RESTful API라는 용어를 같은 의미로 사용할 수 있습니다.

다음은 REST 아키텍처 스타일의 몇 가지 원칙입니다.

통합 인터페이스

통합 인터페이스는 모든 RESTful 웹 서비스 설계의 기본입니다. 이는 서버가 정보를 표준 형식으로 전송함을 나타냅니다. 형식이 지정된 리소스를 REST에서는 표현이라고 합니다. 이 형식은 서버 애플리케이션의 리소스 내부 표현과 다를 수 있습니다. 예를 들어, 서버는 데이터를 텍스트로 저장하지만 HTML 표현 형식으로 전송할 수 있습니다.

통합 인터페이스에는 다음과 같은 네 가지 아키텍처 제약이 따릅니다.

1. 요청은 리소스를 식별해야 합니다. 이 작업은 통합 리소스 식별자를 사용하여 수행됩니다.
2. 클라이언트는 필요한 경우 리소스를 수정하거나 삭제할 수 있는 충분한 정보를 리소스 표현에 보유하고 있습니다. 서버는 리소스를 자세히 설명하는 메타데이터를 전송하여 이 조건을 충족합니다.
3. 클라이언트는 표현을 추가로 처리하는 방법에 대한 정보를 수신합니다. 서버는 클라이언트가 정보를 가장 잘 사용할 수 있는 방법에 대한 메타데이터가 포함된 자체 설명 메시지를 전송하여 이 작업을 수행합니다.
4. 클라이언트는 작업을 완료하는 데 필요한 기타 모든 관련 리소스에 대한 정보를 수신합니다. 서버는 클라이언트가 더 많은 리소스를 동적으로 검색할 수 있도록 표현에 하이퍼링크를 전송하여 이 작업을 수행합니다.

상태 없음

REST 아키텍처에서 상태 없음이란 서버가 이전의 모든 요청과 독립적으로 모든 클라이언트 요청을 완료하는 통신 방법을 말합니다. 클라이언트는 어떤 순서로든 리소스를 요청할 수 있으며, 모든 요청은 상태 없음으로 지정되거나 다른 요청으로부터 격리됩니다. 이 REST API 설계 제약 조건은 서버가 매번 요청을 완전히 이해하고 이행할 수 있음을 의미합니다.

계층형 시스템

계층형 시스템 아키텍처에서 클라이언트는 클라이언트와 서버 사이의 다른 승인된 중개자에 연결할 수 있으며, 계속해서 서버로부터 응답을 수신합니다. 또한 서버는 요청을 다른 서버로 전달할 수 있습니다. 보안, 애플리케이션, 비즈니스 로직 등 다수의 계층이 있는 여러 서버에서 실행되도록 RESTful 웹 서비스를 설계하고 클라이언트 요청을 이행하도록 할 수 있습니다. 이러한 계층은 클라이언트에게는 공개되지 않은 상태로 유지됩니다.

캐싱 가능성

RESTful 웹 서비스는 캐싱을 지원합니다. 캐싱은 서버 응답 시간을 개선하기 위해 클라이언트나 중개자에 일부 응답을 저장하는 프로세스입니다. 예를 들어 모든 페이지에 공통된 머리글과 바닥글 이미지가 있는 웹 사이트를 방문한다고 가정해 보겠습니다. 새 웹 사이트 페이지를 방문할 때마다 서버는 동일한 이미지를 다시 전송해야 합니다. 이를 방지하기 위해 클라이언트는 첫 번째 응답 후 이러한 이미지를 캐시하거나 저장한 다음 캐시에서 직접 이미지를 사용합니다. RESTful 웹 서비스는 스스로를 캐싱 가능 또는 캐싱 불가능으로 정의하는 API 응답을 사용하여 캐싱을 제어합니다.

RESTful API란?

RESTful API는 두 컴퓨터 시스템이 인터넷을 통해 정보를 안전하게 교환하는 데 사용하는 인터페이스입니다. 대부분의 비즈니스 애플리케이션은 다양한 작업을 수행하기 위해 다른 내부 및 타사 애플리케이션과 통신해야 합니다. 예를 들어 월별 급여명세서를 생성하려면 내부 계정 시스템이 고객의 은행 시스템과 데이터를 공유하여 청구서 발행을 자동화하고 내부 근무 시간 기록 애플리케이션과 통신해야 합니다. RESTful API는 안전하고 신뢰할 수 있으며 효율적인 소프트웨어 통신 표준을 따르므로 이러한 정보 교환을 지원합니다.

RESTful API는 어떻게 작동하나요?

RESTful API의 기본 기능은 인터넷 브라우징과 동일합니다. 클라이언트는 리소스가 필요할 때 API를 사용하여 서버에 접속합니다. API 개발자는 클라이언트가 REST API를 사용하는 방법을 서버 애플리케이션 API 설명서를 통해 설명합니다. 모든 REST API 호출의 일반적인 단계는 다음과 같습니다.

1. 클라이언트가 서버에 요청을 보냅니다. 클라이언트는 API 설명서에 따라 서버가 이해할 수 있는 방식으로 요청 형식을 지정합니다.
2. 서버는 클라이언트를 인증하고 클라이언트가 해당 요청을 보낼 권한이 있는지 확인합니다.
3. 서버는 요청을 수신하고 내부적으로 처리합니다.
4. 서버는 클라이언트에게 응답을 반환합니다. 응답에는 요청이 성공적이었는지 여부를 클라이언트에게 알려주는 정보가 포함됩니다. 응답에는 클라이언트가 요청한 모든 정보도 포함됩니다.

REST API 요청 및 응답 세부 정보는 API 개발자가 API를 설계하는 방식에 따라 약간 다릅니다.

REST 대신 GraphQL을 사용하는 이유는 무엇인가요?

REST는 웹 API의 기초 아키텍처 스타일 중 하나입니다. 그러나 세계가 점점 더 상호 연결됨에 따라 강력하고 확장 가능한 애플리케이션을 개발해야 할 필요성이 더욱 커지고 있습니다. REST는 현재 웹 API 빌드를 위한 업계 표준이지만, RESTful 구현에는 몇 가지 반복되는 단점이 있습니다.

1. 데이터 요청: RESTful API를 사용하면 일반적으로 엔드포인트를 통해 필요한 데이터를 요청합니다. 문제는 데이터가 깔끔하게 패키징되지 않을 때 발생합니다. 필요한 데이터는 여러 추상화 계층 뒤에 있을 수 있으며, 데이터를 가져올 수 있는 유일한 방법은 여러 엔드포인트를 사용하는 것입니다. 즉, 모든 데이터를 추출하기 위해 여러 요청을 수행해야 합니다.
2. 과다 가져오기 및 과소 가져오기: 여러 요청을 수행해야 하는 문제점 외에도 각 엔드포인트의 데이터가 엄격하게 정의되므로 기술적으로 원하지 않더라도 해당 API에 정의된 모든 데이터를 반환하게 됩니다.

   이로 인해 요청이 필요 이상의 데이터를 반환하는 과다 가져오기가 발생할 수 있습니다. 예를 들어, 회사 직원 데이터를 요청하면서 특정 부서의 직원 이름을 알고 싶다고 가정해 보겠습니다. 데이터를 반환하는 엔드포인트에는 이름이 포함되지만, 직책이나 생년월일과 같은 다른 데이터도 포함될 수 있습니다. API가 고정되어 있으므로 이름만 요청할 수는 없으며 나머지 데이터도 함께 제공됩니다.

   충분한 데이터가 반환되지 않는 반대의 상황을 과소 가져오기라고 합니다. 요청된 데이터를 모두 가져오려면 서비스에 여러 번 요청해야 할 수 있습니다. 데이터가 구조화된 방식에 따라 비효율적인 쿼리가 발생하여 우려스러운 n+1 문제와 같은 이슈로 이어질 수 있습니다.
3. 느린 개발 반복: 많은 개발자가 애플리케이션의 흐름에 맞게 RESTful API를 조정합니다. 하지만 애플리케이션이 성장함에 따라 프런트엔드와 백엔드 모두에 광범위한 변화가 필요할 수 있습니다. 결과적으로 API가 더 이상 효율적이거나 영향력 있는 방식으로 데이터의 형태에 맞추지 않게 될 수 있습니다. 이렇게 되면 API 수정이 필요하기 때문에 제품 반복 속도가 느려집니다.
4. 규모에 따른 성능: 이러한 복합적인 문제로 인해 많은 영역에서 확장성에 영향을 받게 됩니다. 요청에서 반환되는 데이터가 너무 많거나 너무 적어(이 경우 요청 수가 늘어남) 애플리케이션 측 성능에 영향을 미칠 수 있습니다. 두 상황 모두 네트워크에 불필요한 부담을 일으켜 성능이 저하됩니다. 개발자 측에서는 API가 고정되어 요청하는 데이터에 더 이상 맞지 않기 때문에 개발 속도가 느려질 수 있습니다.

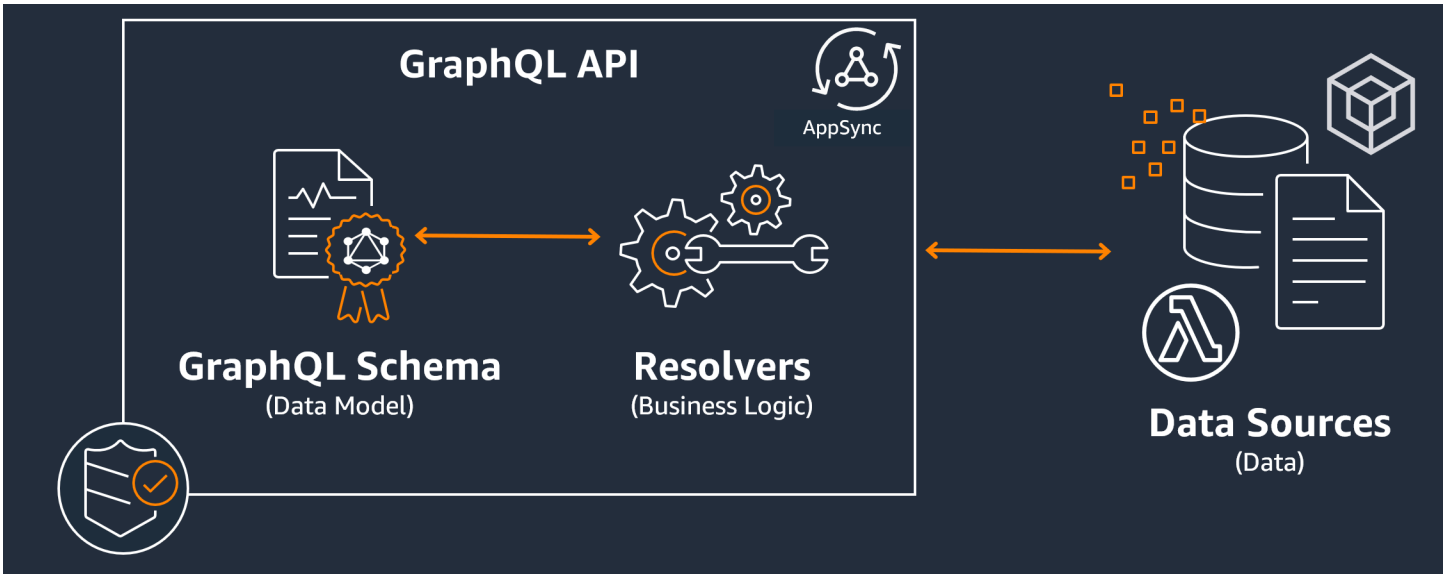
GraphQL의 장점은 REST의 단점을 극복합니다. GraphQL이 개발자에게 제공하는 몇 가지 주요 솔루션은 다음과 같습니다.

1. 단일 엔드포인트: GraphQL은 단일 엔드포인트를 사용하여 데이터를 쿼리합니다. 데이터의 형태에 맞춰 여러 API를 빌드할 필요가 없습니다. 그 결과 네트워크를 통해 전달되는 요청 수가 줄어듭니다.
2. 가져오기: GraphQL은 필요한 데이터를 간단히 정의하여 과다 가져오기 및 과소 가져오기라는 고질적인 문제를 해결합니다. GraphQL을 사용하면 요구 사항에 맞게 데이터를 구성하여 요청한 정보만 수신할 수 있습니다.
3. 추상화: GraphQL API에는 언어에 구애받지 않는 표준을 사용하여 데이터를 설명하는 몇 가지 구성 요소와 시스템이 포함되어 있습니다. 즉, 데이터의 형태와 구조가 표준화되어 있기 때문에 프런트엔드와 백엔드 모두 네트워크를 통해 데이터가 전송되는 방식을 알 수 있습니다. 따라서 양쪽 모두의 개발자가 주변 시스템이 아닌 GraphQL의 시스템으로 작업할 수 있습니다.
4. 신속한 반복: 데이터의 표준화로 인해 개발의 한쪽 엔드에서 변경한 사항이 다른 쪽 엔드에서는 필요하지 않을 수도 있습니다. 예를 들어, GraphQL을 사용하면 데이터 사양을 쉽게 수정할 수 있으므로 프런트엔드 프레젠테이션 변경으로 인해 백엔드가 광범위하게 변경되지 않을 수 있습니다. 애플리케이션이 성장함에 따라 요구 사항에 맞게 데이터 형태를 간단히 정의하거나 수정할 수 있습니다. 따라서 개발 작업이 줄어들 수 있습니다.

이는 GraphQL의 이점 중 일부에 불과합니다. 다음 몇 개의 섹션에서는 GraphQL이 구조화되는 방식과 GraphQL을 REST의 고유한 대안으로 만드는 속성에 대해 알아봅니다.

GraphQL API의 구성 요소

표준 GraphQL API는 쿼리될 데이터의 형태를 처리하는 단일 스키마로 구성됩니다. 스키마는 데이터베이스 또는 Lambda 함수와 같은 하나 이상의 데이터 원본에 연결됩니다. 둘 사이에는 요청에 대한 비즈니스 로직을 처리하는 하나 이상의 해석기가 있습니다. 각 구성 요소는 GraphQL 구현에서 중요한 역할을 합니다. 다음 섹션에서는 이 세 가지 구성 요소와 GraphQL 서비스에서 수행하는 역할을 소개합니다.



주제

- 스키마
- 데이터 원본
- 해석기

스키마

GraphQL 스키마는 GraphQL API의 기반으로, 데이터의 형태를 정의하는 청사진 역할을 합니다. 또한 데이터를 검색 또는 수정하는 방법을 정의하는 클라이언트와 서버 간의 계약이기도 합니다.

GraphQL 스키마는 스키마 정의 언어(SDL)로 작성됩니다. SDL은 구조가 확립된 유형과 필드로 구성되어 있습니다.

- 유형: 유형은 GraphQL이 데이터의 형태와 동작을 정의하는 방식입니다. GraphQL은 이 섹션의 뒷부분에서 설명할 여러 유형을 지원합니다. 스키마에 정의된 각 유형에는 고유한 범위가 포함됩니다. 범위 내에는 GraphQL 서비스에서 사용할 값이나 로직을 포함할 수 있는 하나 이상의 필드가 있습니다. 유형은 다양한 역할을 수행하며, 가장 일반적으로는 객체 또는 스칼라(기본 값 유형) 역할을 합니다.
- 필드: 필드는 유형의 범위 내에 존재하며 GraphQL 서비스에서 요청한 값을 보유합니다. 이는 다른 프로그래밍 언어의 변수와 매우 유사합니다. 필드에 정의하는 데이터의 형태에 따라 요청/응답 작업에서 데이터가 구조화되는 방식이 결정됩니다. 이를 통해 개발자는 서비스의 백엔드가 구현되는 방식을 모르더라도 어떤 결과가 반환될지 예측할 수 있습니다.

스키마의 모습을 시각적으로 확인하기 위해 간단한 GraphQL 스키마의 내용을 살펴보겠습니다. 프로덕션 코드에서 스키마는 일반적으로 `schema.graphql` 또는 `schema.json`이라는 파일에 있습니다. GraphQL 서비스를 구현하는 프로젝트를 살펴본다고 가정해 보겠습니다. 이 프로젝트는 회사 직원 데이터를 저장하고 있으며, `schema.graphql` 파일은 직원 데이터를 검색하고 데이터베이스에 새 직원을 추가하는 데 사용됩니다. 코드는 다음과 같을 수 있습니다.

schema.graphql

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

스키마에는 `Person`, `Query`, `Mutation`이라는 세 가지 유형이 정의되어 있는 것을 볼 수 있습니다. `Person`을 보면, 이 유형을 객체로 만드는 회사 직원 인스턴스의 청사진이라는 것을 짐작할 수 있습니다. 범위 내에는 `id`, `name`, `age`가 있습니다. 이러한 필드는 `Person`의 속성을 정의하는 필드입니다. 즉, 데이터 원본은 각 `Person`의 `name`을 `String` 스칼라(기본) 유형으로 저장하고 `age`를 `Int` 스칼라(기본) 유형으로 저장합니다. `id`는 각 `Person`에 대해 특별하고 고유한 식별자 역할을 합니다. 또한 `!` 기호로 표시되는 필수 값이기도 합니다.

다음 두 객체 유형은 다르게 동작합니다. GraphQL은 스키마에 데이터가 채워지는 방식을 정의하는 특수 객체 유형에 대한 몇 가지 키워드를 예약합니다. `Query` 유형은 소스에서 데이터를 검색합니다. 이 예에서는 쿼리가 데이터베이스에서 `Person` 객체를 검색할 수 있습니다. 이 작업은 RESTful 용어로 `GET` 작업을 떠올리게 할 수 있습니다. `Mutation`은 데이터를 수정합니다. 이 예에서는 변형으로 인해 데이터베이스에 더 많은 `Person` 객체가 추가될 수 있습니다. 이 작업은 `PUT` 또는 `POST`와 같은 상태 변경 작업을 떠올리게 할 수 있습니다. 모든 특수 객체 유형의 동작은 이 섹션의 뒷부분에서 설명합니다.

이 예제의 `Query`가 데이터베이스에서 무언가를 검색한다고 가정해 보겠습니다. `Query`의 필드를 살펴보면 `people`이라는 필드 하나를 볼 수 있습니다. 필드 값은 `[Person]`입니다. 즉, 데이터베이스에서 `Person`의 일부 인스턴스를 검색하려고 합니다. 하지만 대괄호를 추가하면 특정 인스턴스뿐만 아니라 모든 `Person` 인스턴스의 목록을 반환하게 됩니다.

Mutation 유형은 데이터 수정과 같은 상태 변경 작업을 수행하는 역할을 합니다. 변형은 데이터 원본에서 일부 상태 변경 작업을 수행하는 역할을 합니다. 이 예제에서 변형에는 데이터베이스에 새 Person 객체를 추가하는 addPerson이라는 작업이 포함되어 있습니다. 이 변형은 Person을 사용하며 id, name, age 필드에 대한 입력이 필요합니다.

이 시점에서 어떤 동작을 수행하며 함수 이름과 파라미터가 있는 함수와 매우 비슷해 보인다고 가정한다면 코드 구현 없이 addPerson과 같은 작업이 어떻게 작동하는지 궁금할 수 있습니다. 스키마는 선언 역할만 하기 때문에 지금은 작동하지 않습니다. addPerson의 동작을 구현하려면 해석기를 추가해야 합니다. 해석기는 관련 필드(이 경우 addPerson 작업)가 호출될 때마다 실행되는 코드 단위입니다. 작업을 사용하려면 어느 시점에 해석기 구현을 추가해야 합니다. 어떤 면에서는 스키마 작업을 함수 선언으로, 해석기는 정의로 생각할 수 있습니다. 해석기에 대해서는 다른 섹션에서 설명하겠습니다.

이 예제에서는 스키마가 데이터를 조작할 수 있는 가장 간단한 방법만 보여 줍니다. GraphQL 및 AWS AppSync의 기능을 활용하여 복잡하고 강력하며 확장 가능한 애플리케이션을 구축합니다. 다음 섹션에서는 스키마에서 활용할 수 있는 다양한 유형과 필드 동작을 모두 정의하겠습니다.

GraphQL 유형

GraphQL은 다양한 유형을 지원합니다. 이전 섹션에서 보았듯이 유형은 데이터의 형태나 동작을 정의합니다. 유형은 GraphQL 스키마의 기본 구성 요소입니다.

유형은 입력과 출력으로 분류할 수 있습니다. 입력은 특수 객체 유형(Query, Mutation 등)에 대한 인수로 전달될 수 있는 유형인 반면, 출력 유형은 데이터를 저장하고 반환하는 데만 사용됩니다. 유형 및 분류 목록은 아래와 같습니다.

- 객체: 객체에는 엔터티를 설명하는 필드가 있습니다. 예를 들어 객체는 authorName, publishingYear와 같이 특성을 설명하는 필드가 있는 book 등이 될 수 있습니다. 이러한 유형은 엄밀히 말하면 출력 유형입니다.
- 스칼라: int, 문자열 등과 같은 기본 유형입니다. 일반적으로 필드에 할당됩니다. authorName 필드를 예로 들면 'John Smith'와 같은 이름을 저장하는 String 스칼라에 할당할 수 있습니다. 스칼라는 입력 유형과 출력 유형이 모두 될 수 있습니다.
- 입력: 입력을 사용하면 필드 그룹을 인수로 전달할 수 있습니다. 객체와 구조가 매우 비슷하지만, 특수 객체에 인수로 전달할 수 있습니다. 입력을 사용하면 범위에 스칼라, 열거형 및 기타 입력을 정의할 수 있습니다. 입력은 입력 유형만 가능합니다.
- 특수 객체: 특수 객체는 상태 변경 작업을 수행하며 서비스의 까다로운 작업 대부분을 수행합니다. 쿼리, 변형, 구독이라는 세 가지 특수 객체 유형이 있습니다. 일반적으로 쿼리는 데이터를 가져오고, 변형은 데이터를 조작하며, 구독은 지속적인 통신을 위해 클라이언트와 서버 간의 양방향 연결을 열고 유지합니다. 특수 객체는 기능으로 볼 때 입력도 출력도 아닙니다.

- 열거형: 열거형은 미리 정의된 유효 값 목록입니다. 열거형을 호출하는 경우 범위에 정의된 값만 열거형 값이 될 수 있습니다. 예를 들어 교통 신호 목록을 나타내는 `trafficLights`라는 열거형이 있는 경우 `redLight` 및 `greenLight`와 같은 값을 가질 수 있지만 `purpleLight` 값은 가질 수 없습니다. 실제 신호등에는 신호가 너무 많기 때문에 열거형을 사용하여 신호를 정의하고 `trafficLight`를 참조할 때 이러한 열거형만 유효 값이 되도록 할 수 있습니다. 열거형은 입력 유형과 출력 유형이 모두 될 수 있습니다.
- 결합/인터페이스: 결합을 사용하면 클라이언트가 요청한 데이터에 따라 요청에서 하나 이상의 항목을 반환할 수 있습니다. 예를 들어 `title` 필드가 포함된 `Book` 유형과 `name` 필드가 포함된 `Author` 유형이 있는 경우 두 유형 간에 결합을 생성할 수 있습니다. 클라이언트가 데이터베이스에서 'Julius Caesar'라는 문구를 쿼리하려는 경우 결합은 `Book` `title`에서 Julius Caesar(윌리엄 셰익스피어의 희곡)를 반환하고 `Author` `name`에서 Julius Caesar(Commentarii de Bello Gallico의 저자)를 반환할 수 있습니다. 결합은 출력 유형만 될 수 있습니다.

  인터페이스는 객체가 구현해야 하는 필드 집합으로, 인터페이스에 정의된 필드를 구현해야 하는 Java와 같은 프로그래밍 언어의 인터페이스와 약간 유사합니다. 예를 들어 `title` 필드를 포함하는 `Book`이라는 인터페이스를 만들었다고 가정해 보겠습니다. 이후 `Book`을 구현한 `Novel`이라는 유형을 만들었다고 가정합니다. `Novel`은 `title` 필드를 포함해야 합니다. 하지만 `Novel`은 `ISBN`의 `pageCount`와 같이 인터페이스에 없는 다른 필드도 포함할 수 있습니다. 인터페이스는 출력 유형만 될 수 있습니다.

다음 섹션에서는 GraphQL에서 각 유형이 작동하는 방식을 설명합니다.

개체

GraphQL 객체는 프로덕션 코드에서 볼 수 있는 기본 유형입니다. GraphQL에서는 객체를 다른 언어의 변수와 비슷한 여러 필드의 그룹으로 생각할 수 있으며, 각 필드는 값을 보유할 수 있는 유형(일반적으로 스칼라 또는 다른 객체)으로 정의됩니다. 객체는 서비스 구현에서 검색/조작할 수 있는 데이터 단위를 나타냅니다.

객체 유형은 `Type` 키워드를 사용하여 선언됩니다. 스키마 예제를 약간 수정해 보겠습니다.

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

type Occupation {
```

```
  title: String
}
```

여기에서 객체 유형은 Person 및 Occupation입니다. 각 객체에는 고유한 유형이 있는 고유한 필드가 있습니다. GraphQL에는 필드를 다른 유형으로 설정할 수 있는 기능이 있습니다. Person의 occupation 필드에 Occupation 객체 유형이 포함되어 있는 것을 볼 수 있습니다. GraphQL은 데이터를 설명할 뿐 서비스 구현은 설명하지 않기 때문에 이렇게 연결할 수 있습니다.

스칼라

스칼라는 본질적으로 값을 보유하는 기본 유형입니다. AWS AppSync에는 기본 GraphQL 스칼라와 AWS AppSync 스칼라라는 두 가지 유형의 스칼라가 있습니다. 스칼라는 일반적으로 객체 유형 내에 필드 값을 저장하는 데 사용됩니다. 기본 GraphQL 유형에는 Int, Float, String, Boolean, ID가 포함됩니다. 이전 예제를 다시 사용해 보겠습니다.

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

type Occupation {
  title: String
}
```

name 및 title 필드를 분리해 보면 둘 다 String 스칼라를 보유합니다. Name은 'John Smith'와 같은 문자열 값을 반환하고 제목은 'firefighter'와 같은 결과를 반환할 수 있습니다. 일부 GraphQL 구현에서는 Scalar 키워드를 사용하고 유형의 동작을 구현하는 사용자 지정 스칼라도 지원합니다. 그러나 지금은 AWS AppSync에서 사용자 지정 스칼라를 지원하지 않습니다. 스칼라 목록은 AWS AppSync의 스칼라 유형을 참조하세요.

입력

입력 및 출력 유형의 개념으로 인해 인수를 전달할 때 특정 제한이 발생합니다. 공통으로 전달해야 하는 유형, 특히 객체가 제한됩니다. 입력 유형을 사용하여 이 규칙을 우회할 수 있습니다. 입력은 스칼라, 열거형 및 기타 입력 유형을 포함하는 유형입니다.

입력은 input 키워드를 사용하여 정의됩니다.

```
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
}
```

보는 것과 같이 원래 유형을 모방하는 별도의 입력이 있을 수 있습니다. 이러한 입력은 다음과 같이 필드 작업에 자주 사용됩니다.

```
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
}
```

Person을 생성하기 위해 Occupation의 자리에 occupationInput을 계속해서 전달하는 방식을 주목하세요.

이 방식은 입력에 대한 한 가지 시나리오에 불과합니다. 객체를 반드시 1:1로 복사할 필요는 없으며, 프로덕션 코드에서는 이 방법으로 사용하지 않을 가능성이 큽니다. 입력해야 하는 항목만 인수로 정의하여 GraphQL 스키마를 활용하는 것이 좋습니다.

또한 동일한 입력을 여러 작업에 사용할 수 있지만 권장되지는 않습니다. 스키마의 요구 사항이 변경될 경우를 대비하여 각 작업에는 입력의 고유한 사본이 포함되는 것이 이상적입니다.

특수 객체

GraphQL은 스키마가 데이터를 검색/조작하는 방식에 대한 일부 비즈니스 로직을 정의하는 특수 객체를 위한 몇 가지 키워드를 예약합니다. 스키마에는 이러한 키워드가 최대 하나씩만 있을 수 있습니다. 이러한 키워드는 클라이언트가 GraphQL 서비스에 대해 실행하는 모든 요청 데이터의 진입점 역할을 합니다.

type 키워드를 사용하여 특수 객체도 정의합니다. 일반 객체 유형과는 다르게 사용되지만 구현은 매우 비슷합니다.

Queries

쿼리는 읽기 전용 가져오기를 수행하여 소스에서 데이터를 가져온다는 점에서 GET 작업과 매우 유사합니다. GraphQL에서 Query는 서버에 요청을 보내는 클라이언트의 모든 진입점을 정의합니다. GraphQL 구현에는 항상 Query가 있습니다.

다음은 이전 스키마 예제에서 사용한 Query 및 수정된 객체 유형입니다.

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

Query에는 데이터 원본의 Person 인스턴스 목록을 반환하는 people라는 필드가 포함되어 있습니다. 애플리케이션의 동작을 변경해야 하는데 이제 별도의 목적으로 Occupation 인스턴스 목록만 반환해야 한다고 가정해 보겠습니다. 쿼리에 간단히 추가할 수 있습니다.

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

GraphQL에서는 쿼리를 요청의 단일 소스로 취급할 수 있습니다. 보는 것과 같이 서로 다른 엔드포인트를 사용하여 동일한 결과(.../api/1/people 및 .../api/1/occupations)를 달성할 수 있는 RESTful 구현보다 훨씬 간단할 수 있습니다.

이 쿼리에 대한 해석기 구현이 있다고 가정하면 이제 실제 쿼리를 수행할 수 있습니다. Query 유형이 존재하지만 애플리케이션의 코드에서 실행되도록 하려면 명시적으로 호출해야 합니다. 다음과 같은 query 키워드를 사용하여 수행할 수 있습니다.

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

보는 것과 같이 이 쿼리는 getItems라고 하며 people(Person 객체의 목록) 및 occupations(Occupation 객체의 목록)를 반환합니다. people에서는 각 Person의 name 필드만 반환하는 반면, 각 Occupation의 title 필드를 반환합니다. 응답은 다음과 같을 수 있습니다.

```
{
  "data": {
    "people": [
      {
        "name": "John Smith"
      },
      {
        "name": "Andrew Miller"
      },
      .
      .
```

```
      .
    ],
    "occupations": [
      {
        "title": "Firefighter"
      },
      {
        "title": "Bookkeeper"
      },
      .
      .
      .
    ]
  }
}
```

예제 응답은 데이터가 쿼리의 형태를 어떻게 따르는지 보여 줍니다. 검색된 각 항목은 필드 범위 내에 나열됩니다. people 및 occupations는 항목을 별도의 목록으로 반환합니다. 이 방법이 유용하긴 하지만, 사람들의 이름과 직업 목록을 반환하도록 쿼리를 수정하는 것이 더 편리할 수도 있습니다.

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

Person 유형에 유형 Occupation의 occupation 필드가 포함되어 있기 때문에 이는 유효한 수정입니다. people의 범위 내에 나열되면 각 Person의 name과 함께 관련된 Occupation이 title별로 반환됩니다. 응답은 다음과 같을 수 있습니다.

```
}
  "data": {
    "people": [
      {
        "name": "John Smith",
        "occupation": {
          "title": "Firefighter"
        }
      },
```

```
      {
        "name": "Andrew Miller",
        "occupation": {
          "title": "Bookkeeper"
        }
      },
      .
      .
      .
    ]
  }
}
```

Mutations

변형은 PUT 또는 POST와 같은 상태 변경 작업과 비슷합니다. 이러한 작업은 쓰기 작업을 수행하여 소스의 데이터를 수정한 다음 응답을 가져옵니다. 또한 데이터 수정 요청의 진입점을 정의합니다. 쿼리와 달리 변형은 프로젝트의 필요에 따라 스키마에 포함되거나 포함되지 않을 수 있습니다. 다음은 스키마 예제의 변형입니다.

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

addPerson 필드는 데이터 원본에 Person을 추가하는 하나의 진입점을 나타냅니다. addPerson은 필드 이름이며 id, name, age는 파라미터, Person은 반환 유형입니다. Person 유형을 다시 살펴보면 다음과 같습니다.

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

occupation 필드를 추가했습니다. 하지만 객체를 인수로 전달할 수 없기 때문에 이 필드를 Occupation으로 직접 설정할 수는 없습니다. 엄밀히 말하면 출력 유형이기 때문입니다. 대신 인수와 동일한 필드를 포함하는 입력을 전달해야 합니다.

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

```
}
```

새 Person 인스턴스를 만들 때 이 필드를 파라미터로 포함하도록 addPerson을 간편하게 업데이트할 수도 있습니다.

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

업데이트된 스키마는 다음과 같습니다.

```
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
}
```

occupation은 occupationInput에서 title 필드를 전달하여 원래 Occupation 객체 대신 Person 생성을 완료합니다. addPerson에 대한 해석기 구현이 있다고 가정하면 이제 실제 변형을 수행할 수 있습니다. Mutation 유형이 존재하지만 애플리케이션의 코드에서 실행되도록 하려면 명시적으로 호출해야 합니다. 다음과 같은 mutation 키워드를 사용하여 수행할 수 있습니다.

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

```
    }
  }
}
```

이 변형을 createPerson이라고 하며 addPerson은 작업입니다. 새 Person을 만들려면 id, name, age, occupation의 인수를 입력하면 됩니다. addPerson의 범위에서는 name, age 등과 같은 다른 필드도 볼 수 있습니다. 응답은 다음과 같으며, 이러한 필드는 addPerson 작업이 완료된 후 반환됩니다. 예제의 마지막 부분은 다음과 같습니다.

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

이 변형을 사용하면 결과는 다음과 같을 수 있습니다.

```
{
  "data": {
    "addPerson": {
      "id": "1",
      "name": "Steve Powers",
      "age": "50",
      "occupation": {
        "title": "Miner"
      }
    }
  }
}
```

보는 것과 같이 응답은 요청한 값을 변형에 정의된 것과 동일한 형식으로 반환했습니다. 혼란을 줄이고 이후 더 많은 쿼리가 필요한 상황을 방지하기 위해 수정된 모든 값을 반환하는 것이 좋습니다. 변형을 사용하면 범위 내에 여러 작업을 포함할 수 있습니다. 변형에 나열된 순서대로 순차적으로 실행됩니다. 예를 들어 데이터 원본에 직함을 추가하는 addOccupation이라는 또 다른 작업을 만들면 addPerson 뒤에 변형에서 이를 호출할 수 있습니다. addPerson이 먼저 처리된 후 addOccupation이 처리됩니다.

Subscriptions

구독은 WebSocket을 사용하여 서버와 클라이언트 간의 지속적인 양방향 연결을 제공합니다. 일반적으로 클라이언트는 서버를 구독하거나 수신합니다. 서버에 서버 측 변경이 있거나 서버에서 이벤트를 수행할 때마다 구독한 클라이언트가 업데이트를 받게 됩니다. 이 유형의 프로토콜은 여러 클라이언트가 구독하고 서버나 다른 클라이언트에서 발생하는 변경 사항에 대해 알림을 받아야 하는 경우에 유용합니다. 예를 들어 구독을 사용하여 소셜 미디어 피드를 업데이트할 수 있습니다. 사용자 A와 사용자 B라는 두 명의 사용자가 있는데, 둘 다 다이렉트 메시지를 받을 때마다 자동 알림 업데이트를 받도록 구독합니다. 클라이언트 A의 사용자 A가 클라이언트 B의 사용자 B에게 다이렉트 메시지를 보냅니다. 사용자 A의 클라이언트가 다이렉트 메시지를 보내면 서버에서 처리됩니다. 그러면 서버는 사용자 B의 계정에 다이렉트 메시지를 보내고 클라이언트 B에는 자동 알림을 보냅니다.

다음은 스키마 예제에 추가할 수 있는 `Subscription`의 예입니다.

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

이 `personAdded` 필드는 데이터 원본에 새 `Person`이 추가될 때마다 구독한 클라이언트에게 메시지를 보냅니다. `personAdded`에 대한 해석기 구현이 있다고 가정하면 이제 구독을 사용할 수 있습니다. `Subscription` 유형이 존재하지만 애플리케이션의 코드에서 실행되도록 하려면 명시적으로 호출해야 합니다. 다음과 같은 `subscription` 키워드를 사용하여 수행할 수 있습니다.

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

구독은 `personAddedOperation`이며, 작업은 `personAdded`입니다. `personAdded`는 새 `Person` 인스턴스의 `id` 및 `name` 필드를 반환합니다. 변형 예제를 살펴보면 이 작업을 사용하여 `Person`을 추가했습니다.

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

클라이언트가 새로 추가된 `Person`에 대한 업데이트를 구독한 경우 `addPerson` 실행 후 다음과 같은 결과를 확인할 수 있습니다.

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

구독이 제공하는 내용을 요약하면 다음과 같습니다.

구독은 클라이언트와 서버가 빠르면서 안정적인 업데이트를 받을 수 있게 하는 양방향 채널입니다. 일반적으로 표준화되고 안전한 연결을 생성하는 WebSocket 프로토콜을 사용합니다.

구독은 연결 설정 오버헤드를 줄인다는 점에서 유연합니다. 구독하면 클라이언트는 해당 구독을 오랫동안 계속 사용할 수 있습니다. 대부분의 경우 개발자가 구독 기간을 조정하고 요청할 정보를 구성할 수 있도록 함으로써 컴퓨팅 리소스를 효율적으로 사용합니다.

일반적으로 구독을 통해 클라이언트는 한 번에 여러 구독을 설정할 수 있습니다. AWS AppSync에 관해서는 구독은 AWS AppSync 서비스로부터 실시간 업데이트를 수신하는 데만 사용됩니다. 쿼리나 변형을 수행하는 데는 사용할 수 없습니다.

구독의 주요 대안은 지정된 간격으로 쿼리를 전송하여 데이터를 요청하는 폴링입니다. 이 프로세스는 일반적으로 구독보다 덜 효율적이며 클라이언트와 백엔드 모두에 많은 부담을 줍니다.

스키마 예제에서 언급되지 않은 한 가지는 특수 객체 유형도 `schema` 루트에 정의되어야 한다는 사실입니다. 따라서 스키마를 AWS AppSync로 내보내면 다음과 같을 수 있습니다.

schema.graphql

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

.
.
.

type Query {
```

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

열거

열거 또는 열거형은 유형이나 필드에 사용할 수 있는 유효한 인수를 제한하는 특수 스칼라입니다. 즉, 스키마에 열거형이 정의될 때마다 관련 유형이나 필드가 열거형의 값으로 제한됩니다. 열거형은 문자열 스칼라로 직렬화됩니다. 프로그래밍 언어에 따라 GraphQL 열거형을 다르게 처리할 수 있다는 점에 유의하세요. 예를 들어 JavaScript에는 기본 열거형 지원이 없으므로 열거형 값을 int 값에 대신 매핑할 수 있습니다.

열거형은 `enum` 키워드를 사용하여 정의됩니다. 다음은 그 예입니다.

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

`trafficLights` 열거형을 호출할 때 인수는 `solidRed`, `solidYellow`, `solidGreen` 등만 사용할 수 있습니다. 명확하지만 제한된 수의 선택 사항이 있는 항목을 묘사하는 데 열거형을 사용하는 것이 일반적입니다.

결합/인터페이스

GraphQL의 인터페이스 및 결합을 참조하세요.

GraphQL 필드

필드는 유형의 범위 내에 존재하며 GraphQL 서비스에서 요청한 값을 보유합니다. 이는 다른 프로그래밍 언어의 변수와 매우 유사합니다. 예를 들어, `Person` 객체 유형은 다음과 같습니다.

```
type Person {
```

```
    name: String
    age: Int
}
```

이 경우 필드는 name 및 age이고 각각 String 및 Int 값을 보유합니다. 위에 표시된 것과 같은 객체 필드를 쿼리 및 변형의 필드(작업)에서 입력으로 사용할 수 있습니다. 그 예로 아래 Query를 참조하세요.

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

people 필드는 데이터 원본의 Person에 있는 모든 인스턴스를 요청합니다. GraphQL 서버에서 Person을 추가하거나 검색할 때 데이터가 유형 및 필드의 형식을 따를 것으로 예상할 수 있습니다. 즉, 스키마의 데이터 구조에 따라 응답에서 데이터가 구조화되는 방식이 결정됩니다.

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

필드는 데이터를 구조화하는 데 중요한 역할을 합니다. 추가 사용자 지정을 위해 필드에 적용할 수 있는 몇 가지 추가 속성이 아래에 설명되어 있습니다.

List

목록은 지정된 유형의 모든 항목을 반환합니다. 대괄호 []를 사용하여 필드 유형에 목록을 추가할 수 있습니다.

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

Query에서 Person 주위의 대괄호는 데이터 원본의 Person에 있는 모든 인스턴스를 배열로 반환한다는 것을 나타냅니다. 응답에서 각 Person의 name 및 age 값은 구분된 단일 목록으로 반환됩니다.

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

특수 객체 유형에만 국한되지 않습니다. 일반 객체 유형의 필드에서도 목록을 사용할 수 있습니다.

null 아님

null 아님 필드는 응답에서 null이 될 수 없는 필드를 나타냅니다. ! 기호를 사용하여 필드를 null이 아닌 필드로 설정할 수 있습니다.

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

```
  people: [Person]
}
```

name 필드는 명시적으로 null이 될 수 없습니다. 데이터 원본을 쿼리하고 이 필드에 null 입력을 제공하면 오류가 발생합니다.

목록과 null이 아닌 필드를 조합할 수 있습니다. 다음 쿼리를 비교해 보세요.

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

.
.
.

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

.
.
.

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

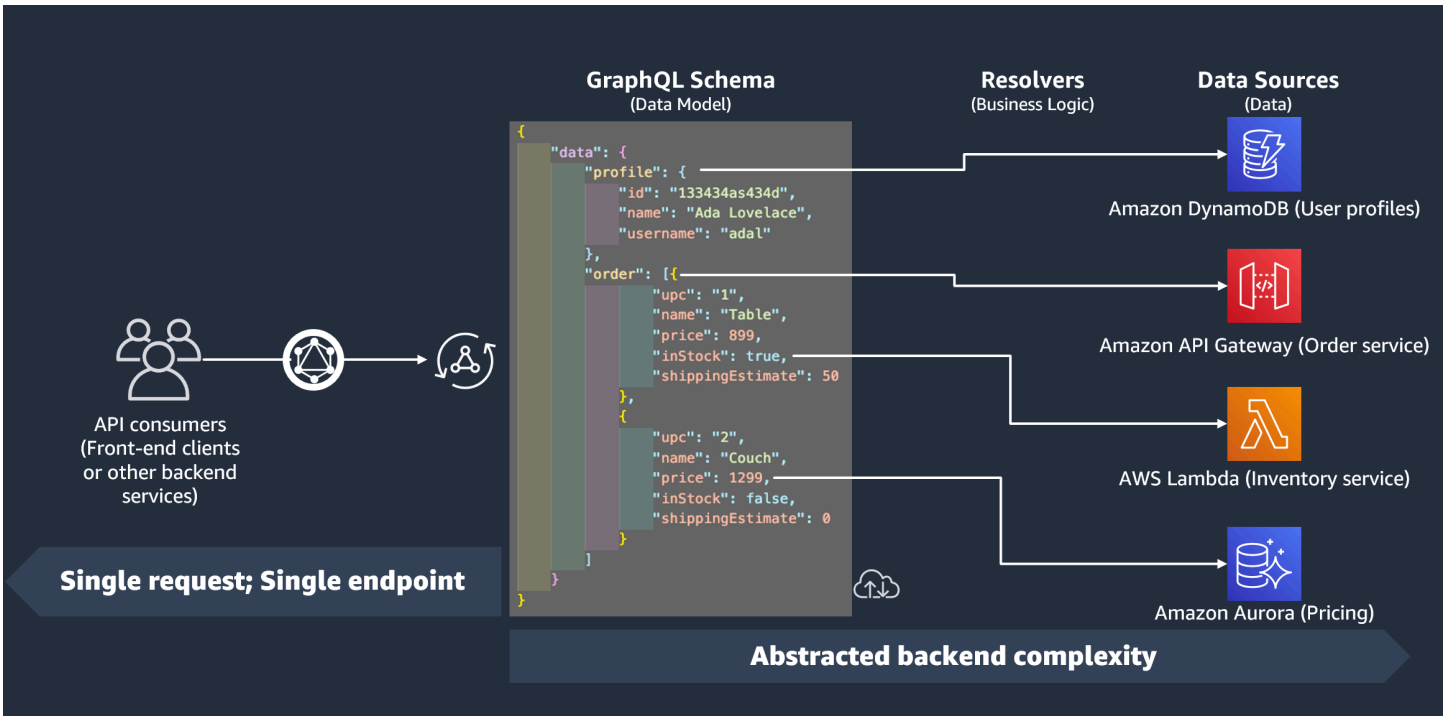
사용 사례 1에서는 목록에 null 항목을 포함할 수 없습니다. 사용 사례 2에서는 목록 자체를 null로 설정할 수 없습니다. 사용 사례 3에서는 목록과 해당 항목이 null일 수 없습니다. 하지만 어떤 경우에도 빈 목록을 반환할 수 있습니다.

보는 것과 같이 GraphQL에는 움직이는 구성 요소가 많이 있습니다. 이 섹션에서는 간단한 스키마의 구조와 스키마가 지원하는 여러 유형 및 필드를 보여드렸습니다. 다음 섹션에서는 GraphQL API의 기타 구성 요소와 이러한 구성 요소가 스키마에서 작동하는 방식에 대해 알아봅니다.

데이터 원본

이전 섹션에서는 스키마가 데이터의 형태를 정의한다는 것을 배웠습니다. 하지만 해당 데이터의 출처는 설명하지 않았습니다. 실제 프로젝트에서 스키마는 서버에 대한 모든 요청을 처리하는 게이트웨이와 같습니다. 요청이 생성되면 스키마는 클라이언트와 인터페이스하는 단일 엔드포인트 역할을 합니

다. 스키마는 데이터 원본의 데이터에 액세스하며 데이터를 처리하고 다시 클라이언트로 전달합니다. 아래 인포그래픽을 참조하세요.



AWS AppSync 및 GraphQL은 BFF(Backend For Frontend) 솔루션을 훌륭하게 구현합니다. 이러한 기능은 함께 작동하여 백엔드를 추상화하여 대규모로 복잡성을 줄입니다. 서비스가 다양한 데이터 원본 또는 마이크로서비스를 사용하는 경우 각 소스(서브그래프)의 데이터 형태를 단일 스키마(수퍼그래프)로 정의하여 기본적으로 복잡성을 어느 정도 추상화할 수 있습니다. 즉, GraphQL API는 하나의 데이터 원본을 사용하도록 제한되지 않습니다. 원하는 수의 데이터 원본을 GraphQL API와 연결하고 해당 데이터 원본이 서비스와 상호 작용하는 방식을 코드에 지정할 수 있습니다.

인포그래픽에서 볼 수 있듯이 GraphQL 스키마에는 클라이언트가 데이터를 요청하는 데 필요한 모든 정보가 포함되어 있습니다. 즉, REST의 경우처럼 여러 요청을 처리하는 대신 단일 요청으로 모든 작업을 처리할 수 있습니다. 이러한 요청은 서비스의 유일한 엔드포인트인 스키마를 거칩니다. 요청이 처리되면 해석기(다음 섹션에서 설명함)가 코드를 실행하여 관련 데이터 원본의 데이터를 처리합니다. 응답이 반환되면 데이터 원본에 연결된 서브그래프가 스키마의 데이터로 채워집니다.

AWS AppSync에서는 여러 데이터 원본 유형을 지원합니다. 아래 테이블에서는 각 유형에 대해 설명하고, 각 유형의 이점을 나열하고, 추가 컨텍스트를 위한 유용한 링크를 제공합니다.

데이터 소스	Description	이점	추가 정보
Amazon DynamoDB	"Amazon DynamoDB는 완전관리형 NoSQL	• 어떤 규모에서든 성능 구현: DynamoDB	• DynamoDB 공식 설명서

데이터 소스	Description	이점	추가 정보
	데이터베이스 서비스로서 원활한 확장성과 함께 빠르고 예측 가능한 성능을 제공합니다. DynamoDB는 분산 데이터베이스를 운영하고 크기 조정하는 데 따른 관리 부담을 줄여서 하드웨어 프로비저닝, 설정 및 구성, 복제, 소프트웨어 패치 또는 클러스터 크기 조정에 대해 걱정할 필요가 없게 합니다. 또한 DynamoDB는 유휴 시 암호화를 제공하여 중요한 데이터 보호와 관련된 운영 부담 및 복잡성을 제거합니다."	는 어떤 규모에서든 일관된 성능을 제공하도록 설계되었습니다. 이는 파티션 사용을 통해 가능합니다. DynamoDB는 테이블을 여러 개의 할당으로 자동 분할하여 여러 노드에 걸쳐 다수의 SSD에 저장합니다. 이렇게 하면 일반적으로 네트워크 처리량이 증가하고 지연 시간이 줄어듭니다. • 대규모 용량: DynamoDB는 트래픽을 모니터링하여 네트워크에 과부하가 장기간 지속되는 경우 처리량을 자동으로 조정할 수 있도록 합니다. • 가용성 및 내결함성: DynamoDB는 여러 개의 물리적으로 격리된 리전에서 지원되며, 각 리전에는 물리적으로 격리된 가용 영역이 여러 개 있습니다. DynamoDB는 서비스 중단 시 자동으로 백업 영역으로 전환	• 파티션 • Auto Scaling • 내결함성 • 모니터링(Monitoring) • 보안 • GraphQL 및 DynamoDB • DynamoDB에 대한 해석기 작업 • 요금 모델

데이터 소스	Description	이점	추가 정보
		합니다. 또한 데이터 보장을 위해 데이터를 수동으로 백업하고 복제할 수도 있습니다. • 로깅 및 모니터링: DynamoDB는 테이블에 대한 여러 분석 도구를 제공합니다. 테이블 성능을 모니터링하고 경보를 생성하여 서비스의 급격한 변경 사항을 알릴 수 있습니다. • 보안: DynamoDB는 엄격한 프로토콜을 따라 데이터가 조직의 보안 요구 사항을 준수하도록 합니다. • AWS AppSync와 통합: DynamoDB는 저희 서비스와 원활하게 통합됩니다. 새 DynamoDB 테이블을 생성하고 이 테이블에서 스키마를 자동으로 생성하여 개발 프로세스를 간소화할 수 있습니다. 또한 해석기의 계정에 있는 기존 DynamoDB 테이블의 데이터를 간편하게 요청할 수 있도록	

데이터 소스	Description	이점	추가 정보
		전체 작업 컬렉션을 제공합니다.	

데이터 소스	Description	이점	추가 정보

AWS Lambda	"AWS Lambda은 서버를 프로비저닝하거나 관리하지 않고도 코드를 실행할 수 있게 해 주는 컴퓨팅 서비스입니다.
	
	Lambda는 고가용성 컴퓨팅 인프라에서 코드를 실행하고 서버와 운영 체제 유지 관리, 용량 프로비저닝 및 자동 조정, 코드 및 보안 패치 배포, 로깅 등 모든 컴퓨팅 리소스 관리를 수행합니다. Lambda를 사용하면 Lambda가 지원하는 언어 런타임 중 하나로 코드를 제공하기만 하면 됩니다."
		• 종량제 모델: Lambda는 리소스를 사용하는 경우에만 요금을 부과합니다. 또한 애플리케이션 요구 사항에 따라 사용되는 리소스의 양을 조정할 수 있습니다.
		• 자동 조정: 때로는 애플리케이션에 특정 프로세스를 위해 추가 컴퓨팅 성능이 필요할 수 있습니다. Lambda를 사용하면 애플리케이션 요구 사항에 맞게 컴퓨팅 리소스를 자동으로 조정할 수 있습니다.
		• 배포 시간 단축: 배포 패키지를 통해 개발 프로세스를 간소화할 수 있습니다. 패키지를 사용하여 Lambda 서비스에 함수 코드를 업로드하세요. 그런 다음 런타임 환경을 사용하여 함수를 테스트하고 실행할 수 있습니다.
		• 다양성: Lambda는 다양한 사용 사례에서 사용할 수 있습니
			• 공식 설명서
			  • 스케일링
			  • 배포
			  • 런타임
			• Lambda 해석기 자습서
			• 요금 모델

데이터 소스	Description	이점	추가 정보
		다. Lambda를 타사 서비스 및 AWS 서비스와 원활하게 통합할 수 있습니다. 일부 예로는 CI/CD 파이프라인 및 대량 메일링 서비스가 있습니다. • AWS AppSync와 통합: 해석기에서 Lambda 함수를 간편하게 호출하여 요청을 처리할 수 있습니다. 저희 서비스는 Lambda 호출을 수행하기 위한 간소화된 요청 작업을 제공합니다. 단일 호출과 일괄 호출이 모두 허용됩니다.	

데이터 소스	Description	이점	추가 정보
OpenSearch	"Amazon OpenSearch Service는 AWS 클라우드에서 OpenSearch 클러스터를 손쉽게 배포, 운영 및 확장할 수 있도록 해주는 관리형 서비스입니다. Amazon OpenSearch Service는 OpenSearch 및 레거시 Elasticsearch OSS(소프트웨어의 최종 오픈 소스 버전인 7.10까지)를 지원합니다. 클러스터를 생성할 때 어떤 검색 엔진을 사용할지 선택할 수 있습니다. OpenSearch는 로그 분석, 실시간 애플리케이션 모니터링, 클릭스트림 분석 같은 사용 사례를 위한 완전한 오픈 소스 검색 및 분석 엔진입니다. 자세한 내용은 OpenSearch 설명서를 참조하세요. Amazon OpenSearch Service는 OpenSearch 클러스터에 대한 모든 리소스를 프로비저닝하고 시작합니다. 또한 실패한 OpenSearch Service 노드를 자동	• 확장: OpenSearch Serverless를 통해 서비스 요구 사항에 맞게 서비스를 쉽게 확장할 수 있습니다. • 데이터 모으기: OpenSearch Ingestion을 사용하여 데이터를 가져오고 처리하고 분석할 수 있습니다. 데이터 모으기를 위한 여러 애플리케이션이 있으며, 여기에서 찾을 수 있습니다. • 보안: OpenSearch는 IAM, CloudTrail, VPC, 인증 등을 포함한 AWS 보안 구성을 관리할 수 있습니다. • 가용성: OpenSearch는 서비스 내에서 다양한 리전 및 가용 영역도 지원합니다. • AWS AppSync와 통합: AWS AppSync에서는 GraphQL API를 사용하여 계정에서 기존 OpenSearch Service 도메인의 데이터를 저장하고 검색할 수 있습니다.	• 공식 설명서 • 서버리스 • 요금 모델

데이터 소스	Description	이점	추가 정보
	으로 감지한 다음 교체해 자체 관리형 인프라와 관련된 오버헤드를 줄입니다. API를 한 번만 호출하거나 콘솔에서 몇 번만 클릭하여 클러스터를 조정할 수 있습니다."		
HTTP 엔드포인트	HTTP 엔드포인트를 데이터 원본으로 사용할 수 있습니다. AWS AppSync는 파라미터 및 페이로드와 같은 관련 정보와 함께 엔드포인트에 요청을 보낼 수 있습니다. HTTP 응답은 해석기에 노출되며, 해석기는 작업이 완료된 후 최종 응답을 반환합니다.	• Lambda와 같은 서비스와 통합되지 않는 간단한 애플리케이션에 유용합니다.	• 해석기 참조

데이터 소스	Description	이점	추가 정보

Amazon EventBridge	"EventBridge는 이벤트를 사용하여 애플리케이션 구성 요소를 서로 연결하는 서버리스 서비스로, 확장 가능한 이벤트 기반 애플리케이션을 더 쉽게 구축할 수 있습니다. EventBridge를 사용하여 자체 개발 애플리케이션, AWS 서비스 및 타사 소프트웨어와 같은 소스의 이벤트를 조직 전체의 소비자 애플리케이션으로 라우팅할 수 있습니다. EventBridge는 이벤트를 수집, 필터링, 변환 및 전달하는 간단하고 일관된 방법을 제공하므로 새로운 애플리케이션을 빠르게 빌드할 수 있습니다."	• 이벤트 기반 아키텍처: 이벤트 기반 아키텍처를 활용할 수 있습니다.
		• 일정: EventBridge Scheduler를 사용하면 이벤트 패턴 대신 cron 표현식을 사용하여 작업과 규칙을 자동화하거나 시간 간격을 설정할 수 있습니다.
		• 파이프: EventBridge 파이프를 사용하면 이벤트를 대상에 전송하기 전에 이벤트 버스를 추가 필터링 이벤트 패턴 및 데이터 변환을 통한 보강을 포함하는 파이프로 대체할 수 있습니다.
		• AWS AppSync와 통합: AWS AppSync에서는 해석기를 사용하여 이벤트 버스로 이벤트를 보낼 수 있습니다.
			• 공식 설명서
			  • 파이프
			  • 스케줄러
			• 해석기 참조
			• 요금 모델

데이터 소스	Description	이점	추가 정보

관계형 데이터베이스	"Amazon Relational Database Service(RDS)는 AWS 클라우드에서 관계형 데이터베이스를 더 쉽게 설치, 운영 및 확장할 수 있는 웹 서비스입니다. 이 서비스는 산업 표준 관계형 데이터베이스를 위한 경제적이고 크기 조절이 가능한 용량을 제공하고 공통 데이터베이스 관리 작업을 관리합니다."	• 간편한 관리: RDS는 정기적으로 리소스에 대한 유지 관리를 수행합니다. 이러한 유지 관리에는 DB 인스턴스의 기본 하드웨어, 기본 운영 체제(OS) 또는 데이터베이스 엔진 버전에 대한 업데이트가 수반되는 경우가 많습니다. 일반적인 상황에서는 업데이트 수행 시기를 결정할 수 있습니다(예외로는 보안 패치가 있음).
		• 권장 사항: RDS의 권장 사항 기능은 인스턴스의 잠재적 문제를 해결하기 위한 자동화된 제안을 제공합니다.
		• 가용성: RDS는 전 세계의 여러 물리적 리전에서 사용할 수 있습니다. 데이터베이스 요구 사항을 여러 노드에 간편하게 분산하여 고객에게 더 나은 서비스를 제공할 수 있습니다.
		• 사용자 지정: RDS는 대기업의 요구 사항
			• 공식 설명서
			  • 기능
			  • Maintenance
			  • 권장 사항
			  • 스토리지 옵션
			  • 가용성
			  • 보안
			• 요금 모델

데이터 소스	Description	이점	추가 정보
		을 충족하도록 맞춤화되었습니다. RDS는 컴퓨팅, 빠른 배포, 확장성 및 스토리지를 위한 다양한 옵션을 제공합니다. • 보안: RDS는 여러 도구 및 서비스와 통합되어 사용자, 데이터베이스, 네트워크 수준에서 데이터베이스 보안을 유지합니다. • AWS AppSync와 통합: 안정적인 백엔드 솔루션을 찾고 있다면 AWS AppSync를 통해 인스턴스를 데이터 원본으로 사용하여 데이터를 전송, 처리, 저장 및 반환할 수 있습니다.	

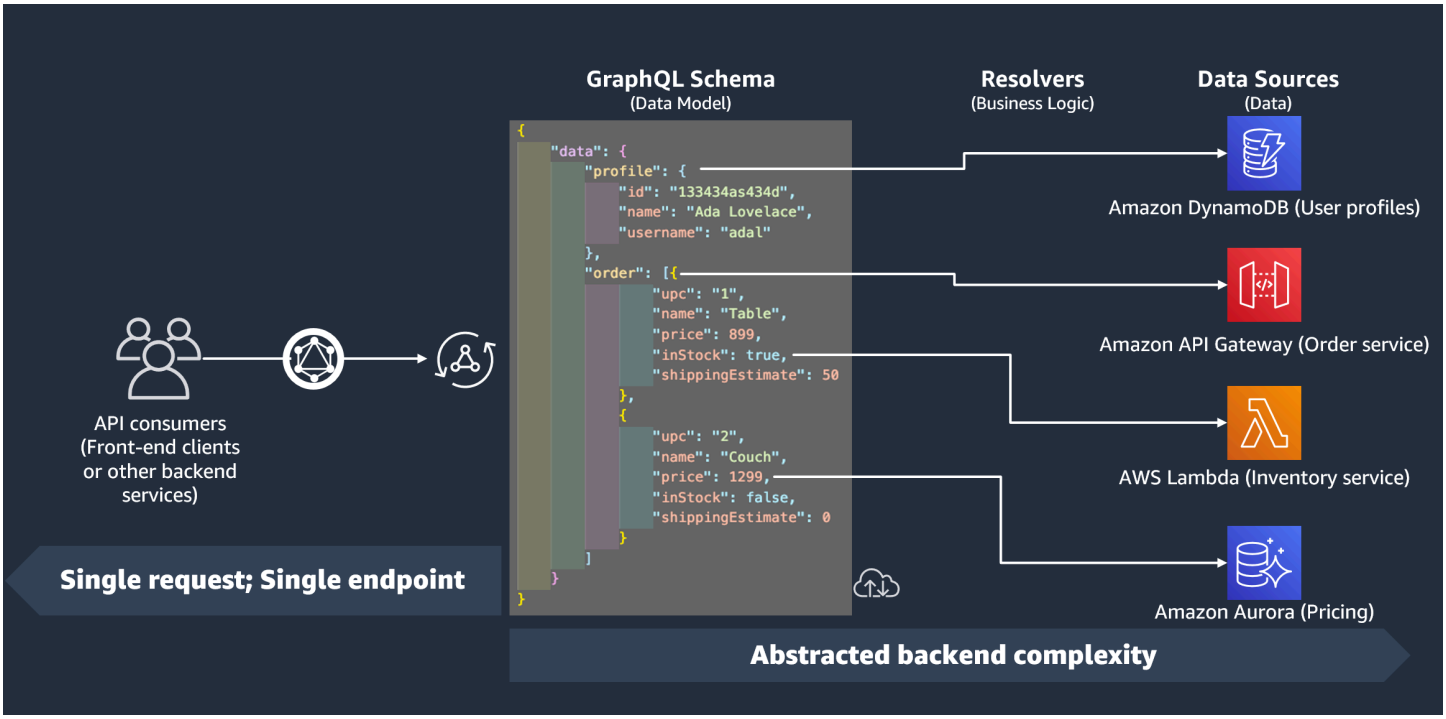
데이터 소스	Description	이점	추가 정보
데이터 원본 사용 안 함	데이터 원본 서비스를 사용할 계획이 없다면 none으로 설정할 수 있습니다. 명시적으로는 여전히 데이터 원본으로 분류되지만 none 데이터 원본은 저장 매체가 아닙니다. 그럼에도 불구하고 특정 상황에서는 데이터 조작 및 패스스루에 여전히 유용합니다.	• 데이터 변환과 같은 작업에 잠재적으로 유용함 • 로컬에서 문제를 해결할 때 유용함	• 해석기 참조

> **Tip**
>
> 데이터 원본이 AWS AppSync와 상호 작용하는 방식에 대한 자세한 내용은 데이터 원본 연결을 참조하세요.

해석기

이전 섹션에서 스키마와 데이터 원본의 구성 요소에 대해 알아보았습니다. 이제 스키마와 데이터 원본이 상호 작용하는 방식을 알아보겠습니다. 모든 작업은 해석기에서 시작됩니다.

해석기는 서비스에 요청이 들어왔을 때 해당 필드의 데이터가 해석되는 방식을 처리하는 코드 단위입니다. 해석기는 스키마의 유형 내 특정 필드에 연결됩니다. 쿼리, 변형 및 구독 필드 작업의 상태 변경 작업을 구현하는 데 가장 일반적으로 사용됩니다. 해석기는 클라이언트의 요청을 처리한 다음 결과를 반환하며, 결과는 객체 또는 스칼라와 같은 출력 유형 그룹일 수 있습니다.



해석기 런타임

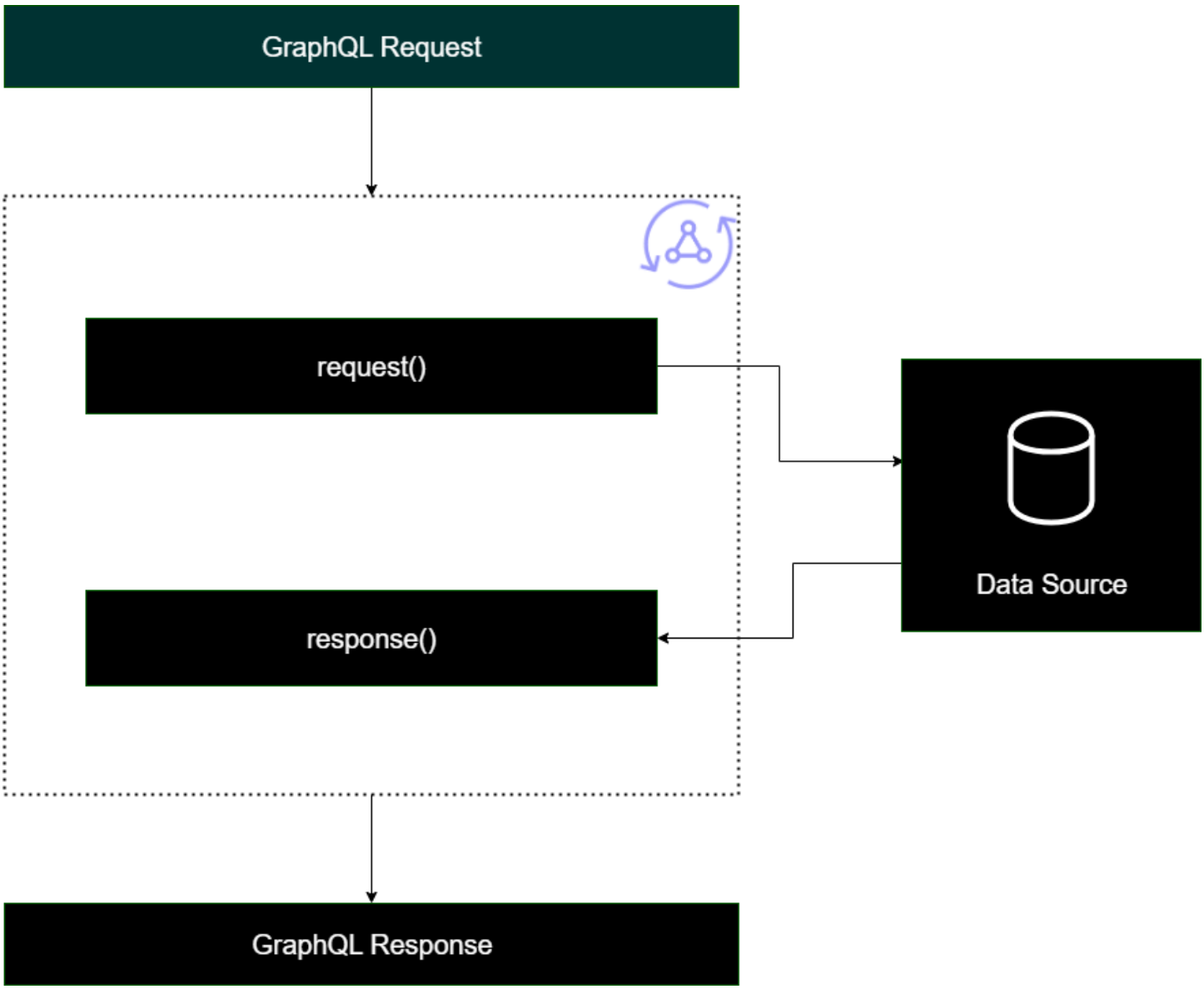
AWS AppSync에서는 먼저 해석기의 런타임을 지정해야 합니다. 해석기 런타임은 해석기가 실행되는 환경을 나타냅니다. 또한 해석기를 작성할 언어도 지정합니다. AWS AppSync는 현재 JavaScript용 APPSYNC_JS 및 VTL(Velocity Template Language)을 지원합니다. JavaScript용 해석기 및 함수에 대한 JavaScript 런타임 기능 또는 VTL용 해석기 매핑 템플릿 유틸리티 참조를 참조하세요.

해석기 구조

코드와 관련해서는 해석기를 몇 가지 방식으로 구성할 수 있습니다. 단위 해석기와 파이프라인 해석기가 있습니다.

단위 해석기

단위 해석기는 데이터 원본에 대해 실행되는 단일 요청 및 응답 핸들러를 정의하는 코드로 구성됩니다. 요청 핸들러는 컨텍스트 객체를 인수로 사용하고 데이터 원본을 호출하는 데 사용된 요청 페이로드를 반환합니다. 응답 핸들러는 실행된 요청의 결과와 함께 데이터 소스로부터 페이로드를 다시 받습니다. 응답 핸들러는 페이로드를 GraphQL 응답으로 변환하여 GraphQL 필드를 해석합니다.



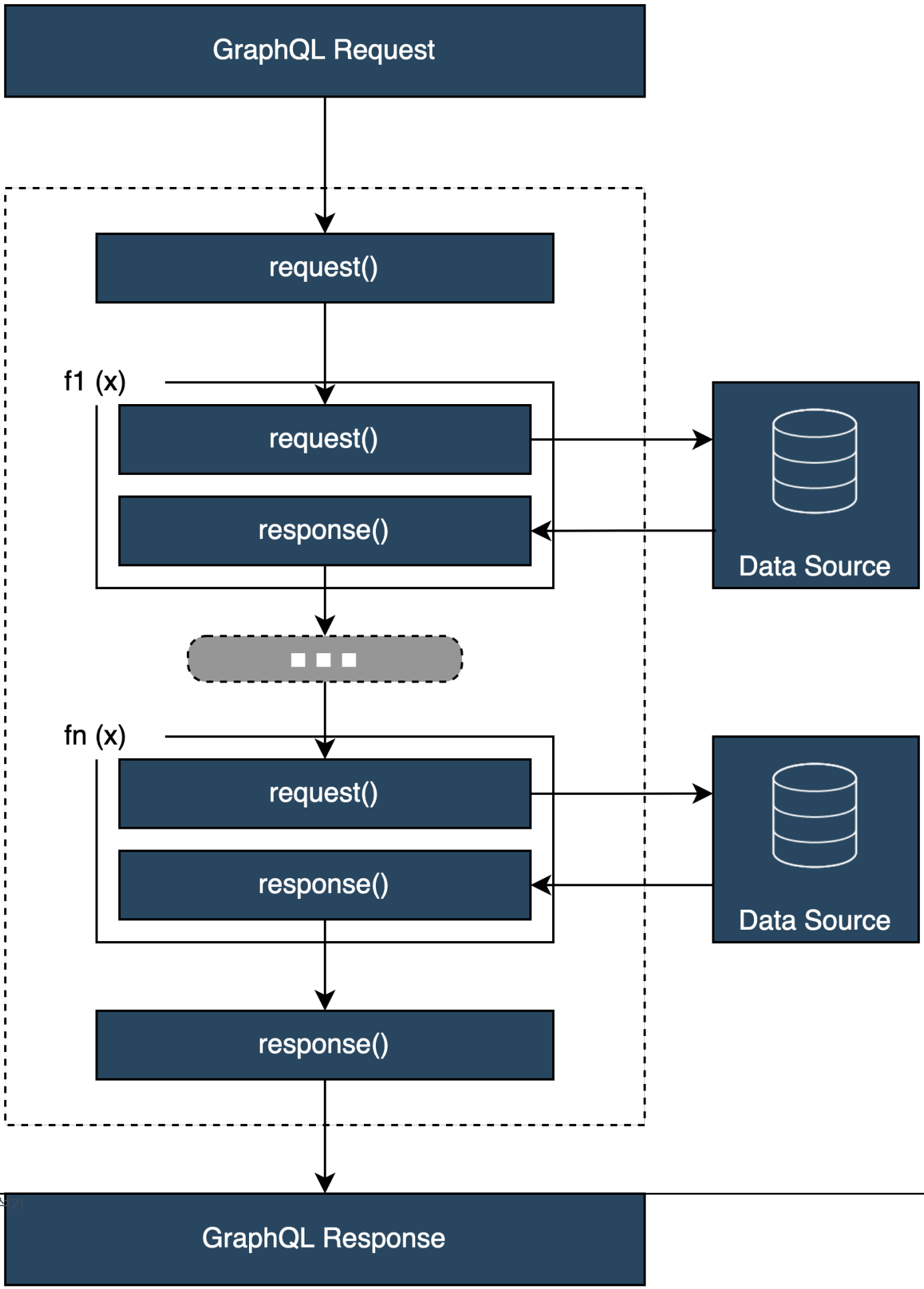
파이프라인 해석기

파이프라인 해석기를 구현할 때는 다음과 같은 일반적인 구조를 따릅니다.

- 사전 단계: 클라이언트가 요청을 보내면 사용 중인 스키마 필드(일반적으로 쿼리, 변형, 구독)의 해석기에 요청 데이터가 전달됩니다. 해석기는 데이터가 해석기를 통해 이동하기 전에 일부 사전 처리 작업을 수행할 수 있는 전 단계 핸들러를 사용하여 요청 데이터 처리를 시작합니다.
- 함수: 이전 단계가 실행되면 요청이 함수 목록으로 전달됩니다. 목록의 첫 번째 함수는 데이터 소스에 대해 실행됩니다. 함수는 자체 요청 및 응답 핸들러를 포함하는 해석기 코드의 하위 집합입니다. 요청 핸들러는 요청 데이터를 가져와 데이터 원본에 대해 작업을 수행합니다. 응답 핸들러는 데이터 원본의 응답을 다시 목록으로 전달하기 전에 해당 데이터 원본의 응답을 처리합니다. 함수가 두 개

이상인 경우 요청 데이터는 목록의 다음 함수로 전송되어 실행됩니다. 목록에 있는 함수는 개발자가 정의한 순서대로 순차적으로 실행됩니다. 모든 함수가 실행되면 최종 결과가 사후 단계로 전달됩니다.

- 사후 단계: 사후 단계는 GraphQL 응답으로 전달하기 전에 최종 함수의 응답에 대한 몇 가지 최종 작업을 수행할 수 있는 핸들러 함수입니다.


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

해석기 핸들러 구조

핸들러는 일반적으로 Request 및 Response라는 함수입니다.

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

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

단위 해석기에는 이러한 함수 집합이 하나뿐입니다. 파이프라인 해석기에는 사전 단계와 사후 단계를 위한 이러한 함수 집합이 있고 함수별로 추가 집합이 있습니다. 시각적으로 확인해 볼 수 있도록 간단한 Query 유형을 살펴보겠습니다.

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

이 쿼리는 유형 String의 helloWorld라는 필드 하나가 있는 간단한 쿼리입니다. 이 필드가 항상 'Hello World'라는 문자열을 반환하게 한다고 가정해 보겠습니다. 이 동작을 구현하려면 이 필드에 해석기를 추가해야 합니다. 단위 해석기에서는 다음과 같이 추가할 수 있습니다.

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

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

데이터를 요청하거나 처리하지 않기 때문에 request는 비워 두어도 됩니다. 또한 데이터 원본이 None이라고 가정할 수도 있는데, 이 코드를 간접 호출할 필요가 없다는 뜻입니다. 응답은 단순히 'Hello World'를 반환합니다. 이 해석기를 테스트하려면 다음 쿼리 유형을 사용하여 요청해야 합니다.

```
query helloWorldTest {
  helloWorld
}
```

이 쿼리는 helloWorld 필드를 반환하는 helloWorldTest라는 쿼리입니다. helloWorld 필드 해석기는 또한 실행 시 다음 응답을 실행하고 반환합니다.

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

이와 같은 상수를 반환하는 작업은 가장 간단하게 할 수 있는 작업입니다. 실제로는 입력, 목록 등을 반환하게 됩니다. 더 복잡한 예는 다음과 같습니다.

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

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

여기서는 Books의 목록을 반환합니다. DynamoDB 테이블을 사용하여 책 데이터를 저장한다고 가정해 보겠습니다. 핸들러는 다음과 같을 수 있습니다.

```
/**
 * 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;
}
```

요청은 내장된 스캔 작업을 사용하여 테이블의 모든 항목을 검색하고 결과를 컨텍스트에 저장한 다음 응답으로 전달했습니다. 응답은 결과 항목을 가져와 응답에 반환했습니다.

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

        ...

      ]
    }
  }
}
```

해석기 컨텍스트

해석기에서 핸들러 체인의 각 단계에서는 이전 단계의 데이터 상태를 알고 있어야 합니다. 한 핸들러의 결과를 저장하고 다른 핸들러에 인수로 전달할 수 있습니다. GraphQL은 네 가지 기본 해석기 인수를 정의합니다.

해석기 기본 인수	Description
`obj`, `root`, `parent` 등.	상위 항목의 결과입니다.
`args`	GraphQL 쿼리의 필드에 제공된 인수입니다.
`context`	모든 해석기에 제공되며 현재 로그인한 사용자 또는 데이터베이스 액세스와 같은 중요한 컨텍스트 정보를 보유하는 값입니다.
`info`	현재 쿼리와 관련된 필드별 정보와 스키마 세부 정보를 포함하는 값입니다.

AWS AppSync에서 context(ctx) 인수는 위에서 언급한 모든 데이터를 보유할 수 있습니다. 요청별로 생성되는 객체이며 권한 부여 보안 인증, 결과 데이터, 오류, 요청 메타데이터 등과 같은 데이터를 포함합니다. 컨텍스트는 프로그래머가 요청의 다른 부분에서 오는 데이터를 조작할 수 있는 간단한 방법입니다. 이 스니펫을 다시 살펴보세요.

```
/**
 * 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;
}
```

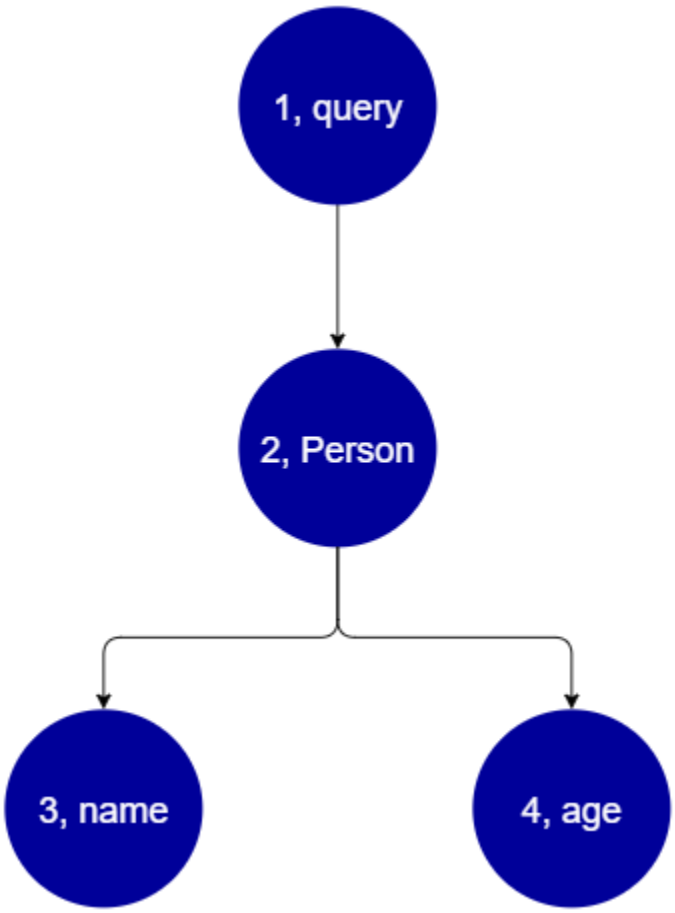
요청에는 컨텍스트(ctx)가 인수로 제공되는데, 이는 요청의 상태입니다. 테이블의 모든 항목을 스캔한 다음 result의 컨텍스트에 결과를 다시 저장합니다. 그러면 컨텍스트가 응답 인수로 전달되고, 응답 인수는 result에 액세스하여 내용을 반환합니다.

요청 및 구문 분석

GraphQL 서비스에 쿼리를 생성하는 경우 먼저 구문 분석 및 검증 프로세스를 거친 후 실행해야 합니다. 요청이 구문 분석되고 추상 구문 트리로 번역됩니다. 스키마에 대해 여러 검증 알고리즘을 실행하여 트리의 내용을 검증합니다. 검증 단계가 완료되면 트리의 노드를 순회하여 처리합니다. 해석기가 간접적으로 호출되고, 결과가 컨텍스트에 저장되며, 응답이 반환됩니다. 다음 쿼리를 예로 들어 보겠습니다.

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

name 및 age 필드와 함께 Person이 반환됩니다. 이 쿼리를 실행하면 다음과 같은 트리가 나옵니다.



트리에서 보면 이 요청이 스키마의 루트에서 `Query`를 검색하는 것으로 보입니다. 쿼리 내에서 `Person` 필드가 해석됩니다. 이전 예제를 보면 사용자의 입력, 값 목록 등일 수 있습니다. `Person`은 필요한 필드(`name` 및 `age`)를 보유하는 객체 유형과 연결되어 있을 가능성이 높습니다. 이 두 개의 하위 필드를 찾으면 주어진 순서대로(`name` 다음 `age`) 해석됩니다. 트리가 완전히 해석되면 요청이 완료되고 클라이언트에게 다시 전송됩니다.

GraphQL의 추가 속성

GraphQL은 대규모로 단순성과 견고성을 유지하기 위한 몇 가지 설계 원칙으로 구성됩니다.

선언적

GraphQL은 선언적이므로 사용자는 쿼리하려는 필드만 선언하여 데이터를 설명(형태 지정)할 수 있습니다. 응답은 이러한 속성에 대한 데이터만 반환합니다. 예를 들어, ISBN 13 `id` 값이 *9780199536061*인 DynamoDB 테이블에서 `Book` 객체를 검색하는 작업은 다음과 같습니다.

```
{
```

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

응답은 페이로드의 필드(name, year, author)만 반환하며 다른 필드는 반환하지 않습니다.

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

이러한 설계 원칙으로 인해 GraphQL은 복잡한 시스템에서 REST API가 다루는 과다 가져오기와 과소 가져오기라는 고질적인 문제를 해결합니다. 그 결과 데이터 수집이 더 효율적으로 이루어지고 네트워크 성능이 향상됩니다.

계층적

GraphQL은 요청된 데이터를 사용자가 애플리케이션의 요구에 맞게 구성할 수 있다는 점에서 유연합니다. 요청된 데이터는 항상 GraphQL API에 정의된 속성의 유형과 구문을 따릅니다. 예를 들어, 다음 스니펫은 Book *9780199536061*에 연결된 모든 저장된 인용 문자열과 페이지를 반환하는 quotes라는 새로운 필드 범위를 사용한 getBook 작업을 보여 줍니다.

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

이 쿼리를 실행하면 다음 결과가 반환됩니다.

```
{
  "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
         }
      ]
    }
  }
}
```

보는 것과 같이 요청된 책에 연결된 quotes 필드는 쿼리에서 설명한 것과 동일한 형식의 배열로 반환되었습니다. 여기에 표시되지는 않았지만 GraphQL은 검색하는 데이터의 위치를 상관하지 않는다는 추가 이점이 있습니다. Books 및 quotes는 별도로 저장할 수 있지만, GraphQL은 연결이 존재하는 한 정보를 계속 검색합니다. 즉, 쿼리는 단일 요청으로 다수의 독립형 데이터를 검색할 수 있습니다.

내관적

GraphQL은 스스로 문서화하는 내관적 성격을 지닙니다. 사용자가 스키마 내의 기본 유형과 필드를 볼 수 있는 몇 가지 내장된 작업을 지원합니다. 예를 들어, 다음은 date 및 description 필드가 있는 Foo 유형입니다.

```
type Foo {
```

```
 date: String
 description: String
}
```

_type 작업을 사용하여 스키마 아래에 있는 입력 메타데이터를 찾을 수 있습니다.

```
{
  __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
      }
    }
  }
}
```

이 작업은 다음 응답을 반환합니다.

```
{
  "__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
      },
    ]
  }
}
```

이 기능을 사용하여 특정 GraphQL 스키마가 지원하는 유형과 필드를 확인할 수 있습니다. GraphQL은 이러한 내관적 작업을 다양하게 지원합니다. 자세한 정보는 내부 검사를 참조하세요.

강력한 유형 지정

GraphQL은 유형 및 필드 시스템을 통해 강력한 유형 지정을 지원합니다. 스키마에서 무언가를 정의할 때는 런타임 전에 검증할 수 있는 유형이 있어야 합니다. 또한 GraphQL의 구문 사양을 따라야 합니다. 이 개념은 다른 언어의 프로그래밍과 다르지 않습니다. 예를 들어 이전의 Foo 유형은 다음과 같습니다.

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

Foo가 생성될 객체라는 것을 알 수 있습니다. Foo의 인스턴스 안에는 date 및 description 필드가 있으며, 둘 다 String 기본 유형(스칼라)입니다. 구문상으로는 Foo가 선언되었고 해당 필드가 범위 내에 존재한다는 것을 알 수 있습니다. 이렇게 유형 검사와 논리적 구문을 조합하면 GraphQL API를 간결하고 이해하기 쉽게 만들 수 있습니다. GraphQL의 유형 지정 및 구문 사양은 여기에서 확인할 수 있습니다.

시작하기: 첫 번째 GraphQL API 만들기

AWS AppSync 콘솔을 사용하여 GraphQL API를 구성하고 실행할 수 있습니다. GraphQL API에는 일반적으로 세 가지 구성 요소가 필요합니다.

1. GraphQL 스키마 - GraphQL 스키마는 API의 청사진입니다. 작업이 실행될 때 요청할 수 있는 유형과 필드를 정의합니다. 스키마를 데이터로 채우려면 데이터 원본을 GraphQL API에 연결해야 합니다. 이 빠른 시작 안내서에서는 사전 정의된 모델을 사용하여 스키마를 만들어 봅니다.
2. 데이터 원본 - GraphQL API를 채우는 데 필요한 데이터가 포함된 리소스입니다. DynamoDB 테이블, Lambda 함수 등이 될 수 있습니다. AWS AppSync는 강력하고 확장 가능한 GraphQL API를 빌드할 수 있도록 다양한 데이터 원본을 지원합니다. 데이터 원본은 스키마의 필드에 연결됩니다. 필드에 대해 요청이 수행될 때마다 소스의 데이터가 필드를 채웁니다. 이 메커니즘은 해석기에서 제어합니다. 이 빠른 시작 안내서에서는 스키마와 함께 사전 정의된 모델을 사용하여 데이터 원본을 만들어 봅니다.
3. 해석기 - 해석기는 스키마 필드를 데이터 원본에 연결하는 역할을 합니다. 소스에서 데이터를 검색한 다음 필드에 정의된 내용을 기반으로 결과를 반환합니다. AWS AppSync는 GraphQL API용 해석기를 작성할 수 있도록 JavaScript와 VTL을 모두 지원합니다. 이 빠른 시작 안내서에서는 스키마와 데이터 원본을 기반으로 해석기가 자동으로 생성됩니다. 이 섹션에서는 이에 대해 자세히 다루지 않습니다.

AWS AppSync는 모든 GraphQL 구성 요소의 생성 및 구성을 지원합니다. 콘솔을 열면 다음 방법을 사용하여 API를 생성할 수 있습니다.

1. 사전 정의된 모델을 통해 API를 생성하고 이를 지원하는 새로운 DynamoDB 테이블(데이터 원본)을 설정하여 사용자 지정된 GraphQL API를 설계합니다.
2. 빈 스키마를 사용하고 데이터 원본이나 해석기 없이 GraphQL API를 설계합니다.
3. DynamoDB 테이블을 사용하여 데이터를 가져오고 스키마의 유형과 필드를 생성합니다.
4. AWS AppSync의 WebSocket 기능과 Pub/Sub 아키텍처를 사용하여 실시간 API를 개발합니다.
5. 기존 GraphQL API(소스 API)를 사용하여 병합된 API에 연결합니다.

> **Note**
>
> 고급 도구로 작업하기 전에 스키마 설계 섹션을 검토하는 것이 좋습니다. 이 안내서에서는 AWS AppSync에서 더 복잡한 애플리케이션을 빌드하기 위해 개념적으로 사용할 수 있는 보다 간단한 예를 설명합니다.

또한 AWS AppSync는 GraphQL API를 생성하기 위한 몇 가지 콘솔 외 옵션을 지원합니다. 다음이 포함됩니다.

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

다음 예제는 사전 정의된 모델 및 DynamoDB를 사용하여 GraphQL API의 기본 구성 요소를 생성하는 방법을 보여 줍니다.

주제

- 1단계: 스키마 시작
- 2단계: 콘솔 둘러보기
- 3단계: GraphQL 변형을 사용하여 데이터 추가
- 4단계: GraphQL 쿼리를 사용하여 데이터 가져오기
- 보충 섹션

1단계: 스키마 시작

이 예제에서는 사용자가 ## ### 또는 ### ##과 같은 집안일 알림을 위해 Todo 항목을 만들 수 있게 하는 Todo API를 생성합니다. 이 API는 DynamoDB 테이블에서 상태가 지속되는 GraphQL 작업 사용 방법을 보여 줍니다.

개념적으로 첫 번째 GraphQL API를 만드는 데는 세 가지 주요 단계가 있습니다. 스키마(유형 및 필드)를 정의하고 데이터 원본을 필드에 연결한 다음 비즈니스 로직을 처리하는 해석기를 작성해야 합니다. 하지만 콘솔 환경에 따라 순서는 달라집니다. 먼저 데이터 원본이 스키마와 상호 작용하는 방식을 정의한 다음 스키마와 해석기를 정의하겠습니다.

GraphQL API 만들기

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
2. 대시보드에서 API 생성을 선택합니다.
3. GraphQL API가 선택된 상태에서 처음부터 설계를 선택합니다. 그리고 다음(Next)을 선택합니다.
4. API 이름의 경우 미리 채워진 이름을 **`Todo API`**로 변경한 후 다음을 선택합니다.

   > **Note**
   > 여기에는 다른 옵션도 있지만, 이 예제에서는 사용하지 않겠습니다.

5. GraphQL 리소스 지정 섹션에서 다음을 수행합니다.

   a. 지금 DynamoDB 테이블을 기반으로 유형 생성을 선택합니다.

      > **Note**
      > 즉, 데이터 원본으로 연결할 새 DynamoDB 테이블을 생성한다는 의미입니다.

   b. 모델 이름 필드에 **`Todo`**를 입력합니다.

      > **Note**
      > 첫 번째 요구 사항은 스키마를 정의하는 것입니다. 이 모델 이름은 유형 이름이 되므로, 실제로 수행하는 작업은 스키마에 존재하게 될 `Todo`라는 `type`을 만드는 것입니다.
      >
      > ```
      > type Todo {}
      > ```

   c. 필드에서 다음 작업을 수행합니다.

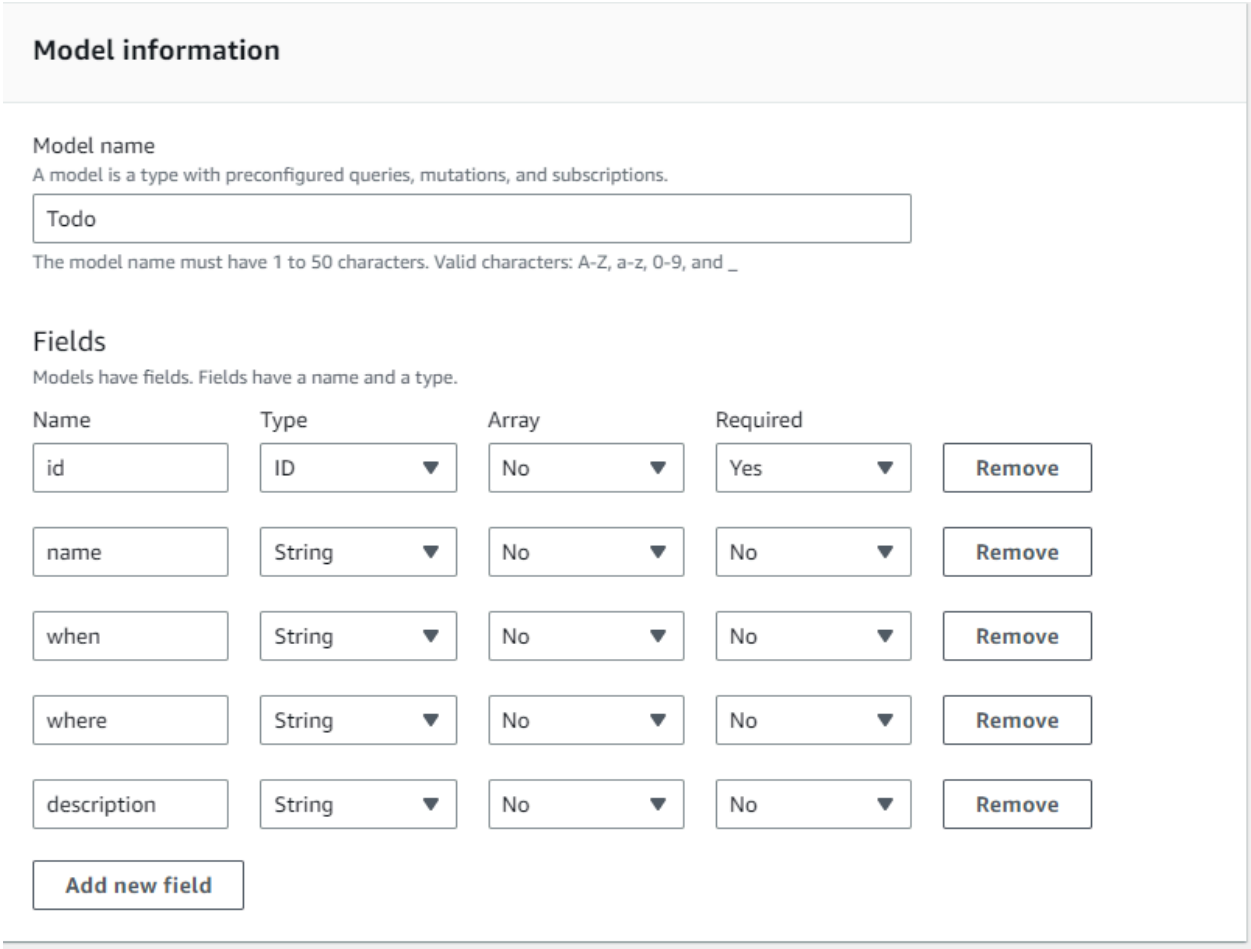
      i. 이름이 **`id`**, 유형이 `ID`, 필수 여부가 `Yes`로 설정된 필드를 만듭니다.

         > **Note**
         > 이러한 필드는 `Todo` 유형 범위 내에 존재하게 됩니다. 여기서 필드 이름은 `ID!` 유형의 `id`라고 불리게 됩니다.

```
type Todo {
 id: ID!
}
```

AWS AppSync는 다양한 사용 사례를 위한 여러 스칼라 값을 지원합니다.

ii. 새 필드 추가를 사용하여 Name 값이 **name**, **when**, **where**, **description**으로 설정된 필드 4개를 추가로 생성합니다. 해당 Type 값은 String이며, Array 및 Required 값은 모두 No로 설정됩니다. 형식은 다음과 같습니다.



Note

전체 유형과 해당 필드는 다음과 같습니다.

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

```
  when: String
  where: String
  description: String
}
```

미리 정의된 이 모델을 사용하여 스키마를 생성하므로 데이터 원본을 쉽게 채울 수 있도록 `create`, `delete`, `update`와 같은 유형에 따라 여러 개의 표준 문안 변형이 스키마에도 채워집니다.

d. 모델 테이블 구성에서 테이블 이름(예: **`TodoAPITable`**)을 입력합니다. 프라이머리 키를 `id`로 설정합니다.

> **Note**
>
> 기본적으로 *`TodoAPITable`*이라는 새로운 DynamoDB 테이블을 만들고 있으며, 이 테이블은 API에 프라이머리 데이터 원본으로 연결됩니다. 프라이머리 키는 이에 앞서 정의한 필수 `id` 필드로 설정되어 있습니다. 참고로 이 새 테이블은 비어 있으며 파티션 키 외에는 아무것도 들어 있지 않습니다.

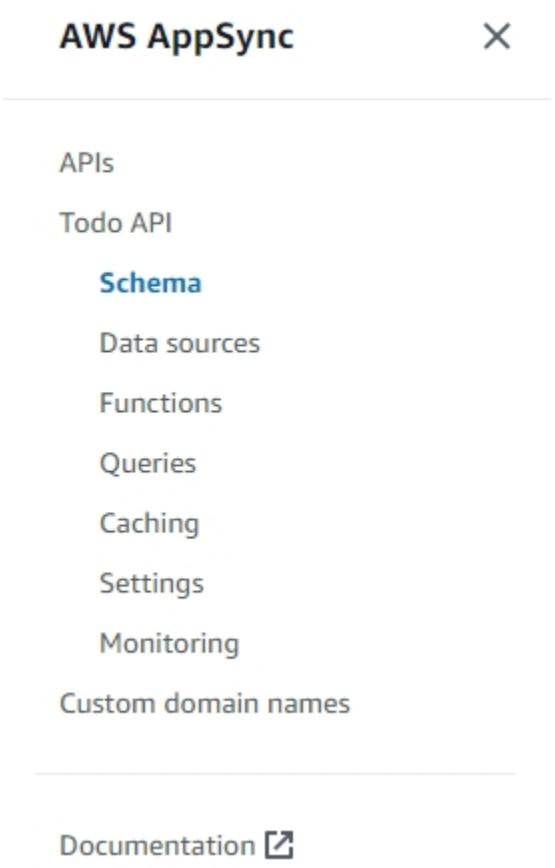
e. 다음(Next)을 선택합니다.

6. 변경 사항을 검토한 다음 API 생성을 선택합니다. AWS AppSync 서비스가 API 생성을 완료할 때까지 잠시 기다립니다.

스키마와 DynamoDB 데이터 원본을 사용하여 GraphQL API를 성공적으로 생성했습니다. 위의 단계를 요약하자면 완전히 새로운 GraphQL API를 만들어 보았습니다. API 이름을 정의한 다음 첫 번째 유형을 추가하여 스키마 정의를 추가했습니다. 유형과 필드를 정의한 다음 데이터가 없는 새로운 DynamoDB 테이블을 생성하여 필드 중 하나에 데이터 원본을 연결했습니다.

2단계: 콘솔 둘러보기

DynamoDB 테이블에 데이터를 추가하기 전에 AWS AppSync 콘솔 환경의 기본 기능을 검토해야 합니다. 페이지 왼쪽의 AWS AppSync 콘솔 탭을 통해 사용자는 AWS AppSync에서 제공하는 주요 구성 요소 또는 구성 옵션을 쉽게 탐색할 수 있습니다.



스키마 디자이너

방금 생성한 스키마를 보려면 스키마를 선택합니다. 스키마의 내용을 검토해 보면 개발 프로세스를 간소화하는 데 도움을 주기 위한 다양한 작업이 이미 로드되어 있음을 알 수 있습니다. 스키마 편집기에서 코드를 스크롤하다 보면 이전 섹션에서 정의한 모델에 도달하게 됩니다.

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

해당 모델은 스키마 전체에서 사용되는 기본 유형이 되었습니다. 이 유형에서 자동으로 생성된 변형을 사용하여 데이터 원본에 데이터를 추가하겠습니다.

다음은 스키마 편집기에 대한 몇 가지 추가 팁과 정보입니다.

1. 코드 편집기에는 앱을 작성할 때 사용할 수 있는 linting 및 오류 검사 기능이 있습니다.
2. 콘솔 오른쪽에는 생성된 GraphQL 형식과 다른 최상위 형식(예: 쿼리)에 대한 해석기가 표시됩니다.
3. 스키마에 새 형식을 추가하면(예: `type User {...}`) AWS AppSync 프로비저닝 DynamoDB 리소스가 생길 수 있습니다. 이러한 리소스에는 적절한 기본 키, 정렬 키와 GraphQL 데이터 액세스 패턴과 가장 잘 맞는 인덱스 설계가 포함됩니다. 상단에서 리소스 생성을 선택하고 메뉴에서 이러한 사용자 정의 형식 중 하나를 선택하면 스키마 설계에서 다른 필드 옵션을 선택할 수 있습니다. 이 내용은 스키마 설계 섹션에서 다루겠습니다.

해석기 구성

스키마 디자이너의 해석기 섹션에는 스키마의 모든 유형과 필드가 포함되어 있습니다. 필드 목록을 스크롤하다 보면 연결을 선택하여 특정 필드에 해석기를 연결할 수 있음을 알 수 있습니다. 이렇게 하면 해석기 코드를 작성할 수 있는 코드 편집기가 열립니다. AWS AppSync는 VTL과 JavaScript 런타임을 모두 지원하며, 페이지 상단에서 작업을 선택하고 런타임 업데이트를 선택하여 변경할 수 있습니다. 페이지 하단에서는 여러 작업을 순차적으로 실행하는 함수를 만들 수도 있습니다. 그러나 해석기는 상급 주제이므로 이 섹션에서는 다루지 않겠습니다.

데이터 원본

DynamoDB 테이블을 보려면 데이터 원본을 선택합니다. `Resource` 옵션(사용 가능한 경우)을 선택하면 데이터 원본의 구성을 볼 수 있습니다. 이 예에서는 DynamoDB 콘솔로 연결됩니다. 여기에서 데이터를 편집할 수 있습니다. 데이터 원본을 선택한 다음 편집을 선택하여 일부 데이터를 직접 편집할 수도 있습니다. 데이터 원본을 삭제해야 하는 경우 데이터 원본을 선택한 다음 삭제를 선택하면 됩니다. 마지막으로, 데이터 원본 생성을 선택한 다음 이름과 유형을 구성하여 새 데이터 원본을 만들 수 있습니다. 참고로 이 옵션은 AWS AppSync 서비스를 기존 리소스에 연결하기 위한 것입니다. AWS AppSync에서 리소스를 인식하려면 관련 서비스를 사용하여 계정에 리소스를 생성해야 합니다.

쿼리

쿼리와 변형을 보려면 쿼리를 선택합니다. 모델을 사용하여 GraphQL API를 만들었을 때 AWS AppSync는 테스트 목적으로 일부 헬퍼 변형과 쿼리를 자동으로 생성했습니다. 쿼리 편집기의 왼쪽에는 탐색기가 있습니다. 이 목록은 모든 변형과 쿼리를 보여 줍니다. 여기에서 사용하려는 작업과 필드는 이름 값을 클릭하여 손쉽게 활성화할 수 있습니다. 이렇게 하면 편집기의 중앙 부분에 코드가 자동으로 표시됩니다. 여기에서 값을 수정하여 변형과 쿼리를 편집할 수 있습니다. 편집기 하단에는 작업의 입력 변수에 대한 필드 값을 입력할 수 있는 쿼리 변수 편집기가 있습니다. 편집기 상단에서 실행을 선택하면 실행할 쿼리/변형을 선택할 수 있는 드롭다운 목록이 나타납니다. 이 실행의 결과물은 페이지 오른쪽에 표시됩니다. 상단의 탐색기 섹션으로 돌아가서 작업(쿼리, 변형, 구독)을 선택한 다음 + 기호

를 선택하여 해당 특정 작업의 새 인스턴스를 추가할 수 있습니다. 페이지 상단에는 쿼리 실행을 위한 권한 부여 모드가 포함된 또 다른 드롭다운 목록이 있습니다. 그러나 이 섹션에서는 해당 기능을 다루지 않습니다(자세한 내용은 보안 참조).

설정

GraphQL API의 일부 구성 옵션을 보려면 설정을 선택합니다. 여기에서 로깅, 추적, 웹 애플리케이션 방화벽 기능과 같은 일부 옵션을 활성화할 수 있습니다. 또한 새 권한 부여 모드를 추가하여 외부로의 원치 않는 유출로부터 데이터를 보호할 수도 있습니다. 그러나 이러한 옵션은 고급 옵션이므로 이 섹션에서는 다루지 않습니다.

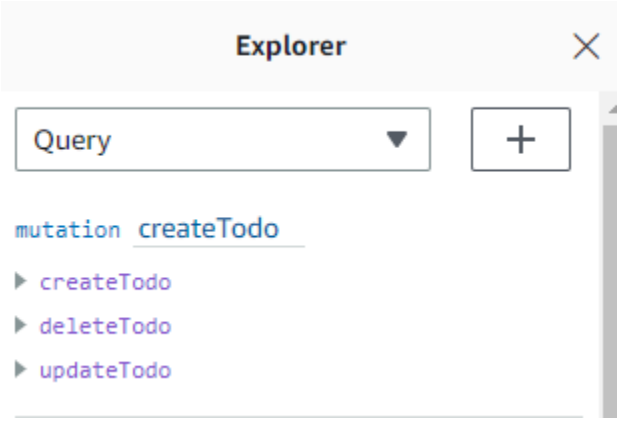
> **Note**
>
> 기본 권한 부여 모드인 `API_KEY`에서는 API 키를 사용하여 애플리케이션을 테스트합니다. 이는 새로 생성된 모든 GraphQL API에 제공되는 기본 권한 부여입니다. 프로덕션에는 다른 방법을 사용하는 것이 좋습니다. 이 섹션의 예제에서는 API 키만 사용합니다. 지원되는 권한 부여 방법에 대한 자세한 내용은 보안을 참조하세요.

3단계: GraphQL 변형을 사용하여 데이터 추가

다음 단계는 GraphQL 변형을 사용하여 빈 DynamoDB 테이블에 데이터를 추가하는 것입니다. 변형은 GraphQL의 필수 작업 유형 중 하나입니다. 스키마에 정의되어 있으며 데이터 원본의 데이터를 조작할 수 있습니다. REST API의 경우에는 `PUT` 또는 `POST`와 같은 작업과 매우 유사합니다.

데이터 원본에 데이터 추가

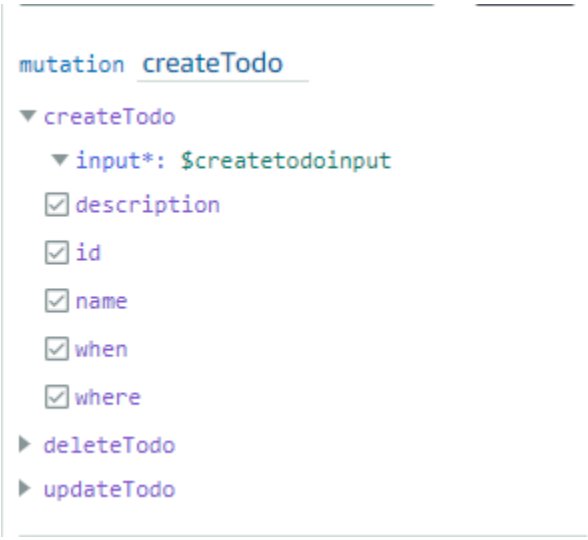
1. 아직 수행하지 않았다면 AWS Management Console에 로그인하고 AppSync 콘솔을 엽니다.
2. 테이블에서 API를 선택합니다.
3. 왼쪽 탭에서 쿼리를 선택합니다.
4. 테이블 왼쪽의 탐색기 탭에는 쿼리 편집기에 이미 정의된 여러 변형 및 쿼리가 표시될 수 있습니다.



> **Note**
>
> 이 변형은 실제로 스키마에 `Mutation` 유형으로 존재합니다. 다음과 같은 코드가 있습니다.
>
> ```
> type Mutation {
>  createTodo(input: CreateTodoInput!): Todo
>  updateTodo(input: UpdateTodoInput!): Todo
>  deleteTodo(input: DeleteTodoInput!): Todo
> }
> ```
>
> 보는 것과 같이 여기에서의 작업은 쿼리 편집기 내부의 작업과 유사합니다.

AWS AppSync는 앞서 정의한 모델에서 이러한 작업을 자동으로 생성했습니다. 이 예제에서는 `createTodo` 변형을 사용하여 *`TodoAPITable`* 테이블에 항목을 추가합니다.

5. `createTodo` 변형 아래에서 `createTodo` 작업을 확장하여 선택합니다.



위 그림과 같이 모든 필드의 확인란을 활성화합니다.

> **Note**
>
> 여기에 표시되는 특성은 변형의 여러 수정 가능한 요소입니다. `input`은 `createTodo`의 파라미터로 생각할 수 있습니다. 확인란이 있는 다양한 옵션은 작업이 수행된 후 응답에 반환될 필드입니다.

6. 화면 중앙의 코드 편집기에서 작업이 `createTodo` 변형 아래에 표시되는 것을 확인할 수 있습니다.

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

> **Note**
>
> 이 스니펫을 제대로 설명하려면 스키마 코드도 살펴봐야 합니다. `mutation createTodo($createtodoinput: CreateTodoInput!){}` 선언은 작업 중 하나인 `createTodo`가 있는 변형입니다. 전체 변형은 스키마에 있습니다.

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

편집기의 변형 선언으로 돌아가면 파라미터는 CreateTodoInput의 필수 입력 유형이 $createtodoinput이라는 객체입니다. 참고로 CreateTodoInput(그리고 변형의 모든 입력)도 스키마에 정의되어 있습니다. 예를 들어, 다음은 CreateTodoInput에 대한 보일러플레이트 코드입니다.

```
input CreateTodoInput {
 name: String
 when: String
 where: String
 description: String
}
```

여기에는 모델에서 정의한 필드인 name, when, where 및 description이 포함됩니다. 편집기 코드로 돌아가면 createTodo(input: $createtodoinput) {}에서 입력을 $createtodoinput으로 선언하는데, 이는 변형 선언에도 사용되었습니다. 이렇게 하는 이유는 GraphQL이 제공된 유형에 대해 입력을 검증할 수 있게 하고 올바른 입력과 함께 사용되도록 하기 위함입니다.
편집기 코드의 마지막 부분에는 작업 수행 후 응답에 반환될 필드가 표시됩니다.

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

이 편집기 아래의 쿼리 변수 탭에는 다음 데이터를 포함할 수 있는 일반 createtodoinput 객체가 있습니다.

```
{
```

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

> **Note**
>
> 여기서 앞서 언급한 입력에 대한 값을 할당합니다.
>
> ```
> input CreateTodoInput {
>  name: String
>  when: String
>  where: String
>  description: String
> }
> ```

DynamoDB 테이블에 넣으려는 정보를 추가하여 `createtodoinput`을 변경합니다. 이 경우에는 미리 알림으로 사용할 몇 가지 Todo 항목을 만들려고 했습니다.

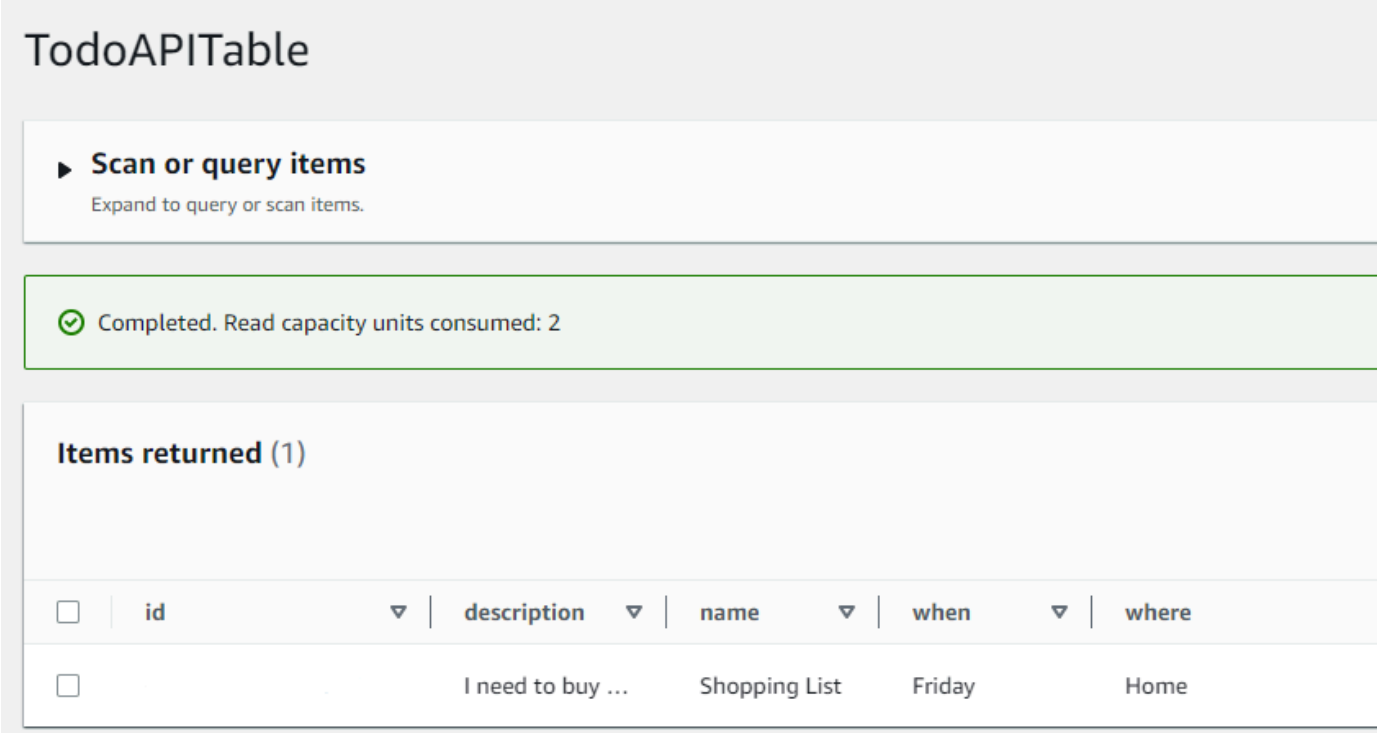
```
{
  "createtodoinput": {
    "name": "Shopping List",
    "when": "Friday",
    "where": "Home",
    "description": "I need to buy eggs"
  }
}
```

7. 편집기 상단에서 실행을 선택합니다. 드롭다운 목록에서 createTodo를 선택합니다. 편집기의 오른쪽에 응답이 표시됩니다. 다음과 같을 것입니다.

```
{
  "data": {
    "createTodo": {
      "where": "Home",
      "when": "Friday",
      "name": "Shopping List",
```

```
      "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
      "description": "I need to buy eggs"
    }
  }
}
```

DynamoDB 서비스로 이동하면 이제 다음 정보와 함께 데이터 원본의 항목이 표시됩니다.



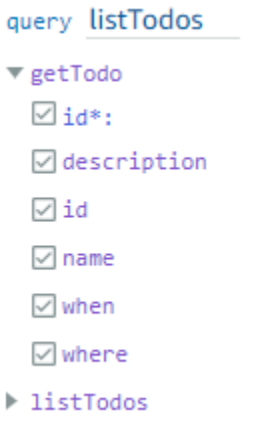
작업을 요약하자면 GraphQL 엔진이 레코드를 파싱하고 해석기가 이를 Amazon DynamoDB 테이블에 삽입했습니다. 다시 DynamoDB 콘솔에서 이를 확인할 수 있습니다. `id` 값에 전달할 필요는 없습니다. `id`가 생성되어 결과에 반환됩니다. 이 예제에서는 DynamoDB 리소스에 설정된 파티션 키에 대해 GraphQL해석기의 `autoId()` 함수를 사용했기 때문입니다. 해석기를 빌드하는 방법에 대해서는 다른 섹션에서 설명하겠습니다. 반환된 `id` 값을 기록해 두세요. 다음 섹션에서 GraphQL 쿼리로 데이터를 검색하는 데 사용할 것입니다.

4단계: GraphQL 쿼리를 사용하여 데이터 가져오기

이제 데이터베이스에 레코드가 있으므로 쿼리를 실행하면 결과를 얻게 됩니다. 쿼리는 GraphQL의 다른 필수 작업 중 하나입니다. 데이터 원본에서 정보를 파싱하고 검색하는 데 사용됩니다. REST API의 경우 `GET` 작업과 유사합니다. GraphQL 쿼리의 주요 장점은 적시에 관련 데이터를 가져올 수 있도록 애플리케이션의 정확한 데이터 요구 사항을 지정할 수 있다는 것입니다.

데이터 원본 쿼리

1. 아직 수행하지 않았다면 AWS Management Console에 로그인하고 AppSync 콘솔을 엽니다.
2. 테이블에서 API를 선택합니다.
3. 왼쪽 탭에서 쿼리를 선택합니다.
4. 테이블 왼쪽의 탐색기 탭에서 `query listTodos` 아래의 `getTodo` 작업을 확장합니다.



5. 코드 편집기에 작업 코드가 표시됩니다.

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

`(id:"")`에서 변형 작업에서 결과에 저장한 값을 입력합니다. 이 예에서는 다음과 같습니다.

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

6. 실행을 선택하고 listTodos를 선택합니다. 편집기의 오른쪽에 결과가 표시됩니다. 이 예에서는 다음과 같습니다.

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

> **Note**
>
> 쿼리는 사용자가 지정한 필드만 반환합니다. 필요하지 않은 필드는 반환 필드에서 삭제하여 선택 취소할 수 있습니다.
>
> ```
> {
>     description
>     id
>     name
>     when
>     where
>   }
> ```
>
> 삭제하려는 필드 옆의 탐색기 탭에서 확인란을 선택 해제할 수도 있습니다.

7. 데이터 원본에 항목을 만드는 단계를 반복한 다음 `listTodos` 작업과 함께 쿼리 단계를 반복하여 `listTodos` 작업을 시도할 수도 있습니다. 다음은 두 번째 작업을 추가한 예입니다.

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

```
}
```

listTodos 작업을 호출하면 이전 항목과 새 항목이 모두 반환됩니다.

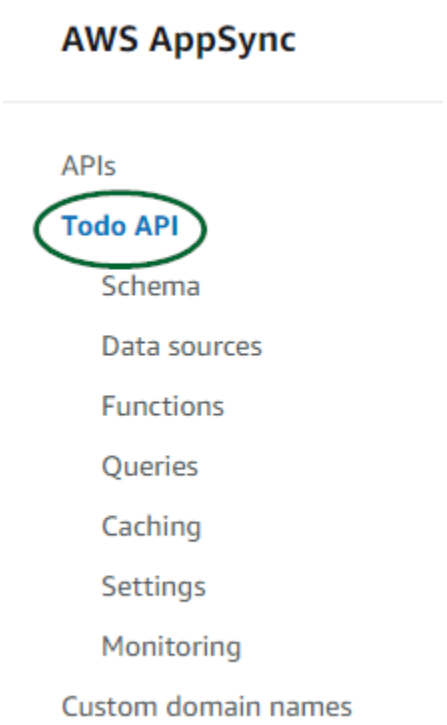
```
{
  "data": {
    "listTodos": {
      "items": [
        {
          "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
          "name": "Shopping List",
          "when": "Friday",
          "where": "Home",
          "description": "I need to buy eggs"
        },
        {
          "id": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",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

보충 섹션

이 섹션은 고급 AWS AppSync 주제에 대한 참조입니다. 다른 작업을 하기 전에 보충 자료 섹션을 먼저 읽어보는 것이 좋습니다.

통합

콘솔 탭에서 API 이름을 선택하면 통합 페이지가 나타납니다.



이 페이지에는 API 설정 단계가 요약되어 있으며 클라이언트 애플리케이션을 빌드하기 위한 다음 단계를 간략하게 설명합니다. 애플리케이션과 통합 섹션은 구성 및 코드 생성을 통해 API와 iOS, Android 및 JavaScript 애플리케이션을 연결하는 프로세스를 자동화하는 AWS Amplify 도구 체인 사용에 대한 세부 정보를 제공합니다. Amplify 도구 체인은 GraphQL 프로비저닝 및 CI/CD에 대한 워크플로를 포함하여 로컬 워크스테이션에서의 프로젝트 빌드에 대한 전체 지원을 제공합니다.

또한 클라이언트 샘플 섹션에는 엔드 투 엔드 경험을 테스트하기 위한 샘플 클라이언트 애플리케이션(예: JavaScript, iOS, Android 이 나열되어 있습니다. 시작하려면 이러한 샘플 및 필수 정보(예: 엔드포인트 URL)가 포함된 구성 파일을 복제해 다운로드합니다. AWS Amplify 도구 체인 페이지의 지침에 따라 앱을 실행합니다.

보충 자료

- GraphQL API 설계 - 이 가이드에서는 데이터 원본이나 해석기 없이 빈 스키마를 사용하여 GraphQL을 생성하는 방법을 포괄적으로 설명합니다.

GraphQL API 설계

AWS AppSync에서 콘솔 환경을 사용하여 GraphQL API를 생성할 수 있습니다. 샘플 스키마 시작 섹션에서 이 내용을 간략히 살펴보았습니다. 하지만 이 가이드에는 AWS AppSync에서 활용할 수 있는 옵션 및 구성의 전체 카탈로그가 나와 있지 않았습니다.

콘솔에서 GraphQL API를 생성하는 경우 살펴보아야 할 몇 가지 옵션이 있습니다. 샘플 스키마 시작 가이드에서는 사전 정의된 모델에서 API를 생성하는 방법을 알아보았습니다. 다음 섹션에서는 AWS AppSync에서 GraphQL API를 생성하기 위한 나머지 옵션과 구성을 안내합니다.

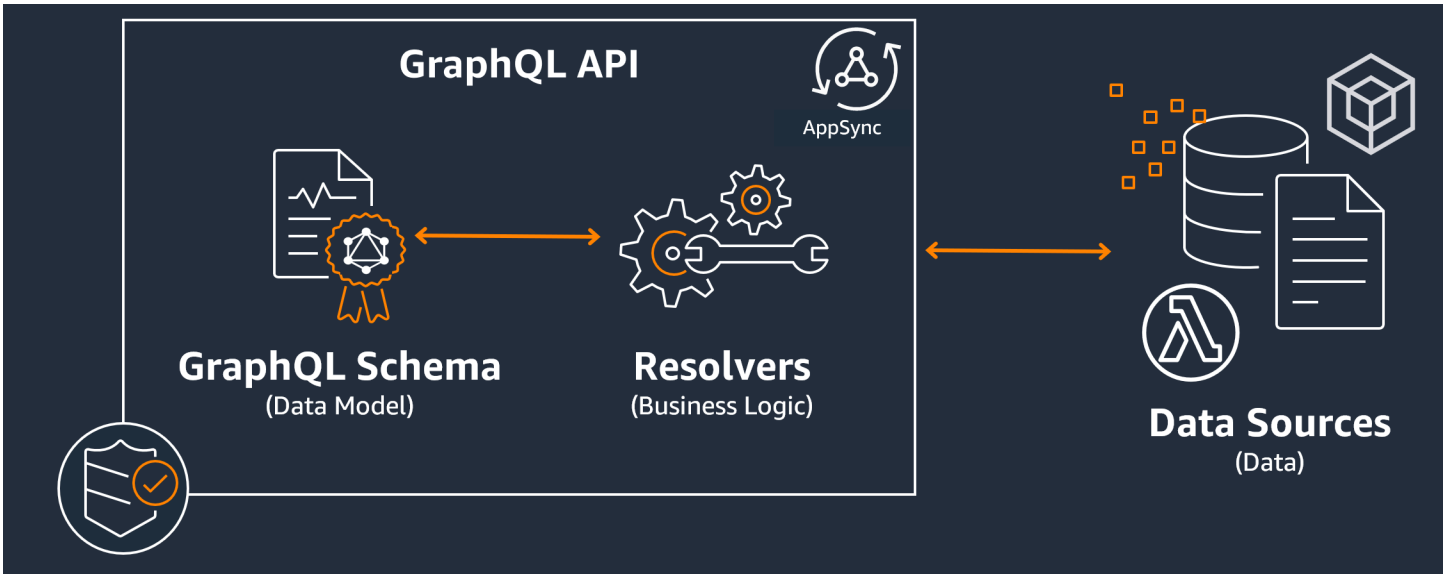
이 섹션에서는 다음 개념을 검토하게 됩니다.

1. Blank APIs or imports: 이 가이드는 GraphQL API를 만들기 위한 전체 생성 프로세스를 안내합니다. 모델 없이 빈 템플릿에서 GraphQL을 생성하고, 스키마의 데이터 소스를 구성하고, 필드에 첫 번째 해석기를 추가하는 방법을 알아봅니다.
2. Real-time data: 이 가이드에서는 AWS AppSync의 WebSocket 엔진으로 API를 생성하는 데 사용할 수 있는 옵션을 보여 줍니다.
3. Merged APIs: 이 가이드에서는 여러 기존 GraphQL API의 데이터를 연결하고 병합하여 새 GraphQL API를 만드는 방법을 보여줍니다.
4. the section called “RDS 내부 검사”: 이 가이드에서는 데이터 API를 사용하여 Amazon RDS 테이블을 통합하는 방법을 보여줍니다.

GraphQL API 구조화(빈 API 또는 가져온 API)

빈 템플릿에서 GraphQL API를 생성하기 전에 GraphQL과 관련된 개념을 검토하는 것이 도움이 될 것입니다. GraphQL API에는 세 가지 기본 구성 요소가 있습니다.

1. 스키마는 데이터의 형태와 정의를 포함하는 파일입니다. 클라이언트가 GraphQL 서비스에 요청을 보내면 반환되는 데이터는 스키마의 사양을 따릅니다. 자세한 내용은 스키마 섹션을 참조하세요.
2. 데이터 소스는 스키마에 연결되어 있습니다. 요청이 만들어지면 여기에서 데이터를 검색하고 수정합니다. 자세한 내용은 Data sources 섹션을 참조하세요.
3. 해석기는 스키마와 데이터 소스 사이에 위치합니다. 요청이 만들어지면 해석기는 소스의 데이터에 대해 작업을 수행한 다음 응답으로 결과를 반환합니다. 자세한 내용은 Resolvers 섹션을 참조하세요.



AWS AppSync에서는 스키마와 해석기의 코드를 생성, 편집 및 저장할 수 있도록 하여 API를 관리합니다. 데이터 소스는 데이터베이스, DynamoDB 테이블, Lambda 함수와 같은 외부 리포지토리에서 가져옵니다. AWS 서비스를 사용하여 데이터를 저장하거나 저장할 계획이라면 AWS 계정의 데이터를 GraphQL API에 연결할 때 AWS AppSync에서 거의 끊김 없는 경험을 제공합니다.

다음 섹션에서는 AWS AppSync 서비스를 사용하여 이러한 각 구성 요소를 만드는 방법을 알아봅니다.

주제

- 1단계: 스키마 설계
- 2단계: 데이터 원본 연결
- 3단계: 해석기 구성
- 4단계: API 사용: CDK 예제

1단계: 스키마 설계

GraphQL 스키마는 모든 GraphQL 서버 구현의 기초입니다. 각 GraphQL API는 요청의 데이터가 채워지는 방식을 설명하는 유형과 필드를 포함하는 단일 스키마로 정의됩니다. API를 통한 데이터 흐름과 수행되는 작업은 스키마를 기준으로 검증되어야 합니다.

GraphQL 유형 시스템은 일반적으로 GraphQL 서버의 기능을 기술하고 쿼리가 유효한지 확인하는 데 사용됩니다. 서버의 유형 시스템은 종종 해당 서버의 스키마라고 하며 다양한 객체 유형, 스칼라 유형, 입력 유형 등으로 구성될 수 있습니다. GraphQL은 선언적이면서 강력한 유형입니다. 즉, 유형은 런타임에 잘 정의되고 지정된 값만 반환합니다.

AWS AppSync를 사용하면 GraphQL 스키마를 정의하고 구성할 수 있습니다. 다음 섹션에서는 AWS AppSync의 서비스를 사용하여 GraphQL 스키마를 처음부터 생성하는 방법을 설명합니다.

GraphQL 스키마 구조화

> **Tip**
>
> 계속하기 전에 스키마 섹션을 검토하는 것이 좋습니다.

GraphQL은 API 서비스를 구현하기 위한 강력한 도구입니다. GraphQL의 웹 사이트에서는 GraphQL을 다음과 같이 기술합니다.

"GraphQL은 API용 쿼리 언어이자 기존 데이터로 이러한 쿼리를 수행하기 위한 런타임입니다. GraphQL은 API의 데이터에 대한 완전하고 이해하기 쉬운 설명을 제공하고, 클라이언트가 필요한 사항과 더 이상 필요하지 않은 사항을 정확히 요청할 수 있는 기능을 제공하며, 시간이 지남에 따라 API를 더 쉽게 발전시킬 수 있게 하고, 강력한 개발자 도구를 사용할 수 있도록 합니다."

이 섹션에서는 GraphQL 구현의 맨 첫 부분인 스키마를 다룹니다. 위의 인용문에 따르면 스키마는 'API의 데이터에 대한 완전하고 이해하기 쉬운 설명을 제공'하는 역할을 합니다. 즉, GraphQL 스키마는 서비스의 데이터, 작업 및 작업 간의 관계를 텍스트로 표현한 것입니다. 스키마는 GraphQL 서비스 구현을 위한 기본 진입점으로 간주됩니다. 당연하게도 스키마는 프로젝트에서 가장 먼저 만드는 항목 중 하나인 경우가 많습니다. 계속하기 전에 스키마 섹션을 검토하는 것이 좋습니다.

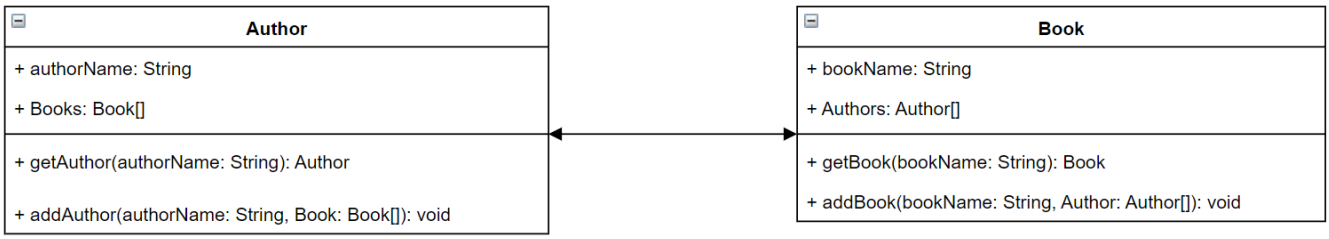
스키마 섹션을 인용하자면 GraphQL 스키마는 스키마 정의 언어(SDL)로 작성됩니다. SDL은 구조가 확립된 유형과 필드로 구성되어 있습니다.

- 유형: 유형은 GraphQL이 데이터의 형태와 동작을 정의하는 방식입니다. GraphQL은 이 섹션의 뒷부분에서 설명할 여러 유형을 지원합니다. 스키마에 정의된 각 유형에는 고유한 범위가 포함됩니다. 범위 내에는 GraphQL 서비스에서 사용할 값이나 로직을 포함할 수 있는 하나 이상의 필드가 있습니다. 유형은 다양한 역할을 수행하며, 가장 일반적으로는 객체 또는 스칼라(기본 값 유형) 역할을 합니다.
- 필드: 필드는 유형의 범위 내에 존재하며 GraphQL 서비스에서 요청한 값을 보유합니다. 이는 다른 프로그래밍 언어의 변수와 매우 유사합니다. 필드에 정의하는 데이터의 형태에 따라 요청/응답 작업에서 데이터가 구조화되는 방식이 결정됩니다. 이를 통해 개발자는 서비스의 백엔드가 구현되는 방식을 모르더라도 어떤 결과가 반환될지 예측할 수 있습니다.

가장 간단한 스키마에는 다음과 같은 세 가지 데이터 범주가 포함됩니다.

1. 스키마 루트: 루트는 스키마의 진입점을 정의합니다. 또한 추가, 삭제, 수정과 같은 데이터에 대한 일부 작업을 수행할 필드를 가리킵니다.
2. 유형: 데이터의 형태를 나타내는 데 사용되는 기본 유형입니다. 주로 정의된 특성을 가진 대상의 객체나 추상적 표현으로 생각할 수 있습니다. 예를 들어, 데이터베이스에 있는 사람을 나타내는 `Person` 객체를 만들 수 있습니다. 각 개인의 특성은 `Person` 내에 필드로 정의됩니다. 개인의 이름, 나이, 직업, 주소 등 무엇이든 될 수 있습니다.
3. 특수 객체 유형: 스키마의 작업 동작을 정의하는 유형입니다. 각 특수 객체 유형은 스키마별로 한 번씩 정의됩니다. 먼저 스키마 루트에 배치된 다음 스키마 본문에 정의됩니다. 특수 객체 유형의 각 필드는 해석기가 구현할 단일 작업을 정의합니다.

이 개념을 잘 이해하기 위해 저자와 저자가 쓴 책을 저장하는 서비스를 만든다고 가정해 보세요. 저자마다 이름이 있고 저술한 책도 여러 권 있습니다. 각 책에는 이름과 관련 저자 목록이 있습니다. 또한 책과 저자를 추가하거나 검색할 수 있는 기능도 필요합니다. 이 관계를 간단한 UML로 표현하면 다음과 같을 수 있습니다.



GraphQL에서 `Author` 및 `Book` 엔터티는 스키마의 서로 다른 두 가지 객체 유형을 나타냅니다.

```
type Author {
}

type Book {
}
```

`Author`는 `authorName`과 `Books`를 포함하는 반면, `Book`은 `bookName`과 `Authors`를 포함합니다. 이러한 유형은 유형 범위 내의 필드로 표시될 수 있습니다.

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

```
}

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

보는 것과 같이 유형 표현은 다이어그램과 매우 비슷합니다. 하지만 메서드의 경우 조금 더 까다로워집니다. 메서드는 몇 가지 특수 객체 유형 중 하나에 필드로 배치됩니다. 특수 객체 분류는 동작에 따라 결정됩니다. GraphQL에는 쿼리, 변형, 구독이라는 세 가지 기본 특수 객체 유형이 포함되어 있습니다. 자세한 내용은 특수 객체를 참조하세요.

getAuthor와 getBook 둘 다 데이터를 요청하므로 다음과 같은 Query 특수 객체 유형에 배치됩니다.

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

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

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

작업은 쿼리에 연결되며, 쿼리 자체는 스키마에 연결됩니다. 스키마 루트를 추가하면 특수 객체 유형(이 경우 Query)이 진입점 중 하나로 정의됩니다. 다음과 같은 schema 키워드를 사용하여 수행할 수 있습니다.

```
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
}
```

마지막 두 메서드를 살펴보면 addAuthor 및 addBook은 데이터베이스에 데이터를 추가하고 있으므로 Mutation 특수 객체 유형에 정의됩니다. 그러나 유형 페이지를 보면 객체를 직접 참조하는 입력은 엄밀히는 출력 유형이기 때문에 허용되지 않는다는 사실도 알 수 있습니다. 이 경우에는 Author 또는 Book을 사용할 수 없으므로 동일한 필드로 입력 유형을 만들어야 합니다. 이 예제에서는 AuthorInput 및 BookInput을 추가했는데, 둘 다 해당 유형의 동일한 필드를 수락합니다. 그런 다음 입력값을 파라미터로 사용하여 변형을 생성합니다.

```
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
}
```

방금 수행한 단계를 다시 살펴보겠습니다.

1. 엔터티를 나타내는 Book 및 Author 유형을 사용하여 스키마를 만들었습니다.
2. 엔터티의 특성이 포함된 필드를 추가했습니다.
3. 데이터베이스에서 이 정보를 검색하는 쿼리를 추가했습니다.
4. 데이터베이스의 데이터를 조작하기 위해 변형을 추가했습니다.
5. GraphQL의 규칙을 준수하기 위해 변형의 객체 파라미터를 대체하는 입력 유형을 추가했습니다.
6. GraphQL 구현이 루트 유형 위치를 이해할 수 있도록 루트 스키마에 쿼리와 변형을 추가했습니다.

보는 것과 같이 스키마를 생성하는 프로세스는 일반적으로 데이터 모델링(특히 데이터베이스 모델링)으로부터 많은 개념을 가져옵니다. 스키마는 원본의 데이터 형태와 맞는 것으로 생각할 수 있습니다. 또한 해석기가 구현할 모델 역할도 합니다. 다음 섹션에서는 다양한 AWS 지원 도구 및 서비스를 사용하여 스키마를 만드는 방법을 알아봅니다.

> **Note**
>
> 다음 섹션의 예제는 실제 애플리케이션에서 실행하도록 설계된 것이 아닙니다. 직접 애플리케이션을 빌드할 수 있도록 명령을 보여 주기 위한 용도로만 제공됩니다.

스키마 생성

스키마는 `schema.graphql`이라는 파일에 위치하게 됩니다. AWS AppSync를 사용하면 다양한 방법을 사용하여 GraphQL API에 대한 새로운 스키마를 생성할 수 있습니다. 이 예에서는 빈 스키마와 함께 빈 API를 생성해 보겠습니다.

Console

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.

   a. 대시보드에서 API 생성을 선택합니다.

   b. API 옵션에서 GraphQL API, 처음부터 설계, 다음을 차례로 선택합니다.

      i. API 이름의 경우 미리 채워진 이름을 애플리케이션에 필요한 이름으로 변경합니다.

      ii. 연락처 세부 정보에는 API 관리자를 식별할 연락처를 입력하면 됩니다. 이 필드는 선택 사항입니다.

      iii. 프라이빗 API 구성에서 프라이빗 API 기능을 활성화할 수 있습니다. 프라이빗 API는 구성된 VPC 엔드포인트(VPCE)에서만 액세스할 수 있습니다. 자세한 내용은 프라이빗 API를 참조하세요.

      이 예에서는 이 기능을 활성화하지 않는 것이 좋습니다. 입력 내용을 검토한 후 다음을 선택합니다.

   c. GraphQL 유형 생성에서 데이터 원본으로 사용할 DynamoDB 테이블을 생성할지, 아니면 이 단계를 건너뛰고 나중에 생성할지 선택할 수 있습니다.

   이 예제에서는 나중에 GraphQL 리소스 생성을 선택합니다. 별도의 섹션에서 리소스를 생성할 것입니다.

   d. 입력 내용을 검토한 다음 API 생성을 선택합니다.

2. 특정 API의 대시보드로 이동하게 됩니다. API 이름은 대시보드 상단에 표시되므로 알아볼 수 있습니다. 그렇지 않은 경우 사이드바에서 API를 선택한 다음 API 대시보드에서 API를 선택하면 됩니다.

   - API 이름 아래에 있는 사이드바에서 스키마를 선택합니다.

3. 스키마 편집기에서 `schema.graphql` 파일을 구성할 수 있습니다. 파일이 비어 있거나 모델에서 생성된 유형으로 채워져 있을 수 있습니다. 오른쪽에는 스키마 필드에 해석기를 연결할 수 있는 해석기 섹션이 있습니다. 이 섹션에서는 해석기는 살펴보지 않겠습니다.

CLI

> **Note**
>
> CLI를 사용하는 경우 서비스에서 리소스에 액세스하고 리소스를 생성할 수 있는 올바른 권한이 있어야 합니다. 서비스에 액세스해야 하는 관리자가 아닌 사용자를 위해 최소 권한 정

책을 설정할 수도 있습니다. AWS AppSync 정책에 대한 자세한 내용은 AWS AppSync에 대한 자격 증명 및 액세스 관리를 참조하세요.
또한 아직 읽어보지 않은 경우 먼저 콘솔 버전을 읽어보시기 바랍니다.

1. 아직 완료하지 않았다면 AWS CLI를 설치한 다음 구성을 추가합니다.
2. `create-graphql-api` 명령을 실행하여 GraphQL API 객체를 생성합니다.

   이 특정 명령에 대해 두 개의 파라미터를 입력해야 합니다.

   1. API의 `name`.
   2. `authentication-type` 또는 API에 액세스하는 데 사용되는 보안 인증 유형(IAM, OIDC 등)

   > **Note**
   > 필수로 구성해야 하지만 일반적으로 CLI 구성 값으로 기본 설정되는 `Region`과 같은 다른 파라미터도 있습니다.

   예를 들어 명령은 다음과 같을 수 있습니다.

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

   CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

```
{
    "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**
>
> 기존 스키마를 가져와 base-64 blob을 사용하여 AWS AppSync 서비스에 업로드하는 선택적 명령입니다. 이 예제에서는 이 명령을 사용하지 않을 것입니다.

`start-schema-creation` 명령을 실행합니다.

이 특정 명령에 대해 두 개의 파라미터를 입력해야 합니다.

1. 이전 단계의 `api-id`입니다.
2. `definition` 스키마는 base-64로 인코딩된 바이너리 blob입니다.

예를 들어 명령은 다음과 같을 수 있습니다.

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

출력이 반환됩니다.

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

이 명령은 처리 후 최종 출력을 반환하지 않습니다. 결과를 보려면 별도의 명령인 `get-schema-creation-status`를 사용해야 합니다. 참고로 이 두 명령은 비동기식이므로 스키마를 생성하는 중에도 출력 상태를 확인할 수 있습니다.

CDK

> **Tip**
>
> CDK를 사용하기 전에 CDK의 공식 설명서와 함께 AWS AppSync의 CDK 참조를 검토하는 것이 좋습니다.
> 아래 나열된 단계는 특정 리소스를 추가하는 데 사용되는 스니펫의 일반적인 예시만 보여줍니다. 프로덕션 코드에서는 이 예시가 올바르게 작동하는 솔루션이 아닙니다. 또한 이미 작동하는 앱을 가지고 있는 것으로 가정합니다.

1. CDK의 시작점은 약간 다릅니다. 이상적으로는 `schema.graphql` 파일이 이미 생성되어 있어야 합니다. `.graphql` 파일 확장자가 있는 새 파일을 생성하기만 하면 됩니다. 빈 파일이어도 괜찮습니다.
2. 대개는 사용 중인 서비스에 가져오기 지침을 추가해야 할 수 있습니다. 예를 들면 다음 형식을 따를 수 있습니다.

   ```
   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'
   ```

   GraphQL API를 추가하려면 스택 파일이 AWS AppSync 서비스를 가져와야 합니다.

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

   > **Note**
   >
   > 즉, appsync 키워드로 전체 서비스를 가져오게 됩니다. 앱에서 이 서비스를 사용하려면 AWS AppSync 구성이 `appsync.construct_name` 형식을 사용해야 합니다. 예를 들어, GraphQL API를 만들고 싶다면 new `appsync.GraphqlApi(args_go_here)`를 사용할 수 있습니다. 다음 단계에서는 이 과정을 설명합니다.

3. 가장 기본적인 GraphQL API에는 API의 name 및 schema 경로가 포함됩니다.

   ```
   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**
>
> 이 스니펫이 어떤 역할을 하는지 살펴보겠습니다. `api`의 범위 내에서 `appsync.GraphqlApi(scope: Construct, id: string, props: GraphqlApiProps)`를 호출하여 새로운 GraphQL API를 만들고 있습니다. 범위는 `this`이며, 현재 객체를 참조합니다. ID는 *`API_ID`*이며, 생성되었을 때 AWS CloudFormation에서 GraphQL API의 리소스 이름이 됩니다. `GraphqlApiProps`에는 GraphQL API의 `name` 및 `schema`가 포함되어 있습니다. `schema`는 `.graphql` 파일(*`schema_name.graphql`*)의 절대 경로(`__dirname`)를 검색하여 스키마(`SchemaFile.fromAsset`)를 생성합니다. 실제 시나리오에서는 스키마 파일이 CDK 앱 내에 있을 수 있습니다.
> GraphQL API에서 변경한 내용을 사용하려면 앱을 재배포해야 합니다.

스키마에 유형 추가

이제 스키마를 추가했으니 입력 유형과 출력 유형을 모두 추가할 수 있습니다. 참고로 여기에 있는 유형은 실제 코드에서 사용해서는 안 됩니다. 프로세스 이해를 돕기 위한 예시일 뿐입니다.

먼저 객체 유형을 생성하겠습니다. 실제 코드에서는 이러한 유형으로 시작할 필요가 없습니다. GraphQL의 규칙과 구문만 따르면 언제든지 원하는 유형을 만들 수 있습니다.

> **Note**
>
> 다음 몇 개의 섹션에서 스키마 편집기를 사용할 예정이니 계속 열어 두세요.

Console

- 유형 이름과 함께 `type` 키워드를 사용하여 객체 유형을 생성할 수 있습니다.

  ```
  type Type_Name_Goes_Here {}
  ```

  유형의 범위 내에 객체의 특성을 나타내는 필드를 추가할 수 있습니다.

  ```
  type Type_Name_Goes_Here {
  ```

```
  # Add fields here
}
```

다음은 그 예입니다:

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

> **Note**
>
> 이 단계에서는 필수 `id` 필드는 `ID`로, `title` 필드는 `String`으로, `date` 필드는 `AWSDateTime`으로 저장되는 일반 객체 유형을 추가했습니다. 유형 및 필드의 목록과 그 역할을 보려면 스키마를 참조하세요. 스칼라 목록과 그 역할을 보려면 유형 참조를 참조하세요.

CLI

> **Note**
>
> 아직 읽어보지 않은 경우 먼저 콘솔 버전을 읽어보시기 바랍니다.

- `create-type` 명령을 실행하여 객체 유형을 만들 수 있습니다.

  이 특정 명령에 대해 몇 가지 파라미터를 입력해야 합니다.

  1. API의 `api-id`.
  2. `definition` 또는 유형의 내용. 콘솔 예제에서는 다음과 같았습니다.

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

  3. 입력의 `format`. 이 예제에서는 `SDL`을 사용합니다.

예를 들어 명령은 다음과 같을 수 있습니다.

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

CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

```
{
    "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**
>
> 이 단계에서는 필수 `id` 필드는 `ID`로, `title` 필드는 `String`으로, `date` 필드는 `AWSDateTime`으로 저장되는 일반 객체 유형을 추가했습니다. 유형 및 필드의 목록과 그 역할을 보려면 스키마를 참조하세요. 스칼라 목록과 그 역할을 보려면 유형 참조를 참조하세요.
>
> 추가로 설명하자면, 정의를 직접 입력하는 것은 크기가 작은 유형에서는 효과적이지만, 크기가 크거나 여러 개인 유형에는 적합하지 않다는 사실을 확인하셨을 수 있습니다. `.graphql` 파일에 모든 유형을 추가한 다음 입력으로 전달하도록 선택할 수 있습니다.

CDK

> **Tip**
>
> CDK를 사용하기 전에 CDK의 공식 설명서와 함께 AWS AppSync의 CDK 참조를 검토하는 것이 좋습니다.

아래 나열된 단계는 특정 리소스를 추가하는 데 사용되는 스니펫의 일반적인 예시만 보여 줍니다. 프로덕션 코드에서는 이 예시가 올바르게 작동하는 솔루션이 아닙니다. 또한 이미 작동하는 앱을 가지고 있는 것으로 가정합니다.

유형을 추가하려면 `.graphql` 파일에 추가해야 합니다. 예를 들어 콘솔 예시에서는 다음과 같았습니다.

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

다른 파일처럼 스키마에 유형을 직접 추가할 수 있습니다.

> **Note**
>
> GraphQL API에서 변경한 내용을 사용하려면 앱을 재배포해야 합니다.

객체 유형에는 문자열 및 정수와 같은 스칼라 유형인 필드가 있습니다. AWS AppSync에서는 기본 GraphQL 스칼라 외에도 `AWSDateTime`과 같은 향상된 스칼라 유형을 사용할 수도 있습니다. 또한 느낌표로 끝나는 모든 필드는 필수 필드입니다.

특히 `ID` 스칼라 유형은 `String` 또는 `Int`인 고유한 식별자입니다. 자동 할당을 위해 해석기 코드에서 이러한 유형을 제어할 수 있습니다.

`Query`와 같은 특수 객체 유형과 위의 예와 같은 '일반' 객체 유형 간에는 모두 `type` 키워드를 사용하고 객체로 간주된다는 점에서 유사점이 있습니다. 하지만 특수 객체 유형(`Query`, `Mutation`, `Subscription`)의 경우 API의 진입점으로 노출되므로 동작은 크게 다릅니다. 또한 데이터보다는 작업의 형태를 잡는 데 더 중점을 둡니다. 자세한 내용은 쿼리 및 변형 유형을 참조하세요.

특수 객체 유형의 경우 다음 단계는 형상화된 데이터에 대한 작업을 수행하기 위해 하나 이상의 유형을 추가하는 것일 수 있습니다. 실제 시나리오에서 모든 GraphQL 스키마에는 데이터 요청을 위한 루트 쿼리 유형이 반드시 있어야 합니다. 쿼리는 GraphQL 서버의 진입점(또는 엔드포인트) 중 하나로 생각할 수 있습니다. 쿼리를 예로 추가해 보겠습니다.

Console

- 쿼리를 만들려면 다른 유형과 마찬가지로 스키마 파일에 추가하기만 하면 됩니다. 쿼리에는 다음과 같이 루트에 Query 유형 및 항목이 필요합니다.

```
schema {
  query: Name_of_Query
}

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

참고로 프로덕션 환경에서는 *Name_of_Query*를 대부분의 경우 단순히 Query라고 합니다. 이 값을 유지하는 것이 좋습니다. 쿼리 유형 내에서 필드를 추가할 수 있습니다. 각 필드는 요청에서 작업을 수행합니다. 따라서 전부는 아니더라도 대부분의 필드가 해석기에 연결됩니다. 하지만 이 섹션에서는 신경 쓰지 않습니다. 필드 작업의 형식과 관련해서는 다음과 같을 수 있습니다.

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

다음은 그 예입니다:

```
schema {
  query: Query
}

type Query {
  getObj: [Obj_Type_1]
}

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

> **Note**
>
> 이 단계에서는 Query 유형을 추가하고 schema 루트에 정의했습니다. Query 유형은 Obj_Type_1 객체 목록을 반환하는 getObj 필드를 정의했습니다. 참고로 Obj_Type_1은 이전 단계의 객체입니다. 프로덕션 코드에서는 일반적으로 Obj_Type_1과 같은 객체로 형태가 지정된 데이터를 사용하여 필드 작업을 수행하게 됩니다. 또한 getObj와 같은 필드에는 일반적으로 비즈니스 로직을 수행하는 해석기가 있습니다. 이 내용은 다른 섹션에서 다룰 예정입니다.
> 추가로 말씀드리자면, AWS AppSync는 내보내기 중에 스키마 루트를 자동으로 추가하므로 엄밀히 따지자면 스키마에 스키마 루트를 직접 추가할 필요가 없습니다. 본 서비스는 중복 스키마를 자동으로 처리합니다. 이 내용은 여기에 모범 사례로 추가하겠습니다.

CLI

> **Note**
>
> 아직 읽어보지 않은 경우 먼저 콘솔 버전을 읽어보시기 바랍니다.

1. create-type 명령을 실행하여 query 정의가 있는 schema 루트를 생성합니다.

   이 특정 명령에 대해 몇 가지 파라미터를 입력해야 합니다.

   1. API의 api-id.
   2. definition 또는 유형의 내용. 콘솔 예제에서는 다음과 같았습니다.

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

   3. 입력의 format. 이 예제에서는 SDL을 사용합니다.

   예를 들어 명령은 다음과 같을 수 있습니다.

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

CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

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

Note

`create-type` 명령에 내용을 올바르게 입력하지 않은 경우 `update-type` 명령을 실행하여 스키마 루트(또는 스키마의 모든 유형)를 업데이트할 수 있습니다. 이 예에서는 `subscription` 정의를 포함하도록 스키마 루트를 일시적으로 변경해 보겠습니다. 이 특정 명령에 대해 몇 가지 파라미터를 입력해야 합니다.

1. API의 `api-id`.
2. 유형의 `type-name`. 콘솔 예제에서는 `schema`였습니다.
3. `definition` 또는 유형의 내용. 콘솔 예제에서는 다음과 같았습니다.

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

`subscription`을 추가한 후의 스키마는 다음과 같습니다.

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

4. 입력의 `format`. 이 예제에서는 `SDL`을 사용합니다.

예를 들어 명령은 다음과 같을 수 있습니다.

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

CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

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

이 예제에서는 미리 형식이 지정된 파일을 추가해도 여전히 작동합니다.

2. `create-type` 명령을 실행하여 `Query` 유형을 생성합니다.

   이 특정 명령에 대해 몇 가지 파라미터를 입력해야 합니다.

   1. API의 `api-id`.
   2. `definition` 또는 유형의 내용. 콘솔 예제에서는 다음과 같았습니다.

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

   3. 입력의 `format`. 이 예제에서는 `SDL`을 사용합니다.

   예를 들어 명령은 다음과 같을 수 있습니다.

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

   CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

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

> **Note**
>
> 이 단계에서는 `Query` 유형을 추가하고 `schema` 루트에 정의했습니다. `Query` 유형은 `Obj_Type_1` 객체 목록을 반환한 `getObj` 필드를 정의했습니다.
> `schema` 루트 코드 `query: Query`에서 `query:` 부분은 쿼리가 스키마에 정의되었음을 나타내고 `Query` 부분은 실제 특수 객체 이름을 나타냅니다.

CDK

> **Tip**
>
> CDK를 사용하기 전에 CDK의 공식 설명서와 함께 AWS AppSync의 CDK 참조를 검토하는 것이 좋습니다.
> 아래 나열된 단계는 특정 리소스를 추가하는 데 사용되는 스니펫의 일반적인 예시만 보여줍니다. 프로덕션 코드에서는 이 예시가 올바르게 작동하는 솔루션이 아닙니다. 또한 이미 작동하는 앱을 가지고 있는 것으로 가정합니다.

쿼리와 스키마 루트를 `.graphql` 파일에 추가해야 합니다. 예제에서는 아래와 비슷했지만, 실제 스키마 코드로 바꿔야 할 수 있습니다.

```
schema {
  query: Query
}

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

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

다른 파일처럼 스키마에 유형을 직접 추가할 수 있습니다.

> **Note**
>
> 스키마 루트 업데이트는 선택 사항입니다. 이 예제에 모범 사례로 추가했습니다. GraphQL API에서 변경한 내용을 사용하려면 앱을 재배포해야 합니다.

이제 객체와 특수 객체(쿼리)를 모두 생성하는 예제를 살펴보았습니다. 또한 이러한 객체를 상호 연결하여 데이터와 작업을 설명하는 방법도 살펴보았습니다. 데이터 설명과 하나 이상의 쿼리만 포함된 스키마를 사용할 수 있습니다. 하지만 데이터 원본에 데이터를 추가하는 또 다른 작업을 추가하려고 합니다. 데이터를 수정하는 `Mutation`이라는 또 하나의 특수 객체 유형을 추가할 것입니다.

Console

- 변형은 `Mutation`이라고 부르겠습니다. `Query`와 마찬가지로, `Mutation` 내부의 필드 작업은 작업을 설명하고 해석기에 연결됩니다. 또한 특수 객체 유형이기 때문에 `schema` 루트에 정의해야 합니다. 다음은 변형의 예입니다.

  ```
  schema {
    mutation: Name_of_Mutation
  }

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

  일반적인 변형은 쿼리처럼 루트에 나열됩니다. 변형은 이름과 함께 `type` 키워드를 사용하여 정의됩니다. *`Name_of_Mutation`*은 일반적으로 `Mutation`으로 불리므로 그대로 유지하는 것이 좋습니다. 또한 각 필드는 작업을 수행합니다. 필드 작업의 형식과 관련해서는 다음과 같을 수 있습니다.

  ```
  Name_of_Mutation(params): Return_Type # version with params
  ```

```
Name_of_Mutation: Return_Type # version without params
```

다음은 그 예입니다:

```
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

이 단계에서는 addObj 필드가 있는 Mutation 유형을 추가했습니다. 이 필드의 기능을 요약해 보겠습니다.

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

addObj는 Obj_Type_1 객체를 사용하여 작업을 수행합니다. 필드로 인해 분명히 알 수도 있지만, : Obj_Type_1 반환 유형에서도 구문을 통해 이를 증명할 수 있습니다. addObj 내부에서는 Obj_Type_1 객체의 id, title, date 필드를 파라미터로 수용합니다. 보는 것과 같이 메서드 선언과 매우 흡사합니다. 하지만 메서드의 동작에 대해서는 아직 설명하지 않았습니다. 앞서 설명했듯이 스키마는 데이터와 작업의 내용을 정의하는 용도로만 사용되며 작동 방식을 정의하는 용도로는 사용할 수 없습니다. 실제 비즈니스 로직을 구현하는 방법은 나중에 첫 번째 해석기를 만들 때 다루게 될 것입니다.

스키마 작업을 완료하면 `schema.graphql` 파일로 내보낼 수 있는 옵션이 있습니다. 스키마 편집기에서 스키마 내보내기를 선택하여 지원되는 형식으로 파일을 다운로드할 수 있습니다.
추가로 말씀드리자면, AWS AppSync는 내보내기 중에 스키마 루트를 자동으로 추가하므로 엄밀히 따지자면 스키마에 스키마 루트를 직접 추가할 필요가 없습니다. 본 서비스는 중복 스키마를 자동으로 처리합니다. 이 내용은 여기에 모범 사례로 추가하겠습니다.

CLI

> **Note**
> 아직 읽어보지 않은 경우 먼저 콘솔 버전을 읽어보시기 바랍니다.

1. `update-type` 명령을 실행하여 루트 스키마를 업데이트하세요.

   이 특정 명령에 대해 몇 가지 파라미터를 입력해야 합니다.

   1. API의 `api-id`.
   2. 유형의 `type-name`. 콘솔 예제에서는 `schema`였습니다.
   3. `definition` 또는 유형의 내용. 콘솔 예제에서는 다음과 같았습니다.

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

   4. 입력의 `format`. 이 예제에서는 `SDL`을 사용합니다.

   예를 들어 명령은 다음과 같을 수 있습니다.

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

   CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

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

2. create-type 명령을 실행하여 Mutation 유형을 생성합니다.

   이 특정 명령에 대해 몇 가지 파라미터를 입력해야 합니다.

   1. API의 api-id.
   2. definition 또는 유형의 내용. 콘솔 예제에서는 다음과 같았습니다.

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

   3. 입력의 format. 이 예제에서는 SDL을 사용합니다.

   예를 들어 명령은 다음과 같을 수 있습니다.

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

   CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

   ```
   {
       "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**
>
> CDK를 사용하기 전에 CDK의 공식 설명서와 함께 AWS AppSync의 CDK 참조를 검토하는 것이 좋습니다.
> 아래 나열된 단계는 특정 리소스를 추가하는 데 사용되는 스니펫의 일반적인 예시만 보여 줍니다. 프로덕션 코드에서는 이 예시가 올바르게 작동하는 솔루션이 아닙니다. 또한 이미 작동하는 앱을 가지고 있는 것으로 가정합니다.

쿼리와 스키마 루트를 `.graphql` 파일에 추가해야 합니다. 예제에서는 아래와 비슷했지만, 실제 스키마 코드로 바꿔야 할 수 있습니다.

```
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**
>
> 스키마 루트 업데이트는 선택 사항입니다. 이 예제에 모범 사례로 추가했습니다. GraphQL API에서 변경한 내용을 사용하려면 앱을 재배포해야 합니다.

선택적 고려 사항 - 열거형을 상태로 사용

이제 기본 스키마를 만드는 방법을 알게 되었습니다. 그러나 스키마의 기능을 향상하기 위해 추가할 수 있는 많은 항목이 있습니다. 애플리케이션에서 흔히 볼 수 있는 방법 중 하나는 열거형을 상태로 사용하는 방법입니다. 열거형을 사용하여 호출 시 값 집합에서 특정 값을 선택하도록 할 수 있습니다. 이 옵션은 시간이 오래 지나도 크게 변하지 않을 것임을 알고 있는 경우에 유용합니다. 가정해 보자면, 응답에 상태 코드나 문자열을 반환하는 열거형을 추가할 수도 있습니다.

예를 들어 백엔드에 사용자의 게시물 데이터를 저장하는 소셜 미디어 앱을 만들고 있다고 가정해 보겠습니다. 스키마에는 개별 게시물의 데이터를 나타내는 Post 유형이 포함되어 있습니다.

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

Post에는 고유한 id, 게시물 title, 게시한 date 및 앱에서 처리되는 게시물 상태를 나타내는 PostStatus라는 열거형이 포함됩니다. 작업을 위해 모든 게시물 데이터를 반환하는 쿼리를 만들겠습니다.

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

또한 데이터 원본에 게시물을 추가하는 변형도 생성할 것입니다.

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

스키마를 보면 PostStatus 열거형에 여러 상태가 있는 것을 볼 수 있습니다. success(게시물 처리 완료), pending(게시물 처리 중), error(게시물을 처리할 수 없음)의 세 가지 기본 상태가 필요할 수 있습니다. 다음과 같이 열거형을 추가할 수 있습니다.

```
enum PostStatus {
  success
  pending
```

```
  error
}
```

전체 스키마는 다음과 같을 수 있습니다.

```
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
}
```

사용자가 애플리케이션에 Post를 추가하면 해당 데이터를 처리하기 위해 addPost 작업이 호출됩니다. addPost에 연결된 해석기가 데이터를 처리하는 동안 poststatus를 작업의 상태로 계속 업데이트합니다. 쿼리 시 Post에는 데이터의 최종 상태가 포함됩니다. 여기서는 스키마에서 데이터가 작동하는 방식에 대해서만 설명하고 있다는 점을 기억하세요. 요청을 이행하기 위해 데이터를 처리하는 실제 비즈니스 로직을 구현하는 해석기의 구현에 대해서는 많은 사항을 가정합니다.

선택적 고려 사항 - 구독

AWS AppSync의 구독은 변형에 대한 응답으로 간접적으로 호출됩니다. 스키마의 Subscription 지시문과 @aws_subscribe() 유형을 사용하여 이 호출을 구성하면 하나 이상의 구독을 호출하는 변형을 지정할 수 있습니다. 구독 구성에 대한 자세한 내용은 실시간 데이터를 참조하세요.

선택적 고려 사항 - 관계 및 페이지 매김

DynamoDB 테이블에 백만 개의 Posts가 저장되어 있고 그 데이터 중 일부를 반환하려고 한다고 가정해 보겠습니다. 그러나 위에 제공된 예제 쿼리는 모든 게시물만 반환합니다. 요청을 할 때마다 모든 게시물을 가져오고 싶지는 않을 수 있습니다. 대신 데이터에 페이지를 지정할 수 있습니다. 스키마를 다음과 같이 변경합니다.

- getPosts 필드에 nextToken(반복자) 및 limit(반복 제한)이라는 두 개의 입력 인수를 추가합니다.
- Posts(Post 객체 목록 검색) 및 nextToken(반복자) 필드를 포함하는 새 PostIterator 유형을 추가합니다.
- Post 객체 목록이 아닌 PostIterator를 반환하도록 getPosts를 변경합니다.

```
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
}
```

`PostIterator` 유형을 사용하면 `Post` 객체 목록의 일부와 다음 부분을 가져오기 위해 `nextToken`을 반환할 수 있습니다. `PostIterator` 내부에는 페이지 매김 토큰(`nextToken`)과 함께 반환되는 `Post` 항목(`[Post]`)의 목록이 있습니다. AWS AppSync에서는 해석기를 통해 Amazon DynamoDB에 연결되고 암호화된 토큰으로 자동 생성됩니다. 이 매핑 템플릿은 `limit` 인수의 값을 `maxResults` 파라미터로 변환하고 `nextToken` 인수의 값을 `exclusiveStartKey` 파라미터로 변환합니다. AWS AppSync 콘솔의 예제 또는 기본 제공 템플릿 샘플은 해석기 참조(JavaScript)를 참조하세요.

2단계: 데이터 원본 연결

데이터 원본은 GraphQL API가 상호작용할 수 있는 AWS 계정의 리소스입니다. AWS AppSync는 AWS Lambda, Amazon DynamoDB, 관계형 데이터베이스(Amazon Aurora Serverless), Amazon OpenSearch Service, HTTP 엔드포인트와 같은 다양한 데이터 원본을 지원합니다. AWS AppSync API는 단일 위치에서 데이터를 집계할 수 있도록 여러 데이터 원본과 상호 작용하도록 구성할 수 있습니다. AWS AppSync는 사용자 계정의 기존 AWS 리소스를 사용하거나 스키마 정의에서 사용자를 대신하여 DynamoDB 테이블을 프로비저닝할 수 있습니다.

다음 섹션에서는 GraphQL API에 데이터 원본을 연결하는 방법을 알아봅니다.

데이터 원본 유형

AWS AppSync 콘솔에서 스키마를 생성했으므로 이제 여기에 데이터 원본을 연결할 수 있습니다. API를 처음으로 만들 때 사전 정의된 스키마를 생성하는 동안 Amazon DynamoDB 테이블을 프로비저닝하는 옵션이 있습니다. 그러나 이 섹션에서는 이 옵션에 대해 다루지 않습니다. 이에 대한 예는 스키마 시작 섹션에서 확인할 수 있습니다.

대신 AWS AppSync가 지원하는 모든 데이터 원본을 살펴보겠습니다. 애플리케이션에 적합한 솔루션을 선택할 때는 여러 가지 요소를 고려하게 됩니다. 아래 섹션에서는 각 데이터 원본에 대한 몇 가지 추가 컨텍스트를 제공합니다. 데이터 원본에 대한 일반 정보는 데이터 원본을 참조하세요.

Amazon DynamoDB

Amazon DynamoDB는 확장 가능한 애플리케이션을 위한 AWS의 주요 스토리지 솔루션 중 하나입니다. DynamoDB의 핵심 구성 요소는 단순한 데이터 모음인 테이블입니다. 일반적으로 `Book` 또는 `Author` 등의 엔터티를 기반으로 테이블을 만들게 됩니다. 테이블 엔트리 정보는 각 엔트리에 고유한 필드 그룹인 항목으로 저장됩니다. 전체 항목은 데이터베이스의 행 및 레코드를 나타냅니다. 예를 들

어, Book 엔트리의 항목에는 해당 값과 함께 `title` 및 `author`이(가) 포함될 수 있습니다. `title` 및 `author` 등의 개별 필드를 특성이라고 하며, 이는 관계형 데이터베이스의 열 값과 유사합니다.

짐작할 수 있듯이 테이블은 애플리케이션의 데이터를 저장하는 데 사용됩니다. AWS AppSync를 사용하면 DynamoDB 테이블을 GraphQL API에 연결하여 데이터를 조작할 수 있습니다. 프런트엔드 웹 및 모바일 블로그에서 이 사용 사례를 확인하세요. 이 애플리케이션을 통해 사용자는 소셜 미디어 앱에 가입할 수 있습니다. 사용자는 그룹에 가입하고 그룹에 가입한 다른 사용자에게 브로드캐스트되는 게시물을 업로드할 수 있습니다. 해당 애플리케이션은 사용자, 게시물 및 사용자 그룹 정보를 DynamoDB에 저장합니다. GraphQL API(AWS AppSync에서 관리)는 DynamoDB 테이블과 상호 작용합니다. 사용자가 시스템에서 변경하는 내용이 프런트엔드에 반영되면 GraphQL API는 이러한 변경 사항을 검색하여 다른 사용자에게 실시간으로 브로드캐스트합니다.

AWS Lambda

Lambda는 이벤트에 대한 응답으로 코드를 실행하는 데 필요한 리소스를 자동으로 빌드하는 이벤트 기반 서비스입니다. Lambda는 리소스를 실행하기 위한 코드, 종속성 및 구성을 포함하는 그룹 문인 함수를 사용합니다. 함수는 함수를 간접적으로 호출하는 활동 그룹인 트리거를 감지하면 자동으로 실행됩니다. 트리거는 API 호출을 수행하는 애플리케이션, 리소스를 스핀업하는 계정 내 AWS 서비스 등이 될 수 있습니다. 함수가 트리거되면 수정할 데이터가 포함된 JSON 문서인 이벤트를 처리합니다.

Lambda는 코드를 실행하는 데 적합하며, 실행을 위해 리소스를 프로비저닝할 필요가 없습니다. 프런트엔드 웹 및 모바일 블로그에서 이 사용 사례를 확인하세요. 이 사용 사례는 DynamoDB 섹션에 소개된 사용 사례와 약간 비슷합니다. 이 애플리케이션에서 GraphQL API는 게시물 추가(변형) 및 해당 데이터 가져오기(쿼리)와 같은 작업을 정의하는 역할을 합니다. 작업의 기능(예: `getPost ( id: String ! ) : Post`, `getPostsByAuthor ( author: String ! ) : [ Post ]`)을 구현하기 위해 Lambda 함수를 사용하여 인바운드 요청을 처리합니다. 옵션 2: Lambda 해석기로 AWS AppSync에서는 AWS AppSync 서비스를 사용하여 스키마를 유지 관리하고 Lambda 데이터 원본을 작업 중 하나에 연결합니다. 작업이 호출되면 Lambda는 Amazon RDS 프록시와 상호 작용하여 데이터베이스에서 비즈니스 로직을 수행합니다.

Amazon RDS

Amazon RDS를 사용하면 관계형 데이터베이스를 빠르게 빌드하고 구성할 수 있습니다. Amazon RDS에서는 클라우드의 격리된 데이터베이스 환경으로 사용할 일반 데이터베이스 인스턴스를 만들게 됩니다. 이 인스턴스에서는 실제 RDBMS 소프트웨어(PostgreSQL, MySQL 등)인 DB 엔진을 사용합니다. 이 서비스는 AWS 인프라를 이용한 확장성, 패칭 및 암호화와 같은 보안 서비스, 낮은 배포 관리 비용을 제공함으로써 백엔드 작업을 상당히 줄여줍니다.

Lambda 섹션에서 동일한 사용 사례를 확인하세요. 옵션 3: Amazon RDS 해석기로 AWS AppSync에서는 AWS AppSync의 GraphQL API를 Amazon RDS에 직접 연결하는 옵션도 제시되어 있습니다. 이

서비스는 데이터 API를 사용하여 데이터베이스를 GraphQL API와 연결합니다. 해석기는 필드(일반적으로 쿼리, 변형 또는 구독)에 연결되며 데이터베이스에 액세스하는 데 필요한 SQL 문을 구현합니다. 클라이언트가 필드를 호출하는 요청을 보내면 해석기는 문을 실행하고 응답을 반환합니다.

Amazon EventBridge

EventBridge에서는 연결한 서비스 또는 애플리케이션(이벤트 소스)으로부터 이벤트를 수신하고 정해진 규칙에 따라 이벤트를 처리하는 파이프라인인 이벤트 버스를 만들게 됩니다. 이벤트는 실행 환경의 상태가 일부 변하는 것을 의미하고, 규칙은 이벤트에 대한 필터 집합입니다. 규칙은 이벤트 패턴 또는 이벤트 상태 변화의 메타데이터(ID, 리전, 계정 번호, ARN 등)를 따릅니다. 이벤트가 이벤트 패턴과 매칭되면 EventBridge는 파이프라인을 통해 대상 서비스(대상)로 이벤트를 보내고 규칙에 지정된 작업을 트리거합니다.

EventBridge는 상태가 변경되는 작업을 일부 다른 서비스로 라우팅하는 데 적합합니다. 프런트엔드 웹 및 모바일 블로그에서 이 사용 사례를 확인하세요. 이 예시는 여러 팀이 서로 다른 서비스를 유지 관리하는 전자 상거래 솔루션을 보여줍니다. 이러한 서비스 중 하나는 프런트엔드에서 배송의 단계(주문 생성, 진행 중, 발송됨, 배송됨 등)마다 고객에게 주문 업데이트를 제공합니다. 그러나 별도의 백엔드 팀에서 주문 시스템 데이터를 유지 관리하기 때문에 이 서비스를 관리하는 프런트엔드 팀은 주문 시스템 데이터에 직접 액세스할 수 없습니다. 백엔드 팀의 주문 시스템은 블랙박스라고도 불릴 정도로 데이터를 구조화하는 방식에 대한 정보를 얻기가 어렵습니다. 하지만 백엔드 팀은 EventBridge에서 관리하는 이벤트 버스를 통해 주문 데이터를 게시하는 시스템을 구축했습니다. 이벤트 버스에서 오는 데이터에 액세스하고 이를 프런트엔드로 라우팅하기 위해 프런트엔드 팀은 AWS AppSync에 있는 GraphQL API를 가리키는 새로운 대상을 만들었습니다. 또한 주문 업데이트와 관련된 데이터만 전송하는 규칙도 만들었습니다. 업데이트가 이루어지면 이벤트 버스의 데이터가 GraphQL API로 전송됩니다. API의 스키마는 데이터를 처리한 후 이를 프런트엔드로 전달합니다.

데이터 원본 없음

데이터 원본을 사용할 계획이 없다면 none으로 설정할 수 있습니다. 명시적으로는 여전히 데이터 원본으로 분류되지만 none 데이터 원본은 저장 매체가 아닙니다. 일반적으로 해석기는 요청을 처리하기 위해 특정 시점에 하나 이상의 데이터 원본을 간접적으로 호출합니다. 그러나 데이터 원본을 조작할 필요가 없는 경우도 있습니다. 데이터 원본을 none(으)로 설정하면 요청이 실행되고 데이터 간접 호출 단계를 건너뛴 다음 응답이 실행됩니다.

EventBridge 섹션에서 동일한 사용 사례를 확인하세요. 스키마에서 변형은 상태 업데이트를 처리한 다음 구독자에게 전송합니다. 해석기의 작동 방식을 생각해보면, 보통 한 번 이상의 데이터 원본 간접 호출이 있습니다. 그러나 이 시나리오의 데이터는 이미 이벤트 버스에 의해 자동으로 전송되었습니다. 즉, 변형이 데이터 원본 간접 호출을 수행할 필요가 없으며, 주문 상태를 로컬에서 간단히 처리할 수 있

습니다. 변형은 none(으)로 설정되며, 이 경우 데이터 원본 간접 호출 없이 패스스루 값으로 작동합니다. 그러면 데이터가 스키마에 채워지고 구독자에게 전송됩니다.

OpenSearch

Amazon OpenSearch Service는 전체 텍스트 검색, 데이터 시각화 및 로깅을 구현하는 도구 모음입니다. 이 서비스를 사용하여 업로드한 구조화된 데이터를 쿼리할 수 있습니다.

이 서비스에서는 OpenSearch의 인스턴스를 만들게 됩니다. 이를 노드라고 합니다. 노드에서는 인덱스를 하나 이상 추가하게 됩니다. 인덱스는 개념적으로 관계형 데이터베이스의 테이블과 조금 비슷합니다. (하지만 OpenSearch는 ACID를 준수하지 않으므로 그와 같은 방식으로 사용해서는 안 됩니다.) OpenSearch 서비스에 업로드하는 데이터로 인덱스를 채우게 됩니다. 데이터가 업로드되면 인덱스 내에 있는 하나 이상의 샤드에 데이터를 인덱싱합니다. 샤드는 일부 데이터를 포함하는 인덱스의 파티션과 같으며 다른 샤드와 별도로 쿼리할 수 있습니다. 업로드가 완료되면 데이터는 문서라는 JSON 파일로 구조화됩니다. 그러면 노드에 문서의 데이터를 쿼리할 수 있습니다.

HTTP 엔드포인트

HTTP 엔드포인트를 데이터 원본으로 사용할 수 있습니다. AWS AppSync는 파라미터 및 페이로드와 같은 관련 정보와 함께 엔드포인트에 요청을 보낼 수 있습니다. HTTP 응답은 해석기에 노출되며, 해석기는 작업이 완료된 후 최종 응답을 반환합니다.

데이터 원본 추가

데이터 원본을 만들었다면 AWS AppSync 서비스, 특히 API에 연결할 수 있습니다.

Console

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
   a. 대시보드에서 API를 선택합니다.
   b. 사이드바에서 데이터 원본을 선택합니다.
2. 데이터 원본 생성을 선택합니다.
   a. 데이터 원본의 이름을 지정합니다. 원하는 경우 설명을 입력할 수도 있습니다.
   b. 데이터 원본 유형을 선택합니다.
   c. DynamoDB의 경우 리전을 선택한 다음 리전의 테이블을 선택해야 합니다. 새로운 일반 테이블 역할을 만들거나 테이블의 기존 역할을 가져와서 테이블과의 상호 작용 규칙을 지정할 수 있습니다. 버전 관리를 사용하도록 설정하여 여러 클라이언트가 동시에 데이터를

업데이트하려고 할 때 각 요청에 대한 데이터 버전을 자동으로 생성할 수 있습니다. 버전 관리는 충돌 감지 및 해결 목적으로 다양한 데이터 변형을 유지 및 관리하는 데 사용됩니다. 또한 데이터 원본을 가져오고 스키마에서 여기에 액세스하는 데 필요한 CRUD, `List`, `Query` 작업의 일부를 생성하는 자동 스키마 생성을 활성화할 수도 있습니다.

OpenSearch의 경우 리전을 선택한 다음 리전의 도메인(클러스터) 을 선택해야 합니다. 새로운 일반 테이블 역할을 만들거나 테이블의 기존 역할을 가져와서 도메인과의 상호 작용 규칙을 지정할 수 있습니다.

Lambda의 경우 리전을 선택한 다음 리전에서 Lambda 함수의 ARN을 선택해야 합니다. 새로운 일반 테이블 역할을 만들거나 테이블의 기존 역할을 가져와서 Lambda 함수와의 상호 작용 규칙을 지정할 수 있습니다.

HTTP의 경우 HTTP 엔드포인트를 입력해야 합니다.

EventBridge의 경우 리전을 선택한 다음 리전에서 이벤트 버스를 선택해야 합니다. 새로운 일반 테이블 역할을 만들거나 테이블의 기존 역할을 가져와서 이벤트 버스와의 상호 작용 규칙을 지정할 수 있습니다.

RDS의 경우 리전을 선택한 다음 암호 저장소(사용자 이름 및 암호), 데이터베이스 이름, 스키마를 선택해야 합니다.

없음의 경우 실제 데이터 원본이 없는 데이터 원본을 추가하게 됩니다. 이는 실제 데이터 원본을 통하지 않고 로컬에서 해석기를 처리하기 위함입니다.

> **Note**
>
> 기존 역할을 가져오는 경우 신뢰 정책이 필요합니다. 자세한 내용은 IAM 신뢰 정책을 참조하세요.

3. 생성을 선택합니다.

> **Note**
>
> 또는 DynamoDB 데이터 원본을 생성하는 경우 콘솔의 스키마 페이지로 이동하여 페이지 상단에서 리소스 생성을 선택한 다음 사전 정의된 모델을 채워 테이블로 변환할 수 있습니다. 이 옵션에서는 기본 유형을 채우거나 가져오고, 파티션 키를 포함한 기본 테이블 데이터를 구성하고, 스키마 변경 사항을 검토합니다.

CLI

- `create-data-source` 명령을 실행하여 데이터 원본을 생성합니다.

  이 특정 명령에 대해 몇 가지 파라미터를 입력해야 합니다.

  1. API의 `api-id`.
  2. 테이블의 `name`.
  3. 데이터 원본의 `type`. 선택한 데이터 원본 유형에 따라 `service-role-arn` 및 `-config` 태그를 입력해야 할 수 있습니다.

  예를 들어 명령은 다음과 같을 수 있습니다.

```
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**
>
> CDK를 사용하기 전에 CDK의 공식 설명서와 함께 AWS AppSync의 CDK 참조를 검토하는 것이 좋습니다.
> 아래 나열된 단계는 특정 리소스를 추가하는 데 사용되는 스니펫의 일반적인 예시만 보여줍니다. 프로덕션 코드에서는 이 예시가 올바르게 작동하는 솔루션이 아닙니다. 또한 이미 작동하는 앱을 가지고 있는 것으로 가정합니다.

특정 데이터 원본을 추가하려면 스택 파일에 구성을 추가해야 합니다. 데이터 원본 유형 목록은 다음에서 찾을 수 있습니다.

- DynamoDbDataSource
- EventBridgeDataSource
- HttpDataSource
- LambdaDataSource
- NoneDataSource

- OpenSearchDataSource
- RdsDataSource

1. 대개는 사용 중인 서비스에 가져오기 지침을 추가해야 할 수 있습니다. 예를 들면 다음 형식을 따를 수 있습니다.

```
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'
```

예를 들어 다음과 같은 방법으로 AWS AppSync 및 DynamoDB 서비스를 가져올 수 있습니다.

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

2. RDS와 같은 일부 서비스는 데이터 원본을 생성하기 전에 스택 파일에 몇 가지 추가 설정이 필요합니다(예: VPC 생성, 역할, 액세스 보안 인증 정보). 자세한 내용은 관련 CDK 페이지의 예시를 참조하세요.
3. 대부분의 데이터 원본(특히 AWS 서비스)의 경우 스택 파일에 데이터 원본의 새 인스턴스를 생성하게 됩니다. 일반적으로 다음과 같습니다.

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

예를 들어 Amazon DynamoDB 테이블의 예는 다음과 같습니다.

```
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**
>
> 대부분의 데이터 원본은 필수 속성을 하나 이상 갖습니다(? 기호 없이 표시됨). 어떤 속성이 필요한지 알아보려면 CDK 설명서를 참조하세요.

4. 다음으로 데이터 원본을 GraphQL API에 연결해야 합니다. 권장되는 방법은 파이프라인 해석기용 함수를 만들 때 추가하는 것입니다. 예를 들어 아래 스니펫은 DynamoDB 테이블의 모든 요소를 스캔하는 함수입니다.

```
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,
});
```

dataSource 속성에서 GraphQL API(add_api)를 호출하고 기본 제공되는 메서드(addDynamoDbDataSource) 중 하나를 사용하여 테이블과 GraphQL API 사이의 연결을 설정할 수 있습니다. 인수는 AWS AppSync 콘솔(이 예에서는 data_source_name_in_console)에 존재할 이 링크의 이름과 테이블 메서드(add_ddb_table)입니다. 이 주제에 대한 자세한 내용은 다음 섹션에서 해석기 제작을 시작하면서 설명하겠습니다.

다른 방법으로도 데이터 원본을 연결할 수 있습니다. 기술적으로는 테이블 함수의 속성 목록에 api를 추가할 수 있습니다. 예를 들어 다음은 3단계의 스니펫이지만, 여기에는 GraphQL API가 포함된 api 속성이 있습니다.

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

```
});

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

 ...

  api: add_api
});
```

또는 GraphqlApi 구성을 별도로 호출할 수도 있습니다.

```
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);
```

함수의 속성에만 연결을 만드는 것이 좋습니다. 그렇지 않으면 AWS AppSync 콘솔에서 수동으로 해석기 함수를 데이터 원본에 연결하거나(콘솔 값 data_source_name_in_console을(를) 계속 사용하려는 경우) data_source_name_in_console_2 등의 다른 이름으로 함수에 별도의 연결을 만들어야 합니다. 이는 속성이 정보를 처리하는 방식에 제한이 있기 때문입니다.

> **Note**
>
> 변경 사항을 확인하려면 앱을 다시 배포해야 합니다.

IAM 신뢰 정책

데이터 원본에 대해 기존 IAM 역할을 사용하려는 경우 Amazon DynamoDB 테이블에서 PutItem 같은 AWS 리소스에 대한 작업을 수행할 수 있는 적정 권한을 해당 역할에 부여해야 합니다. 또한 다음 예제 정책에서처럼 AWS AppSync에서 리소스 액세스용으로 활용할 수 있도록 해당 역할에 대해 신뢰 정책을 수정해야 합니다.

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

신뢰 정책에 조건을 추가하여 데이터 원본에 대한 액세스를 원하는 대로 제한할 수도 있습니다. 현재 SourceArn 및 SourceAccount 키를 이러한 조건에 사용할 수 있습니다. 예를 들어 다음 정책에서는 데이터 원본에 대한 액세스를 123456789012 계정으로 제한합니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "123456789012"
        }
      }
    }
  ]
}
```

또는 다음 정책을 사용하여 데이터 원본에 대한 액세스를 특정 API(예: abcdefghijklmnopq)로 제한할 수 있습니다.

```
{
  "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"
        }
      }
    }
  ]
}
```

다음 정책을 사용하여 특정 지역(예: `us-east-1`)의 모든 AWS AppSync API에 대한 액세스를 제한할 수 있습니다.

```
{
  "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/*"
        }
      }
    }
  ]
}
```

다음 섹션(해석기 구성)에서는 해석기 비즈니스 로직을 추가하고 이를 스키마의 필드에 연결하여 데이터 원본의 데이터를 처리합니다.

역할 정책 구성에 대한 자세한 내용은 IAM 사용 설명서의 역할 변경을 참조하세요.

AWS AppSync용 AWS Lambda 해석기의 크로스 계정 액세스에 관한 자세한 내용은 Building cross-account AWS Lambda resolvers for AWS AppSync를 참조하세요.

3단계: 해석기 구성

이전 섹션에서는 GraphQL 스키마와 데이터 원본을 만든 다음 서비스에서 함께 연결하는 방법을 배웠습니다. AWS AppSync 스키마에서 쿼리 및 뮤테이션에 하나 이상의 필드(작업)를 설정했을 수 있습니다. 스키마는 작업이 데이터 소스에서 요청하는 데이터 종류를 설명했지만, 해당 작업이 데이터에 대해 어떻게 동작하는지는 구현하지 못했습니다.

작업의 동작은 항상 해석기에서 구현되며, 해석기는 작업을 수행하는 필드에 연결됩니다. 해석기의 일반적인 작동 방식에 대한 자세한 내용은 해석기 페이지를 참조하세요.

에서 AWS AppSync리졸버는 리졸버가 실행되는 환경인 런타임에 연결되어 있습니다. 런타임에 따라 해석기를 작성할 언어가 결정됩니다. 현재 지원되는 런타임은 APPSYNC_JS () 와 아파치 벨로시티 템플릿 언어 (JavaScriptVTL) 입니다.

해석기를 구현할 때는 다음과 같은 일반적인 구조를 따릅니다.

- 단계 이전: 클라이언트가 요청을 보내면 사용 중인 스키마 필드(일반적으로 쿼리, 뮤테이션, 구독)의 해석기에 요청 데이터가 전달됩니다. 해석기는 데이터가 해석기를 통해 이동하기 전에 일부 사전 처리 작업을 수행할 수 있는 사전 단계 핸들러를 사용하여 요청 데이터 처리를 시작합니다.
- 함수: 사전 단계가 실행되면 요청이 함수 목록으로 전달됩니다. 목록의 첫 번째 함수는 데이터 원본에 대해 실행됩니다. 함수는 자체 요청 및 응답 핸들러를 포함하는 해석기 코드의 하위 집합입니다. 요청 핸들러는 요청 데이터를 가져와 데이터 소스에 대해 작업을 수행합니다. 응답 핸들러는 데이터 소스의 응답을 다시 목록으로 전달하기 전에 해당 데이터 소스의 응답을 처리합니다. 함수가 두 개 이상인 경우 요청 데이터는 목록의 다음 함수로 전송되어 실행됩니다. 목록에 있는 함수는 개발자가 정의한 순서대로 순차적으로 실행됩니다. 모든 함수가 실행되면 최종 결과가 사후 단계로 전달됩니다.
- 단계 이후: 이후 단계는 GraphQL 응답으로 전달하기 전에 최종 함수의 응답에 대한 몇 가지 최종 작업을 수행할 수 있는 핸들러 함수입니다.

이 흐름은 파이프라인 해석기의 예입니다. 파이프라인 해석기는 두 런타임 모두에서 지원됩니다. 하지만 여기서는 파이프라인 해석기가 수행할 수 있는 작업에 대한 간단한 설명입니다. 또한 가능한 해석기 구성을 하나만 설명합니다. 지원되는 리졸버 구성에 대한 자세한 내용은 APPSYNC_JS의 리졸버 개요 또는 VTL의 JavaScript 리졸버 매핑 템플릿 개요를 참조하십시오.

보시다시피 해석기는 모듈식입니다. 해석기의 구성 요소가 제대로 작동하려면 다른 구성 요소에서 실행 상태를 들여다볼 수 있어야 합니다. 해석기 섹션에서 해석기의 각 구성 요소는 인수 집합(`args`, `context` 등)으로 실행 상태에 대한 중요한 정보를 전달할 수 있다는 것을 확인했습니다. 에서는 `context` 이 문제를 에서 엄격하게 처리합니다. AWS AppSync확인 중인 필드에 대한 정보를 담는 컨

테이너입니다. 여기에는 전달되는 인수, 결과, 권한 부여 데이터, 헤더 데이터 등 모든 것이 포함될 수 있습니다. 컨텍스트에 대한 자세한 내용은 APPSYNC_JS의 경우 해석기 컨텍스트 객체 참조를, 또는 VTL의 경우 해석기 매핑 템플릿 컨텍스트 참조를 참조하세요.

리졸버를 구현하는 데 사용할 수 있는 도구는 컨텍스트뿐만이 아닙니다. AWS AppSync 값 생성, 오류 처리, 파싱, 변환 등을 위한 다양한 유틸리티를 지원합니다. APPSYNC_JS의 경우 여기에서, 또는 VTL의 경우 여기에서 유틸리티 목록을 볼 수 있습니다.

다음 섹션에서는 GraphQL API에서 해석기를 구성하는 방법을 알아봅니다.

주제

- 해석기 구성(JavaScript)
- 해석기 구성(VTL)

해석기 구성(JavaScript)

GraphQL 해석기는 형식 스키마의 필드를 데이터 원본에 연결합니다. 해석기는 요청을 이행하는 메커니즘입니다.

AWS AppSync의 해석기는 JavaScript를 사용하여 GraphQL 식을 데이터 원본에서 사용할 수 있는 형식으로 변환합니다. 또는 Apache Velocity Template Language(VTL)로 작성된 매핑 템플릿을 사용하여 GraphQL 식을 데이터 원본이 사용할 수 있는 형식으로 변환할 수 있습니다.

이 섹션에서는 JavaScript를 사용하여 해석기를 구성하는 방법을 설명합니다. 해석기 자습서(JavaScript) 섹션에서는 JavaScript를 사용하여 해석기를 구현하는 방법에 대한 심화된 자습서를 제공합니다. 해석기 참조(JavaScript) 섹션에서는 JavaScript 해석기와 함께 사용할 수 있는 유틸리티 작업에 대한 설명을 제공합니다.

앞서 언급한 자습서를 사용하기 전에 이 안내서를 확인하는 것이 좋습니다.

이 섹션에서는 쿼리와 변형을 위한 해석기를 만들고 구성하는 방법을 살펴보겠습니다.

> **Note**
>
> 이 안내서에서는 스키마를 만들었으며 쿼리 또는 변형이 하나 이상 있다고 가정합니다. 구독(실시간 데이터)을 찾고 있다면 이 안내서를 참조하세요.

이 섹션에서는 아래 스키마를 사용하는 예제와 함께 해석기 구성을 위한 몇 가지 일반적인 단계를 제공합니다.

```
// 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]
}
```

기본 쿼리 해석기 생성

이 섹션에서는 기본 쿼리 해석기를 생성하는 방법을 보여 줍니다.

Console

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
   a. API 대시보드에서 GraphQL API를 선택합니다.
   b. 사이드바에서 스키마를 선택합니다.
2. 스키마와 데이터 원본의 세부 정보를 입력합니다. 자세한 내용은 스키마 설계 및 데이터 원본 연결 섹션을 참조하세요.
3. 스키마 편집기 옆에는 해석기라는 창이 있습니다. 이 상자에는 스키마 창에 정의된 형식 및 필드 목록이 있습니다. 필드에 해석기를 연결할 수 있습니다. 주로 필드 작업에 해석기를 연결하게 될 것입니다. 이 섹션에서는 간단한 쿼리 구성을 살펴보겠습니다. 쿼리 유형에서 쿼리 필드 옆에 있는 연결을 선택합니다.
4. 해석기 연결 페이지의 해석기 유형에서 파이프라인 또는 단위 해석기 중에서 선택할 수 있습니다. 이러한 유형에 대한 자세한 내용은 해석기를 참조하세요. 이 안내서에서는 `pipeline resolvers`를 활용합니다.

> **Tip**
>
> 파이프라인 해석기를 생성하면 데이터 원본이 파이프라인 함수에 연결됩니다. 함수는 파이프라인 해석기를 생성한 후에 생성되므로 이 페이지에는 설정을 위한 옵션이 없습니다. 단위 해석기를 사용하는 경우 데이터 원본이 해석기에 직접 연결되므로 이 페이지에서 설정하게 됩니다.

해석기 런타임의 경우 APPSYNC_JS를 선택하여 JavaScript 런타임을 활성화하세요.

5. 이 API에 캐싱을 활성화할 수 있습니다. 그러나 지금은 이 기능을 끄는 것이 좋습니다. 생성을 선택합니다.
6. 해석기 편집 페이지에는 해석기 핸들러 및 응답(사전 및 사후 단계)에 대한 로직을 구현할 수 있게 해주는 해석기 코드라는 코드 편집기가 있습니다. 자세한 내용은 JavaScript 해석기 개요를 참조하세요.

> **Note**
>
> 이 예에서는 요청을 비워 두고 컨텍스트의 마지막 데이터 원본 결과를 반환하도록 응답을 설정해 보겠습니다.

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

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

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

이 섹션 아래에는 함수라는 테이블이 있습니다. 함수를 사용하면 여러 해석기에서 재사용할 수 있는 코드를 구현할 수 있습니다. 코드를 계속 다시 작성하거나 복사하는 대신, 소스 코드를 함수로 저장하여 필요할 때마다 해석기에 추가되도록 할 수 있습니다.

함수는 파이프라인 작업 목록의 대부분을 구성합니다. 해석기에서 여러 함수를 사용하는 경우 함수 순서를 설정하면 해당 순서대로 순차적으로 실행됩니다. 이러한 함수는 요청 함수가 실행된 후와 응답 함수가 시작되기 전에 실행됩니다.

새 함수를 추가하려면 함수에서 함수 추가를 선택한 다음 새 함수 생성을 선택합니다. 또는 선택할 수 있는 함수 생성 버튼이 표시될 수도 있습니다.

a. 데이터 원본을 선택합니다. 이 항목은 해석기가 작동하는 데이터 원본이 됩니다.

> **Note**
>
> 이 예제에서는 `id`로 `Post` 객체를 검색하는 `getPost`에 대한 해석기를 연결합니다. 이 스키마에 대한 DynamoDB 테이블을 이미 설정했다고 가정해 보겠습니다. 파티션 키는 `id`로 설정되어 있고 비어 있습니다.

b. `Function name`을 입력합니다.

c. 함수 코드에서 함수의 동작을 구현해야 합니다. 헷갈릴 수도 있지만 각 함수에는 고유한 로컬 요청 및 응답 핸들러가 있습니다. 요청이 실행되면 요청을 처리하기 위한 데이터 원본 간접 호출이 수행되고, 응답 핸들러가 데이터 원본 응답을 처리합니다. 결과는 컨텍스트 객체에 저장됩니다. 이후에는 목록의 다음 함수가 실행되거나 마지막 함수인 경우 사후 단계 응답 핸들러로 전달됩니다.

> **Note**
>
> 이 예제에서는 데이터 원본에서 `Post` 객체 목록을 가져오는 `getPost`에 해석기를 연결합니다. 요청 함수가 테이블에서 데이터를 요청하고, 테이블이 응답을 컨텍스트(ctx)로 전달하면, 응답이 컨텍스트에 결과를 반환합니다. AWS AppSync의 강점은 다른 AWS 서비스와의 상호 연결성에 있습니다. DynamoDB를 사용하고 있기 때문에 이와 같은 작업을 단순화할 수 있는 일련의 작업이 있습니다. 다른 데이터 원본 유형에 대한 몇 가지 보일러플레이트 예제도 있습니다.
> 코드는 다음과 같습니다.

```
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;
}
```

이 단계에서는 두 가지 함수를 추가했습니다.

- `request`: 요청 핸들러는 데이터 원본에 대해 검색 작업을 수행합니다. 인수에는 컨텍스트 객체(`ctx`) 또는 특정 작업을 수행하는 모든 해석기에 사용할 수 있는 일부 데이터가 포함됩니다. 예를 들어 권한 부여 데이터, 해석 중인 필드 이름 등이 포함될 수 있습니다. 반환 문은 `Scan` 작업을 수행합니다(예시는 여기를 참조). DynamoDB로 작업 중이므로 해당 서비스의 일부 작업을 사용할 수 있습니다. 스캔은 테이블 내 모든 항목에 대한 기본 가져오기를 수행합니다. 이 작업의 결과는 응답 핸들러로 전달되기 전에 컨텍스트 객체에 `result` 컨테이너로 저장됩니다. `request`는 파이프라인에서 응답보다 먼저 실행됩니다.
- `response`: `request`의 출력을 반환하는 응답 핸들러입니다. 인수는 업데이트된 컨텍스트 객체이고, 반환 문은 `ctx.prev.result`입니다. 이 시점에서는 이 값에 익숙하지 않을 수 있습니다. `ctx`는 컨텍스트 객체를 나타냅니다. `prev`는 파이프라인에서의 이전 작업을 나타내며, 여기에서는 `request`였습니다. `result`에는 파이프라인을 통과하는 해석기의 결과가 포함됩니다. 종합해 보면 `ctx.prev.result`는 마지막으로 수행된 작업, 즉 요청 핸들러의 결과를 반환합니다.

d. 완료했으면 생성을 선택합니다.

7. 해석기 화면으로 돌아가서 함수 아래에서 함수 추가 드롭다운을 선택하고 함수를 함수 목록에 추가합니다.
8. 저장을 선택하여 해석기를 업데이트합니다.

CLI

함수를 추가하는 방법

- `create-function` 명령을 사용하여 파이프라인 해석기용 함수를 생성합니다.

이 특정 명령에 대해 몇 가지 파라미터를 입력해야 합니다.

1. API의 `api-id`.
2. AWS AppSync 콘솔에 있는 함수의 `name`.
3. `data-source-name` 또는 함수가 사용할 데이터 원본의 이름. 이미 생성되어 AWS AppSync 서비스에 있는 GraphQL API에 연결되어 있어야 합니다.
4. `runtime` 또는 함수의 환경 및 언어. JavaScript의 경우 이름은 `APPSYNC_JS`, 런타임은 `1.0.0`이어야 합니다.
5. `code` 또는 함수의 요청 및 응답 핸들러. 수동으로 입력할 수도 있지만 .txt 파일(또는 유사한 형식)에 추가한 다음 인수로 전달하는 것이 훨씬 쉽습니다.

> **Note**
>
> 쿼리 코드는 인수로 전달되는 파일에 있습니다.
>
> ```
> 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;
> }
> ```

예를 들어 명령은 다음과 같을 수 있습니다.

```
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}
```

CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

```
{
    "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**
>
> 함수를 해석기에 연결하는 데 사용되므로 `functionId`를 어딘가에 기록해 두어야 합니다.

해석기 생성

- `create-resolver` 명령을 실행하여 `Query`에 대한 파이프라인 함수를 생성합니다.

  이 특정 명령에 대해 몇 가지 파라미터를 입력해야 합니다.

  1. API의 `api-id`.
  2. `type-name` 또는 스키마의 특수 객체 유형(쿼리, 변형, 구독).
  3. `field-name` 또는 해석기를 연결하려는 특수 객체 유형 내의 필드 작업.
  4. 단위 또는 파이프라인 해석기를 지정하는 `kind`. 파이프라인 함수를 활성화하려면 이 값을 `PIPELINE`으로 설정합니다.

5. `pipeline-config` 또는 해석기에 연결할 함수. 함수의 `functionId` 값을 알고 있어야 합니다. 목록 순서가 중요합니다.
6. `runtime`으로, 여기에서는 `APPSYNC_JS`(자바스크립트)였습니다. `runtimeVersion`은 현재 `1.0.0`입니다.
7. `code`로, 사전 및 사후 단계 핸들러를 포함합니다.

> **Note**
>
> 쿼리 코드는 인수로 전달되는 파일에 있습니다.

```
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;
}
```

예를 들어 명령은 다음과 같을 수 있습니다.

```
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}
```

CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

```
{
    "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**
>
> CDK를 사용하기 전에 CDK의 공식 설명서와 함께 AWS AppSync의 CDK 참조를 검토하는 것이 좋습니다.
> 아래 나열된 단계는 특정 리소스를 추가하는 데 사용되는 스니펫의 일반적인 예시만 보여줍니다. 프로덕션 코드에서는 이 예시가 올바르게 작동하는 솔루션이 아닙니다. 또한 이미 작동하는 앱을 가지고 있는 것으로 가정합니다.

기본 앱에는 다음 항목이 필요합니다.

1. 서비스 가져오기 지시문

2. 스키마 코드
3. 데이터 원본 생성기
4. 함수 코드
5. 해석기 코드

스키마 설계 및 데이터 원본 연결 섹션을 통해 스택 파일에 다음과 같은 형식의 가져오기 지시문이 포함된다는 것을 알 수 있습니다.

```
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**
>
> 이전 섹션에서는 AWS AppSync 구성을 가져오는 방법만 설명했습니다. 실제 코드에서는 앱을 실행하는 것만으로도 더 많은 서비스를 가져와야 합니다. 이 예제에서는 아주 간단한 CDK 앱을 만들기 위해 DynamoDB 테이블인 데이터 원본과 함께 최소한 AWS AppSync 서비스를 가져와야 했습니다. 또한 앱을 배포하기 위해 몇 가지 추가 구성을 가져와야 했습니다.
>
> ```
> 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';
> ```
>
> 각 내용을 요약하면 다음과 같습니다.
>
> - `import * as cdk from 'aws-cdk-lib';`: 이렇게 하면 CDK 앱 및 스택과 같은 구성을 정의할 수 있습니다. 여기에는 메타데이터 조작과 같은 애플리케이션에 유용한 유틸리티 함수도 포함되어 있습니다. 이 가져오기 지시문에 익숙하지만 여기에서 cdk 코어 라이브러리가 사용되지 않는 이유가 궁금하다면 마이그레이션 페이지를 참조하세요.
> - `import * as appsync from 'aws-cdk-lib/aws-appsync';`: AWS AppSync 서비스를 가져옵니다.
> - `import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';`: DynamoDB 서비스를 가져옵니다.
> - `import { Construct } from 'constructs';`: 루트 구성을 정의하는 데 필요합니다.

가져오기 유형은 호출하는 서비스에 따라 달라집니다. 예를 보려면 CDK 설명서를 살펴보세요. 페이지 상단의 스키마는 CDK 앱의 파일로, `.graphql` 파일과는 별개입니다. 스택 파일에서 다음 형식을 사용하여 새 GraphQL과 연결할 수 있습니다.

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

> **Note**
>
> 범위 `add_api`에서는 `appsync.GraphqlApi(scope: Construct, id: string , props: GraphqlApiProps)` 뒤에 오는 new 키워드를 사용하여 새로운 GraphQL API를 추가합니다. 범위는 `this`, CFN id는 `graphQL-example`이며, 속성은 `my-first-api`(콘솔의 API 이름) 및 `schema.graphql`(스키마 파일의 절대 경로) 입니다.

데이터 원본을 추가하려면 먼저 데이터 원본을 스택에 추가해야 합니다. 그런 다음 소스별 메서드를 사용하여 GraphQL API와 연결해야 합니다. 해석기 함수를 만들면 연결이 이루어집니다. 그 동안 `dynamodb.Table`을 사용하여 DynamoDB 테이블을 생성하는 예제를 살펴보겠습니다.

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

> **Note**
>
> 예제에서 이 방법을 사용한다면 CFN ID가 `posts-table`이고 파티션 키가 `id (S)`인 새로운 DynamoDB 테이블을 추가하게 됩니다.

다음으로 스택 파일에 해석기를 구현해야 합니다. 다음은 DynamoDB 테이블의 모든 항목을 스캔하는 간단한 쿼리의 예입니다.

```
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

먼저 add_func라는 함수를 만들었습니다. 이 생성 순서는 다소 직관적이지 않은 것처럼 보일 수 있지만, 해석기 자체를 만들기 전에 파이프라인 해석기에 함수를 만들어야 합니다. 함수는 다음과 같은 형식을 따릅니다.

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

범위는 this, CFN id는 func-get-posts였으며, 속성에는 실제 함수 세부 정보가 포함되어 있습니다. 속성 안에는 다음이 포함되었습니다.

- AWS AppSync 콘솔(`get_posts_func_1`)에 표시될 함수의 `name`.
- 앞서 만든 GraphQL API(`add_api`).
- 데이터 원본. 데이터 원본을 GraphQL API 값에 연결한 다음 함수에 연결하는 지점입니다. 생성한 테이블(`add_ddb_table`)을 가져와서 `GraphqlApi` 메서드(`addDynamoDbDataSource`) 중 하나를 사용하여 GraphQL API(`add_api`)에 연결합니다. ID 값(`table-for-posts`)은 AWS AppSync 콘솔에 있는 데이터 원본의 이름입니다. 소스별 메서드 목록은 다음 페이지를 참조하세요.
  - DynamoDbDataSource
  - EventBridgeDataSource
  - HttpDataSource
  - LambdaDataSource
  - NoneDataSource
  - OpenSearchDataSource
  - RdsDataSource
- 코드에는 간단한 스캔 및 반환인 함수의 요청 및 응답 핸들러가 포함되어 있습니다.
- 런타임은 APPSYNC_JS 런타임 버전 1.0.0을 사용하도록 지정합니다. 참고로 이 버전은 현재 APPSYNC_JS에서 사용할 수 있는 유일한 버전입니다.

다음으로 함수를 파이프라인 해석기에 연결해야 합니다. 다음과 같은 형식을 사용하여 해석기를 생성했습니다.

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

범위는 `this`, CFN id는 `pipeline-resolver-get-posts`였으며, 속성에는 실제 함수 세부 정보가 포함되어 있습니다. 속성 안에는 다음이 포함되었습니다.

- 앞서 만든 GraphQL API(`add_api`).
- 특수 객체 유형 이름. 쿼리 작업이므로 `Query` 값을 추가하기만 하면 됩니다.
- 필드 이름(`getPost`)은 `Query` 유형 아래의 스키마에 있는 필드 이름입니다.
- 코드에는 사전 및 사후 핸들러가 포함되어 있습니다. 이 예제는 함수가 작업을 수행한 후 컨텍스트에 있었던 모든 결과를 반환합니다.
- 런타임은 APPSYNC_JS 런타임 버전 1.0.0을 사용하도록 지정합니다. 참고로 이 버전은 현재 APPSYNC_JS에서 사용할 수 있는 유일한 버전입니다.

- 파이프라인 구성에는 생성한 함수(add_func)에 대한 참조가 포함되어 있습니다.

이 예제에서 수행한 단계를 요약하자면 요청 및 응답 핸들러를 구현한 AWS AppSync 함수를 살펴보았습니다. 함수는 데이터 원본과의 상호 작용을 담당했습니다. 요청 핸들러는 DynamoDB 데이터 원본에 대해 수행할 작업을 지시하도록 AWS AppSync에 Scan 작업을 전송했습니다. 응답 핸들러는 항목의 목록(ctx.result.items)을 반환했습니다. 그런 다음 항목 목록이 Post GraphQL 유형에 자동으로 매핑되었습니다.

기본 변형 해석기 생성

이 섹션에서는 기본 변형 해석기를 생성하는 방법을 보여 줍니다.

Console

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
   a. API 대시보드에서 GraphQL API를 선택합니다.
   b. 사이드바에서 스키마를 선택합니다.
2. 해석기 섹션 및 변형 유형에서 필드 옆에 있는 연결을 선택합니다.

   > **Note**
   > 이 예제에서는 테이블에 Post 객체를 추가하는 createPost에 대한 해석기를 연결합니다. 마지막 섹션과 동일한 DynamoDB 테이블을 사용한다고 가정해 보겠습니다. 파티션 키는 id로 설정되어 있고 비어 있습니다.

3. 해석기 연결 페이지의 해석기 유형에서 pipeline resolvers를 선택합니다. 참고로 해석기에 대한 추가 정보는 여기에서 확인할 수 있습니다. 해석기 런타임의 경우 APPSYNC_JS를 선택하여 JavaScript 런타임을 활성화하세요.
4. 이 API에 캐싱을 활성화할 수 있습니다. 그러나 지금은 이 기능을 끄는 것이 좋습니다. 생성을 선택합니다.
5. 함수 추가를 선택한 다음 새 함수 생성을 선택합니다. 또는 선택할 수 있는 함수 생성 버튼이 표시될 수도 있습니다.
   a. 데이터 원본을 선택합니다. 변형을 사용하여 조작할 데이터가 들어 있는 원본이어야 합니다.
   b. Function name을 입력합니다.

c. 함수 코드에서 함수의 동작을 구현해야 합니다. 이는 변형이므로 요청은 간접적으로 호출된 데이터 원본에서 일부 상태 변경 작업을 수행하는 것이 이상적입니다. 결과는 응답 함수에 의해 처리됩니다.

> **Note**
>
> `createPost`는 파라미터를 데이터로 사용하여 테이블에 새 `Post`를 추가하거나 '퍼팅'합니다. 다음과 같이 추가할 수 있습니다.
>
> ```
> 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;
> }
> ```
>
> 이 단계에서 `request` 및 `response` 함수도 추가했습니다.
>
> - `request`: 요청 핸들러는 컨텍스트를 인수로 받아들입니다. 요청 핸들러 반환 문은 내장된 DynamoDB 작업인 `PutItem` 명령을 수행합니다(예시는 여기 또는 여기 참조). `PutItem` 명령은 파티션 `key` 값(`util.autoid()`에서 자동으로 생성됨)과 컨텍스트 인수 입력의 `attributes`(요청 시 전달할 값)를 가져와 DynamoDB 테이블에 `Post` 객체를 추가합니다. `key`는 `id`이고 `attributes`는 `date` 및 `title` 필드 인수입니다. 둘 다 DynamoDB 테이블에서 작동하도록 `util.dynamodb.toMapValues` 도우미를 통해 미리 형식이 지정되어 있습니다.

- `response`: 응답은 업데이트된 컨텍스트를 수락하고 요청 핸들러의 결과를 반환합니다.

d. 완료했으면 생성을 선택합니다.

6. 해석기 화면으로 돌아가서 함수 아래에서 함수 추가 드롭다운을 선택하고 함수를 함수 목록에 추가합니다.
7. 저장을 선택하여 해석기를 업데이트합니다.

CLI

함수를 추가하는 방법

- `create-function` 명령을 사용하여 파이프라인 해석기용 함수를 생성합니다.

  이 특정 명령에 대해 몇 가지 파라미터를 입력해야 합니다.

  1. API의 `api-id`.
  2. AWS AppSync 콘솔에 있는 함수의 `name`.
  3. `data-source-name` 또는 함수가 사용할 데이터 원본의 이름. 이미 생성되어 AWS AppSync 서비스에 있는 GraphQL API에 연결되어 있어야 합니다.
  4. `runtime` 또는 함수의 환경 및 언어. JavaScript의 경우 이름은 `APPSYNC_JS`, 런타임은 `1.0.0`이어야 합니다.
  5. `code` 또는 함수의 요청 및 응답 핸들러. 수동으로 입력할 수도 있지만 .txt 파일(또는 유사한 형식)에 추가한 다음 인수로 전달하는 것이 훨씬 쉽습니다.

  > **Note**
  >
  > 쿼리 코드는 인수로 전달되는 파일에 있습니다.

```
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;
}
```

예를 들어 명령은 다음과 같을 수 있습니다.

```
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}
```

CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

```
{
    "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**
>
> 함수를 해석기에 연결하는 데 사용되므로 `functionId`를 어딘가에 기록해 두어야 합니다.

해석기 생성

- `create-resolver` 명령을 실행하여 `Mutation`에 대한 파이프라인 함수를 생성합니다.

  이 특정 명령에 대해 몇 가지 파라미터를 입력해야 합니다.

  1. API의 `api-id`.
  2. `type-name` 또는 스키마의 특수 객체 유형(쿼리, 변형, 구독).
  3. `field-name` 또는 해석기를 연결하려는 특수 객체 유형 내의 필드 작업.
  4. 단위 또는 파이프라인 해석기를 지정하는 `kind`. 파이프라인 함수를 활성화하려면 이 값을 `PIPELINE`으로 설정합니다.
  5. `pipeline-config` 또는 해석기에 연결할 함수. 함수의 `functionId` 값을 알고 있어야 합니다. 목록 순서가 중요합니다.
  6. `runtime`으로, 여기에서는 `APPSYNC_JS`(자바스크립트)였습니다. `runtimeVersion`은 현재 `1.0.0`입니다.
  7. `code`로, 사전 및 사후 단계를 포함합니다.

  > **Note**
  >
  > 쿼리 코드는 인수로 전달되는 파일에 있습니다.

  ```
  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;
}
```

예를 들어 명령은 다음과 같을 수 있습니다.

```
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}
```

CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

```
{
    "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**
>
> CDK를 사용하기 전에 CDK의 공식 설명서와 함께 AWS AppSync의 CDK 참조를 검토하는 것이 좋습니다.
> 아래 나열된 단계는 특정 리소스를 추가하는 데 사용되는 스니펫의 일반적인 예시만 보여줍니다. 프로덕션 코드에서는 이 예시가 올바르게 작동하는 솔루션이 아닙니다. 또한 이미 작동하는 앱을 가지고 있는 것으로 가정합니다.

- 변형을 만들려면 같은 프로젝트에 참여하고 있다고 가정하고 쿼리처럼 스택 파일에 추가하면 됩니다. 테이블에 새 `Post`를 추가하는 변형에 대한 수정된 함수와 해석기는 다음과 같습니다.

```
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**
>
> 이 변형과 쿼리는 구조가 비슷하므로 변형을 만들기 위해 변경한 내용만 설명하겠습니다.
> 테이블에 Posts를 추가한다는 사실을 반영하기 위해 함수에서 CFN id를 func-add-post로 변경하고 이름을 add_posts_func_1로 변경했습니다.
> addDynamoDbDataSource 메서드에 필요하므로 데이터 원본에서 table-for-posts-2로 AWS AppSync 콘솔의 테이블(add_ddb_table)에 대한 새로운 연결을 만들었습니다. 이 새 연결은 이전에 만든 것과 동일한 테이블을 계속 사용하지만, 이제 AWS AppSync 콘솔에 두 개의 연결이 있습니다. 하나는 쿼리에 대한 table-for-posts, 다른 하나는 변형에 대한 table-for-posts-2로 생성된 연결입니다. id 값을 자동으로 생성하고 나머지 필드에 대해 클라이언트의 입력을 수락하여 Post를 추가하도록 코드가 변경되었습니다.
> 테이블에 Posts를 추가한다는 사실을 반영하기 위해 해석기에서 id 값을 pipeline-resolver-create-posts로 변경했습니다. 스키마에 변형를 반영하기 위해 유형 이름은 Mutation으로, 이름은 createPost로 변경되었습니다. 파이프라인 구성이 새 변형 함수 add_func_2로 설정되었습니다.

이 예제의 내용을 요약하자면 AWS AppSync는 GraphQL 스키마의 createPost 필드에 정의된 인수를 DynamoDB 작업으로 자동 변환합니다. 이 예제에서는 util.autoId() 도우미를 사용하여 자동으로 생성되는 id의 키를 사용하여 DynamoDB에 레코드를 저장합니다. AWS AppSync 콘솔이나 다른 방법으로 이루어진 요청에서 컨텍스트 인수(ctx.args.input)로 전달하는 다른 모든 필드는 테이블의 특성으로 저장됩니다. 키와 특성 모두 util.dynamodb.toMapValues(values) 도우미를 사용하여 호환되는 DynamoDB 형식에 자동으로 매핑됩니다.

AWS AppSync는 또한 해석기 편집을 위한 테스트 및 디버그 워크플로우를 지원합니다. 모의 `context` 객체를 사용하여 간접 호출 전에 템플릿의 변환된 값을 확인할 수 있습니다. 선택적으로 쿼리를 실행할 때 대화식으로 데이터 원본에 대한 전체 요청을 볼 수 있습니다. 자세한 내용은 해석기 테스트 및 디버깅(JavaScript) 및 모니터링 및 로깅을 참조하세요.

고급 해석기

스키마 설계의 선택적 페이지 매김 섹션을 따르는 경우에도 페이지 매김을 사용하려면 요청에 해석기를 추가해야 합니다. 이 예제에서는 한 번에 요청된 항목 중 일부만 반환하도록 `getPosts`라는 쿼리 페이지 매김을 사용했습니다. 해당 필드의 해석기 코드는 다음과 같을 수 있습니다.

```
/**
 * 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 };
}
```

요청에서는 요청의 컨텍스트를 전달합니다. `limit`은 *20*으로, 첫 번째 쿼리에서 최대 20개의 `Posts`를 반환합니다. `nextToken` 커서는 데이터 원본의 첫 번째 `Post` 항목에 고정됩니다. 이러한 값은 인수로 전달됩니다. 그런 다음 요청은 첫 번째 `Post`부터 시작하여 스캔 제한 횟수까지 스캔을 수행합니다. 데이터 원본은 결과를 컨텍스트에 저장하고, 이 결과는 응답에 전달됩니다. 응답은 검색된 `Posts`를 반환한 다음 `nextToken`을 한도 바로 뒤의 `Post` 항목으로 설정합니다. 정확히 동일한 작업을 수행하지만 첫 번째 쿼리 직후의 오프셋부터 시작하도록 다음 요청이 전송됩니다. 이러한 종류의 요청은 병렬로 수행되지 않고 순차적으로 수행된다는 점에 유의하세요.

해석기 테스트 및 디버그(JavaScript)

AWS AppSync는 데이터 소스에 대해 GraphQL 필드에서 해석기를 실행합니다. 파이프라인 해석기로 작업할 때 함수는 데이터 소스와 상호 작용합니다. JavaScript 해석기 개요 설명된 대로 함수는 JavaScript로 작성되고 `APPSYNC_JS` 런타임에서 실행되는 요청 및 응답 핸들러를 사용하여 데이터 소

스와 통신합니다. 이를 통해 데이터 소스와 통신하기 전과 후에 사용자 지정 로직 및 조건을 제공할 수 있습니다.

개발자가 이러한 해석기를 작성, 테스트 및 디버깅하기 쉽도록 하기 위해 AWS AppSync 콘솔은 개별 필드 해석기까지 모의 데이터와의 GraphQL 요청 및 응답을 생성하는 도구도 제공합니다. 뿐만 아니라, AWS AppSync 콘솔에서 쿼리, 뮤테이션 및 구독을 수행하고 Amazon CloudWatch의 전체 요청에 대한 자세한 로그 스트림을 볼 수 있습니다. 여기에는 데이터 소스의 결과가 포함됩니다.

모의 데이터를 사용하여 테스트

GraphQL 해석기가 호출되는 경우 이 해석기에는 요청에 대한 관련 정보를 포함하는 `context` 객체가 들어 있습니다. 여기에는 클라이언트의 인수, 자격 증명 정보 및 상위 GraphQL 필드의 데이터가 포함됩니다. 또한 데이터 소스의 결과를 저장하며, 이러한 결과는 응답 핸들러에서 사용할 수 있습니다. 이 구조와 프로그래밍 시 사용할 수 있는 헬퍼 유틸리티에 대한 자세한 내용은 해석기 컨텍스트 객체 참조 단원을 참조하세요.

해석기 함수를 작성하거나 편집할 때 mock 또는 test context 객체를 콘솔 편집기로 전달할 수 있습니다. 이렇게 하면 데이터 원본에 대해 실행하지 않고도 요청과 응답 핸들러가 어떻게 평가되는지 알아볼 수 있습니다. 예를 들면 테스트 `firstname: Shaggy` 인수를 전달하고 템플릿 코드에서 `ctx.args.firstname`을 사용하면 이 인수가 어떻게 평가되는지 볼 수 있습니다. 또한 `util.autoId()` 또는 `util.time.nowISO8601()` 같은 유틸리티 헬퍼의 평가도 테스트할 수 있습니다.

해석기 테스트

이 예제에서는 AWS AppSync 콘솔을 사용하여 해석기를 테스트합니다.

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
   a. API 대시보드에서 GraphQL API를 선택합니다.
   b. 사이드바에서 함수를 선택합니다.
2. 기존 함수를 선택합니다.
3. 함수 업데이트 페이지 상단에서 테스트 컨텍스트 선택을 선택한 다음 새 컨텍스트 생성을 선택합니다.
4. 샘플 컨텍스트 객체를 선택하거나 아래 테스트 컨텍스트 구성 창에서 JSON을 수동으로 채웁니다.
5. 텍스트 컨텍스트 이름을 입력합니다.
6. 저장 버튼을 선택합니다.
7. 이 모의 컨텍스트 객체를 사용하여 해석기를 평가하려면 테스트 실행을 선택합니다.

보다 실용적인 예로, 객체에 대한 자동 ID 생성을 사용하고 이를 Amazon DynamoDB에 저장하는 Dog의 GraphQL 유형을 저장하는 앱이 있다고 가정해 보겠습니다. 또한 GraphQL 뮤테이션의 인수에서 일부 값을 작성하고 특정 사용자만 응답을 볼 수 있도록 허용하려고 합니다. 다음 코드 조각은 스키마의 모양을 보여줍니다.

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

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

AWS AppSync 함수를 작성하고 이를 addDog 해석기에 추가하여 뮤테이션을 처리할 수 있습니다. AWS AppSync 함수를 테스트하기 위해 다음 예제와 같이 컨텍스트 객체를 채울 수 있습니다. 다음 객체에는 name 및 age 클라이언트의 인수와 identity 객체에 채워진 username이 있습니다.

```
{
    "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"
    }
}
```

다음 코드를 사용하여 AWS AppSync 함수를 테스트할 수 있습니다.

```
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();
}
```

평가된 요청 및 응답 핸들러에는 테스트 컨텍스트 객체의 데이터와 `util.autoId()`에서 생성된 값이 포함됩니다. 또한, `username`을 `Nadia` 이외의 값으로 변경해야 했다면 권한 부여 확인이 실패하므로 결과가 반환되지 않습니다. 세분화된 액세스 제어에 대한 자세한 내용은 권한 부여 사용 사례 단원을 참조하세요.

AWS AppSync의 API를 사용하여 요청 및 응답 핸들러 테스트

`EvaluateCode` API 명령어를 사용하여 모의 데이터로 코드를 원격 테스트할 수 있습니다. 명령어를 시작하려면 정책에 `appsync:evaluateMappingCode` 권한을 추가했는지 확인하세요. 예:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateCode",
            "Resource": "arn:aws:appsync:<region>:<account>:*"
        }
    ]
}
```

AWS CLI 또는 AWS SDK를 사용하여 명령을 활용할 수 있습니다. 예를 들어 이전 섹션의 Dog 스키마와 해당 AWS AppSync 함수 요청 및 응답 핸들러를 살펴보겠습니다. 로컬 스테이션에서 CLI를 사용하여 코드를 `code.js`라는 파일에 저장한 다음 `context` 객체를 `context.json`이라는 파일에 저장합니다. 쉘에서 다음 명령을 실행합니다.

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

응답에는 핸들러가 반환한 페이로드가 포함된 evaluationResult가 포함되어 있습니다. 또한 평가 중에 핸들러가 생성한 로그 목록이 들어 있는 logs 객체도 포함되어 있습니다. 이렇게 하면 코드 실행을 쉽게 디버깅하고 평가에 대한 정보를 확인하여 문제를 해결하는 데 도움이 됩니다. 예:

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

evaluationResult를 JSON으로 구문 분석하여 다음과 같은 결과를 얻을 수 있습니다.

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

SDK를 사용하면 자주 사용하는 테스트 제품군의 테스트를 쉽게 통합하여 핸들러의 동작을 검증할 수 있습니다. Jest 테스트 프레임워크를 사용하여 테스트를 생성하는 것을 권장하지만, 어떤 테스트 도구도 사용 가능합니다. 다음 코드 조각은 가상의 검증 실행을 보여줍니다. 평가 응답은 유효한 JSON일 것으로 예상하므로 JSON.parse를 사용하여 문자열 응답에서 JSON을 검색합니다.

```
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)
})
```

이 결과는 다음과 같아야 합니다.

```
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
```

라이브 쿼리 디버깅

프로덕션 애플리케이션을 디버깅하기 위한 종단 간 테스트 및 로깅은 어떤 것으로도 대체할 수 없습니다. AppSync에서는 Amazon CloudWatch를 사용하여 오류 및 전체 요청 세부 정보를 기록할 수 있습니다. 또한 AWS AppSync 콘솔을 사용하여 GraphQL 쿼리, 뮤테이션 및 구독을 테스트하고 각 요청에 대한 로그 데이터를 쿼리 편집기에 실시간으로 스트리밍하여 실시간으로 디버깅할 수 있습니다. 구독의 경우 로그는 연결-시간 정보를 표시합니다.

이를 수행하려면 모니터링 및 로깅에 설명된 대로 Amazon CloudWatch 로그를 미리 활성화해야 합니다. 다음에는 AWS AppSync 콘솔에서 쿼리 탭을 선택한 다음 유효한 GraphQL 쿼리를 입력합니다. 오른쪽 하단 섹션에서 로그 창을 클릭하고 드래그하여 로그 보기를 엽니다. 페이지 맨 위에 있는 "재생" 화살표 아이콘을 선택하여 GraphQL 쿼리를 실행합니다. 몇 분 후 작업에 대한 전체 요청 및 응답 로그가 이 섹션으로 스트리밍되어 사용자가 콘솔에서 볼 수 있습니다.

파이프라인 해석기(JavaScript)

AWS AppSync는 GraphQL 필드에서 해석기를 실행합니다. 경우에 따라 애플리케이션 사용 시 단일 GraphQL 필드를 해석하기 위해 여러 작업을 실행해야 합니다. 이제, 파이프라인 해석기를 사용해 개발자는 함수라고 하는 작업을 작성하고 순서대로 실행할 수 있습니다. 파이프라인 해석기는 예를 들어, 필드에 필요한 데이터를 가져오기 전에 권한 부여를 확인해야 하는 애플리케이션에 유용합니다.

JavaScript 파이프라인 해석기의 아키텍처에 대한 자세한 내용은 JavaScript 해석기 개요를 참조하세요.

파이프라인 해석기 생성

AWS AppSync 콘솔에서 스키마 페이지로 이동합니다.

다음 스키마를 저장합니다.

```
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
}
```

변형 유형의 signUp 필드에 파이프라인 해석기를 연결하겠습니다. 오른쪽에 있는 변형 유형에서 `signUp` 변형 필드 옆에 있는 연결을 선택합니다. 해석기를 `pipeline resolver`로 설정하고 `APPSYNC_JS` 런타임을 설정한 다음 해석기를 생성합니다.

파이프라인 해석기는 사용자를 등록해 먼저 입력된 이메일 주소를 확인한 다음 사용자를 시스템에 저장합니다. 이메일 검증을 validateEmail 함수 내에 캡슐화하고 사용자를 saveUser 함수 내에 저장할 것입니다. validateEmail 함수가 먼저 실행되고, 이메일이 유효한 경우 saveUser 함수가 실행됩니다.

다음과 같이 실행 흐름이 진행됩니다.

1. Mutation.signUp 해석기 요청 핸들러
2. validateEmail 함수

3. saveUser 함수
4. Mutation.signUp 해석기 응답 핸들러

API의 다른 해석기 내에서 validateEmail 함수를 다시 사용할 것입니다. GraphQL 필드 간에 변경되기 때문에 `ctx.args`에는 액세스하지 않도록 하겠습니다. 대신 `ctx.stash`를 사용해 `signUp(input: Signup)` 입력 필드 인수의 이메일 속성을 저장할 수 있습니다.

요청 및 응답 함수를 바꿔서 해석기 코드를 업데이트하세요.

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

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

생성 또는 저장을 선택하여 해석기를 업데이트합니다.

함수 생성

파이프라인 해석기 페이지의 함수 섹션에서 함수 추가를 클릭하고 새 함수 생성을 클릭합니다. 해석기 페이지를 거치지 않고 함수를 생성할 수 있는데, 이렇게 하려면 AWS AppSync 콘솔에서 함수 페이지로 이동합니다. 함수 생성 버튼을 선택합니다. 이메일이 유효하고 특정 도메인에서 전송되는지 확인하는 함수를 생성합니다. 이메일이 유효하지 않으면 함수에서 오류가 발생합니다. 이메일이 유효할 경우 제공된 입력을 전달합니다.

없음 유형의 데이터 원본을 생성했는지 확인합니다. 데이터 원본 이름 목록에서 이 데이터 원본을 선택합니다. 함수 이름에 `validateEmail`을 입력합니다. 함수 코드 영역에서 다음 스니펫으로 모든 내용을 덮어씁니다.

```
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;
}
```

입력 내용을 검토한 다음 생성을 선택합니다. 방금 validateEmail 함수를 생성했습니다. 이 단계를 반복하여 다음 코드로 saveUser 함수를 생성합니다(단순화를 위해 없음 데이터 원본을 사용하고 함수가 실행된 후 사용자가 시스템에 저장된 것으로 가장함).

```
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;
}
```

방금 saveUser 함수를 생성했습니다.

파이프라인 해석기에 함수 추가

파이프라인 해석기에 방금 생성한 함수가 자동으로 추가되어야 합니다. 그렇지 않거나 함수 페이지를 통해 함수를 생성한 경우, signUp 해석기 페이지에서 함수 추가를 다시 클릭하여 연결할 수 있습니다. 해석기에 validateEmail 및 saveUser 함수를 모두 추가합니다. validateEmail 함수는 saveUser 함수 앞에 배치해야 합니다. 함수를 더 추가할 때 위로 이동 및 아래로 이동 옵션을 사용해 함수 실행 순서를 다시 정리할 수 있습니다. 변경 사항을 검토한 후 저장을 선택합니다.

쿼리 실행

AWS AppSync 콘솔에서 쿼리 페이지로 이동합니다. 탐색기에서 변형을 사용하고 있는지 확인합니다. 그렇지 않다면 드롭다운 목록에서 Mutation을 선택한 다음 +를 선택합니다. 다음 쿼리를 입력합니다.

```
mutation {
```

```
  signUp(input: {email: "nadia@myvaliddomain.com", username: "nadia"}) {
    id
    username
  }
}
```

다음과 유사하게 반환되어야 합니다.

```
{
  "data": {
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "username": "nadia"
    }
  }
}
```

파이프라인 해석기를 사용하여 성공적으로 사용자를 등록했고, 입력한 이메일을 검증했습니다.

해석기 구성(VTL)

> **Note**
>
> 이제는 APPSYNC_JS 런타임과 해당 문서를 주로 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

GraphQL 해석기는 형식 스키마의 필드를 데이터 원본에 연결합니다. 해석기는 요청을 이행하는 메커니즘입니다. AWS 코드를 작성할 필요 없이 AppSync에서는 스키마로부터 해석기를 자동으로 생성해 연결하거나 기존 테이블에서 스키마를 생성해 해석기를 연결할 수 있습니다.

AWS AppSync의 해석기는 JavaScript를 사용하여 GraphQL 식을 데이터 원본에서 사용할 수 있는 형식으로 변환합니다. 또는 Apache Velocity Template Language(VTL)로 작성된 매핑 템플릿을 사용하여 GraphQL 식을 데이터 원본이 사용할 수 있는 형식으로 변환할 수 있습니다.

이 섹션에서는 VTL을 사용하여 해석기를 구성하는 방법을 보여 줍니다. 해석기 작성을 위한 소개 자습서 스타일의 프로그래밍 가이드는 해석기 매핑 템플릿 프로그래밍 안내서에서 찾을 수 있으며 프로그래밍 시 사용할 수 있는 헬퍼 유틸리티는 해석기 매핑 템플릿 컨텍스트 참조에서 찾을 수 있습니다. AWS 또한 AppSync에는 처음부터 편집 또는 작성할 때 사용할 수 있는 기본 테스트 및 디버깅 흐름이 포함되어 있습니다. 자세한 내용은 해석기 테스트 및 디버깅을 참조하세요.

앞서 언급한 자습서를 사용하기 전에 이 안내서를 확인하는 것이 좋습니다.

이 섹션에서는 해석기를 생성하고, 변형에 대해 해석기를 추가하고, 고급 구성을 사용하는 방법을 알아봅니다.

첫 번째 해석기 생성

이전 섹션의 예제에 따라 첫 번째 단계는 `Query` 유형에 맞는 해석기를 만드는 것입니다.

Console

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
   a. API 대시보드에서 GraphQL API를 선택합니다.
   b. 사이드바에서 스키마를 선택합니다.
2. 페이지 오른쪽에 해석기라는 창이 있습니다. 이 상자에는 페이지 왼쪽의 스키마 창에 정의된 유형 및 필드 목록이 있습니다. 필드에 해석기를 연결할 수 있습니다. 예를 들어 쿼리 유형에서 `getTodos` 필드 옆에 있는 첨부를 선택합니다.
3. 해석기 생성 페이지에서 데이터 원본 연결 안내서에서 만든 데이터 원본을 선택합니다. 매핑 템플릿 구성 창에서 오른쪽의 드롭다운 목록을 사용하여 일반 요청 및 응답 매핑 템플릿을 모두 선택하거나 직접 작성할 수 있습니다.

   > **Note**
   >
   > 요청 매핑 템플릿과 응답 매핑 템플릿의 쌍을 이루는 것을 단위 해석기라고 합니다. 단위 해석기는 일반적으로 기계적 작업을 수행하는 것이 목적이므로 데이터 원본 수가 적은 단일 작업에만 사용하는 것이 좋습니다. 더 복잡한 작업의 경우 여러 데이터 원본에서 순차적으로 여러 작업을 실행할 수 있는 파이프라인 해석기를 사용하는 것이 좋습니다.
   > 요청과 응답 매핑 템플릿의 차이점에 대한 자세한 정보는 단위 해석기를 참조하세요. 파이프라인 해석기 사용에 대한 자세한 내용은 파이프라인 해석기를 참조하세요.

4. 일반적으로 AWS AppSync 콘솔에는 데이터 원본에서 항목을 가져오는 데 사용할 수 있는 내장 템플릿이 있습니다(예: 모든 항목 쿼리, 개별 조회 등). 예를 들어 `getTodos`에 페이지 매김이 없는 스키마 설계의 간단한 스키마 버전에서 항목을 나열하기 위한 요청 매핑 템플릿은 다음과 같습니다.

```
{
    "version" : "2017-02-28",
```

```
    "operation" : "Scan"
}
```

5. 요청과 동반되는 응답 매핑 템플릿은 항상 필요합니다. 콘솔은 다음과 같은 목록에 대한 패스스루 값과 함께 기본값을 제공합니다.

```
$util.toJson($ctx.result.items)
```

이 예제에서 항목의 목록에 대한 context 객체(별칭 $ctx)는 $context.result.items 양식을 갖습니다. GraphQL 작업이 단일 항목을 반환하는 경우 이는 $context.result입니다. AWS AppSync는 이전에 나열된 $util.toJson 함수와 같은 일반 작업에 대해 헬퍼 함수를 제공하여 적절한 응답 형식을 지정합니다. 전체 함수 목록은 해석기 매핑 템플릿 유틸리티 참조를 참조하세요.

6. 해석기 저장 을 선택합니다.

API

1. CreateResolver API를 호출하여 해석기 객체를 생성합니다.
2. UpdateResolver API를 호출하여 해석기의 필드를 수정할 수 있습니다.

CLI

1. create-resolver 명령을 실행하여 해석기를 생성합니다.

   이 특정 명령에 대해 6개의 파라미터를 입력해야 합니다.

   1. API의 api-id.
   2. 스키마에서 수정하려는 유형의 type-name. 콘솔 예제에서는 Query였습니다.
   3. 유형에서 수정하려는 필드의 field-name. 콘솔 예제에서는 getTodos였습니다.
   4. 데이터 원본 연결 안내서에서 만든 데이터 원본의 data-source-name.
   5. 요청의 본문인 request-mapping-template. 콘솔 예제에서는 다음과 같았습니다.

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
}
```

   6. 응답의 본문인 response-mapping-template. 콘솔 예제에서는 다음과 같았습니다.

```
$util.toJson($ctx.result.items)
```

예를 들어 명령은 다음과 같을 수 있습니다.

```
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)"
```

CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

```
{
    "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. 해석기의 필드 또는 매핑 템플릿을 수정하려면 `update-resolver` 명령을 실행합니다.

   `api-id` 파라미터를 제외하고 `create-resolver` 명령에 사용된 파라미터는 `update-resolver` 명령의 새 값으로 덮어써집니다.

변형에 대한 해석기 추가

다음 단계는 `Mutation` 유형에 맞는 해석기를 생성하는 것입니다.

Console

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.

   a. API 대시보드에서 GraphQL API를 선택합니다.

   b. 사이드바에서 스키마를 선택합니다.

2. 변형 유형에서 addTodo 필드 옆에 있는 연결을 선택합니다.
3. 해석기 생성 페이지에서 데이터 원본 연결 안내서에서 만든 데이터 원본을 선택합니다.
4. DynamoDB에 새 항목을 추가하는 경우 요청 템플릿이 변형이므로 매핑 템플릿 구성 창에서 요청 템플릿을 수정해야 합니다. 다음 요청 매핑 템플릿을 사용합니다.

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

5. AWS AppSync는 GraphQL 스키마의 addTodo 필드에 정의된 인수를 DynamoDB 작업으로 자동으로 변환합니다. 이전 예에서는 변형 인수에서 전달된 id의 키를 $ctx.args.id로 사용하여 DynamoDB에 레코드를 저장합니다. 전달하는 다른 모든 필드는 $util.dynamodb.toMapValuesJson($ctx.args)을 사용하여 DynamoDB 특성에 자동으로 매핑됩니다.

   이 해석기의 경우 다음 응답 매핑 템플릿을 사용합니다.

```
$util.toJson($ctx.result)
```

   AWS AppSync는 또한 해석기 편집을 위한 테스트 및 디버그 워크플로를 지원합니다. 모의 context 객체를 사용하여 호출 전에 템플릿의 변환된 값을 확인할 수 있습니다. 선택적으로 쿼리를 실행할 때 대화식으로 데이터 원본에 대한 전체 요청 실행을 볼 수 있습니다. 자세한 내용은 해석기 테스트 및 디버깅 및 모니터링 및 로깅을 참조하세요.
6. 해석기 저장 을 선택합니다.

API

첫 번째 해석기 생성 섹션의 명령과 이 섹션의 파라미터 세부 정보를 활용하여 API로 이 작업을 수행할 수도 있습니다.

CLI

첫 번째 해석기 생성 섹션의 명령과 이 섹션의 파라미터 세부 정보를 활용하여 CLI에서 이 작업을 수행할 수도 있습니다.

이 시점에서 고급 해석기를 사용하지 않는 경우 API 사용에서 설명하는 것과 같이 GraphQL API 사용을 시작할 수 있습니다.

고급 해석기

고급 섹션을 따르고 있으며, 스키마 설계의 샘플 스키마를 빌드하여 페이지 지정 스캔을 수행하는 경우 getTodos 필드에 대한 다음 요청 템플릿을 대신 사용합니다.

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "limit": $util.defaultIfNull(${ctx.args.limit}, 20),
    "nextToken": $util.toJson($util.defaultIfNullOrBlank($ctx.args.nextToken, null))
}
```

이러한 페이지 매김 사용 사례의 경우 응답 매핑에는 커서가 포함되어 있고(따라서 클라이언트가 다음에 시작할 페이지를 알 수 있음) 결과 집합이 포함되어 있어야 하기 때문에 응답 매핑은 단순한 패스스루 이상입니다. 매핑 템플릿은 다음과 같습니다.

```
{
    "todos": $util.toJson($context.result.items),
    "nextToken": $util.toJson($context.result.nextToken)
}
```

이전 응답 매핑 템플릿의 필드가 TodoConnection 형식에 정의된 필드와 일치해야 합니다.

Comments 테이블이 있고 Todo 형식에 대한 주석 필드를 해석하는 관계의 경우([Comment]의 형식 반환) 두 번째 테이블을 기준으로 쿼리를 실행하는 매핑 템플릿을 사용할 수 있습니다. 이를 수행하려면 데이터 원본 연결에서 설명하는 것과 같이 Comments 테이블에 대한 데이터 원본이 이미 생성되어야 합니다.

> **Note**
>
> 두 번째 테이블을 기준으로 쿼리 작업을 사용하는 것은 설명을 돕기 위한 목적일 뿐입니다. 대신 DynamoDB에 대해 다른 작업을 사용할 수 있습니다. 뿐만 아니라 관계를 GraphQL 스키마에서 제어하기 때문에 AWS Lambda 또는 Amazon OpenSearch Service 등과 같은 다른 데이터 원본에서 데이터를 가져올 수도 있습니다.

Console

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.

   a. API 대시보드에서 GraphQL API를 선택합니다.

   b. 사이드바에서 스키마를 선택합니다.

2. Todo 유형에서 comments 필드 옆에 있는 연결을 선택합니다.
3. 해석기 생성 페이지에서 주석 테이블 데이터 원본을 선택합니다. 빠른 시작 안내서에서 주석 테이블의 기본 이름은 AppSyncCommentTable이지만, 지정한 이름에 따라 달라질 수 있습니다.
4. 요청 매핑 템플릿에 다음 스니펫을 추가합니다.

```
{
    "version": "2017-02-28",
    "operation": "Query",
    "index": "todoid-index",
    "query": {
        "expression": "todoid = :todoid",
        "expressionValues": {
            ":todoid": {
                "S": $util.toJson($context.source.id)
            }
        }
    }
}
```

5. context.source는 해석 중인 현재 필드의 상위 객체를 참조합니다. 이 예제에서 source.id는 개별 Todo 객체를 참조하고, 이는 쿼리 표현식에 사용됩니다.

   다음과 같이 패스스루 응답 매핑 템플릿을 사용할 수 있습니다.

```
$util.toJson($ctx.result.items)
```

6. 해석기 저장 을 선택합니다.
7. 마지막으로 콘솔의 스키마 페이지로 돌아가서 addComment 필드에 해석기를 연결하고, Comments 테이블에 대한 데이터 원본을 지정합니다. 이 경우의 요청 매핑 템플릿은 인수로서 주석 처리된 특정 todoid가 포함된 단순한 PutItem이지만, $utils.autoId() 유틸리티를 사용하여 다음과 같이 주석에 대한 고유한 정렬 키를 생성할 수 있습니다.

```
{
```

```
    "version": "2017-02-28",
    "operation": "PutItem",
    "key": {
        "todoid": { "S": $util.toJson($context.arguments.todoid) },
        "commentid": { "S": "$util.autoId()" }
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

다음과 같이 패스스루 응답 템플릿을 사용합니다.

```
$util.toJson($ctx.result)
```

API

첫 번째 해석기 생성 섹션의 명령과 이 섹션의 파라미터 세부 정보를 활용하여 API로 이 작업을 수행할 수도 있습니다.

CLI

첫 번째 해석기 생성 섹션의 명령과 이 섹션의 파라미터 세부 정보를 활용하여 CLI에서 이 작업을 수행할 수도 있습니다.

직접 Lambda 해석기(VTL)

> **Note**
>
> 이제는 APPSYNC_JS 런타임과 해당 문서를 주로 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

직접 Lambda 해석기를 이용하면 AWS Lambda 데이터 원본 사용 시 VTL 매핑 템플릿의 사용을 방지할 수 있습니다. AWS AppSync는 Lambda 함수에 기본 페이로드뿐만 아니라 GraphQL 형식에 대한 Lambda 함수 응답의 기본 변환도 제공할 수 있습니다. 요청 템플릿을 제공할지, 응답 템플릿을 제공할지 또는 제공하지 않을지 선택할 수 있으며 AWS AppSync는 그에 따라 처리합니다.

AWS AppSync에서 제공하는 기본 요청 페이로드 및 응답 번역에 대해 자세히 알아보려면 직접 Lambda 해석기 참조를 참조하세요. AWS Lambda 데이터 원본 설정 및 IAM 신뢰 정책 설정에 대한 자세한 내용은 데이터 원본 연결을 참조하세요.

직접 Lambda 해석기 구성

다음 섹션에서는 Lambda 데이터 원본을 연결하고 Lambda 해석기를 필드에 추가하는 방법을 보여 줍니다.

Lambda 데이터 원본 추가

직접 Lambda 해석기를 활성화하려면 먼저 Lambda 데이터 원본을 추가해야 합니다.

Console

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
   a. API 대시보드에서 GraphQL API를 선택합니다.
   b. 사이드바에서 데이터 원본을 선택합니다.
2. 데이터 원본 생성을 선택합니다.
   a. 데이터 원본 이름에 데이터 원본의 이름(예: **myFunction**)을 입력합니다.
   b. 데이터 원본 유형에 AWS Lambda 함수를 선택합니다.
   c. 리전에서 적절한 리전을 선택합니다.
   d. 함수 ARN에는 드롭다운 목록에서 Lambda 함수를 선택합니다. 함수 이름을 검색하거나 사용하려는 함수의 ARN을 직접 입력할 수 있습니다.
   e. 새 IAM 역할을 만들거나(권장) `lambda:invokeFunction` IAM 권한이 있는 기존 역할을 선택합니다. 기존 역할에는 데이터 원본 연결 섹션에 설명된 대로 신뢰 정책이 필요합니다.

      다음은 리소스에서 작업을 수행하는 데 필요한 권한을 가진 IAM 정책의 예입니다.

```
{
     "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. 생성 버튼을 선택합니다.

CLI

1. create-data-source 명령을 실행하여 데이터 원본 객체를 생성합니다.

   이 특정 명령에 대해 4개의 파라미터를 입력해야 합니다.

   1. API의 api-id.
   2. 데이터 원본의 name. 콘솔 예시에서는 데이터 원본 이름입니다.
   3. 데이터 원본의 type. 콘솔 예시에서는 AWS Lambda 함수입니다.
   4. lambda-config(콘솔 예시에서는 함수 ARN).

   > **Note**
   >
   > 필수로 구성해야 하지만 일반적으로 CLI 구성 값으로 기본 설정되는 Region과 같은 다른 파라미터도 있습니다.

   예를 들어 명령은 다음과 같을 수 있습니다.

```
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
```

   CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

```
{
    "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. 데이터 원본의 특성을 수정하려면 update-data-source 명령을 실행합니다.

   api-id 파라미터를 제외하고 create-data-source 명령에 사용된 파라미터는 update-data-source 명령의 새 값으로 덮어써집니다.

직접 Lambda 해석기 활성화

Lambda 데이터 원본을 생성하고 AWS AppSync가 함수를 간접적으로 호출할 수 있도록 적절한 IAM 역할을 설정하고 나면 해석기 또는 파이프라인 함수에 연결할 수 있습니다.

Console

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
   a. API 대시보드에서 GraphQL API를 선택합니다.
   b. 사이드바에서 스키마를 선택합니다.
2. 해석기 창에서 필드 또는 작업을 선택한 다음 연결 버튼을 선택합니다.
3. 새 해석기 생성 페이지의 드롭다운 목록에서 Lambda 함수를 선택합니다.
4. 직접 Lambda 해석기를 활용하려면 매핑 템플릿 구성 섹션에서 요청 및 응답 매핑 템플릿이 비활성화되었는지 확인하세요.
5. 해석기 저장 버튼을 선택합니다.

CLI

- create-resolver 명령을 실행하여 해석기를 생성합니다.

  이 특정 명령에 대해 6개의 파라미터를 입력해야 합니다.

  1. API의 api-id.
  2. 스키마에 있는 형식의 type-name.
  3. 스키마에 있는 필드의 field-name.
  4. data-source-name 또는 Lambda 함수의 이름.

5. 요청의 본문인 `request-mapping-template`. 콘솔 예시에서는 이 파라미터가 비활성화되어 있었습니다.

```
" "
```

6. 응답의 본문인 `response-mapping-template`. 콘솔 예시에서는 이 파라미터도 비활성화되어 있었습니다.

```
" "
```

예를 들어 명령은 다음과 같을 수 있습니다.

```
aws appsync create-resolver --api-id abcdefghijklmnopqrstuvwxyz --type-name
 Subscription --field-name onCreateTodo --data-source-name LambdaTest --request-
mapping-template " " --response-mapping-template " "
```

CLI에서 출력이 반환됩니다. 다음은 그 예입니다:

```
{
    "resolver": {
        "resolverArn": "arn:aws:appsync:us-west-2:102847592837:apis/
abcdefghijklmnopqrstuvwxyz/types/Subscription/resolvers/onCreateTodo",
        "typeName": "Subscription",
        "kind": "UNIT",
        "fieldName": "onCreateTodo",
        "dataSourceName": "LambdaTest"
    }
}
```

매핑 템플릿을 비활성화하면 몇 가지 추가 동작이 AWS AppSync에서 발생합니다.

- 매핑 템플릿을 비활성화하면 직접 Lambda 해석기 참조에 지정된 기본 데이터 번역을 수락한다는 신호를 AWS AppSync에 보내는 것입니다.
- 요청 매핑 템플릿을 비활성화하면 Lambda 데이터 원본이 전체 컨텍스트 객체로 구성된 페이로드를 수신합니다.
- 응답 매핑 템플릿을 비활성화하면 요청 매핑 템플릿의 버전 또는 요청 매핑 템플릿이 비활성화되었는지 여부에 따라 Lambda 간접 호출의 결과가 번역됩니다.

테스트 및 디버그 해석기(VTL)

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync는 데이터 소스에 대해 GraphQL 필드에서 해석기를 실행합니다. 해석기 매핑 템플릿 개요에서 설명한 바와 같이, 해석기는 템플릿 언어를 사용하여 데이터 소스와 통신합니다. 이를 통해 데이터 소스와 통신하기 전과 후에 동작을 사용자 지정하고 로직 및 조건을 적용할 수 있습니다. 해석기 작성을 설명하는 소개 자습서 스타일의 프로그래밍 가이드는 해석기 매핑 템플릿 프로그래밍 가이드를 참조하세요.

개발자가 이러한 해석기를 작성, 테스트 및 디버깅하기 쉽도록 하기 위해 AWS AppSync 콘솔은 개별 필드 해석기까지 모의 데이터와의 GraphQL 요청 및 응답을 생성하는 도구도 제공합니다. 뿐만 아니라, AWS AppSync 콘솔에서 쿼리, 뮤테이션 및 구독을 수행하고 Amazon CloudWatch에서 전체 요청에 대한 자세한 로그 스트림을 볼 수 있습니다. 여기에는 데이터 원본의 결과가 포함됩니다.

모의 데이터를 사용하여 테스트

GraphQL 해석기가 호출되는 경우 이 해석기에는 요청에 대한 정보를 포함하는 `context` 객체가 들어 있습니다. 여기에는 클라이언트의 인수, 자격 증명 정보 및 상위 GraphQL 필드의 데이터가 포함됩니다. 또한 데이터 소스의 결과가 포함되며, 이러한 결과는 응답 템플릿에서 사용할 수 있습니다. 이 구조와 프로그래밍 시 사용할 수 있는 헬퍼 유틸리티에 대한 자세한 내용은 해석기 매핑 템플릿 컨텍스트 참조 단원을 참조하십시오.

해석기를 작성하거나 편집할 때 mock 또는 test context 객체를 콘솔 편집기로 전달할 수 있습니다. 이렇게 하면 데이터 원본에 대해 실행하지 않고도 요청과 응답 템플릿이 어떻게 평가되는지 알아볼 수 있습니다. 예를 들면 테스트 `firstname: Shaggy` 인수를 전달하고 템플릿 코드에서 `$ctx.args.firstname`을 사용하면 이 인수가 어떻게 평가되는지 볼 수 있습니다. 또한 `$util.autoId()` 또는 `util.time.nowISO8601()` 같은 유틸리티 헬퍼의 평가도 테스트할 수 있습니다.

해석기 테스트

이 예제에서는 AWS AppSync 콘솔을 사용하여 해석기를 테스트합니다.

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.

   a. API 대시보드에서 GraphQL API를 선택합니다.

b. 사이드바에서 스키마를 선택합니다.

2. 아직 추가하지 않았다면 유형 아래 및 필드 옆에서 첨부를 선택하여 해석기를 추가합니다.

   완전한 해석기를 빌드하는 방법에 대한 자세한 내용은 해석기 구성을 참조하세요.

   그렇지 않으면 필드에 이미 있는 해석기를 선택합니다.

3. 해석기 편집 페이지 상단에서 테스트 컨텍스트 선택을 선택한 다음 새 컨텍스트 생성을 선택합니다.
4. 샘플 컨텍스트 객체를 선택하거나 아래 실행 컨텍스트 창에서 JSON을 수동으로 채웁니다.
5. 텍스트 컨텍스트 이름에 입력합니다.
6. 저장 버튼을 선택합니다.
7. 해석기 편집 페이지 상단에서 테스트 실행을 선택합니다.

보다 실용적인 예로, 객체에 대한 자동 ID 생성을 사용하고 이를 Amazon DynamoDB에 저장하는 Dog의 GraphQL 유형을 저장하는 앱이 있다고 가정해 보겠습니다. 또한 GraphQL 변형의 인수에서 일부 값을 작성하고 특정 사용자만 응답을 볼 수 있도록 허용하려고 합니다. 스키마의 모양은 다음과 같을 수 있습니다.

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

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

addDog 뮤테이션의 해석기를 추가할 경우 다음의 예와 같이 컨텍스트 객체를 채울 수 있습니다. 다음 객체에는 name 및 age 클라이언트의 인수와 identity 객체에 채워진 username이 있습니다.

```
{
    "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"
    }
}
```

다음 요청과 응답 매핑 템플릿을 사용하여 이것을 테스트할 수 있습니다.

요청 템플릿

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : { "S" : "$util.autoId()" }
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

응답 템플릿

```
#if ($context.identity.username == "Nadia")
  $util.toJson($ctx.result)
#else
  $util.unauthorized()
#end
```

평가된 템플릿에는 테스트 컨텍스트 객체의 데이터와 `$util.autoId()`에서 생성된 값이 포함됩니다. 또한, `username`을 `Nadia` 이외의 값으로 변경해야 했다면 권한 부여 확인이 실패하므로 결과가 반환되지 않습니다. 세분화된 액세스 제어에 대한 자세한 내용은 권한 부여 사용 사례 단원을 참조하세요.

AWS AppSync의 API를 사용하여 매핑 템플릿을 테스트합니다.

`EvaluateMappingTemplate` API 명령어를 사용하여 모의 데이터로 매핑 템플릿을 원격 테스트할 수 있습니다. 명령어를 시작하려면 정책에 `appsync:evaluateMappingTemplate` 권한을 추가했는지 확인하세요. 예:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateMappingTemplate",
            "Resource": "arn:aws:appsync:<region>:<account>:*"
        }
    ]
}
```

AWS CLI 또는 AWS SDK를 사용하여 명령을 활용할 수 있습니다. 예를 들어 이전 섹션의 Dog 스키마와 해당 요청/응답 매핑 템플릿을 가져옵니다. 로컬 스테이션에서 CLI를 사용하여 요청 템플릿을 `request.vtl`라는 파일에 저장한 다음 `context` 객체를 `context.json`이라는 파일에 저장합니다. 쉘에서 다음 명령을 실행합니다.

```
aws appsync evaluate-mapping-template --template file://request.vtl --context file://
context.json
```

이 명령은 다음 응답을 반환합니다.

```
{
  "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"
}
```

`evaluationResult`에는 제공된 `context`를 사용하여 제공된 템플릿을 테스트한 결과가 포함되어 있습니다. AWS SDK를 사용하여 템플릿을 테스트할 수도 있습니다. 다음은 JavaScript V2용 AWS SDK를 사용하는 예제입니다.

```
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))
```

SDK를 사용하면 자주 사용하는 테스트 제품군의 테스트를 쉽게 통합하여 템플릿의 동작을 검증할 수 있습니다. Jest 테스트 프레임워크를 사용하여 테스트를 생성하는 것을 권장하지만, 어떤 테스트 도구도 사용 가능합니다. 다음 코드 조각은 가상의 검증 실행을 보여줍니다. 평가 응답은 유효한 JSON일 것으로 예상하므로 `JSON.parse`를 사용하여 문자열 응답에서 JSON을 검색합니다.

```
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)
})
```

이 결과는 다음과 같아야 합니다.

```
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
```

라이브 쿼리 디버깅

프로덕션 애플리케이션을 디버깅하기 위한 종단 간 테스트 및 로깅은 어떤 것으로도 대체할 수 없습니다. AppSync에서는 Amazon CloudWatch를 사용하여 오류 및 전체 요청 세부 정보를 기록할 수 있습니다. 또한 AWS AppSync 콘솔을 사용하여 GraphQL 쿼리, 뮤테이션 및 구독을 테스트하고 각 요청에 대한 로그 데이터를 쿼리 편집기에 실시간으로 스트리밍하여 실시간으로 디버깅할 수 있습니다. 구독의 경우 로그는 연결-시간 정보를 표시합니다.

이를 수행하려면 모니터링 및 로깅에 설명된 대로 Amazon CloudWatch 로그를 미리 활성화해야 합니다. 다음에는 AWS AppSync 콘솔에서 쿼리 탭을 선택한 다음 유효한 GraphQL 쿼리를 입력합니다. 오른쪽 하단 섹션에서 로그 창을 클릭하고 드래그하여 로그 보기를 엽니다. 페이지 맨 위에 있는 "재생" 화살표 아이콘을 선택하여 GraphQL 쿼리를 실행합니다. 몇 분 후 작업에 대한 전체 요청 및 응답 로그가 이 섹션으로 스트리밍되어 사용자가 콘솔에서 볼 수 있습니다.

파이프라인 해석기(VTL)

> **Note**
>
> 이제는 APPSYNC_JS 런타임과 해당 문서를 주로 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync는 GraphQL 필드에서 해석기를 실행합니다. 경우에 따라 애플리케이션 사용 시 단일 GraphQL 필드를 해석하기 위해 여러 작업을 실행해야 합니다. 이제, 파이프라인 해석기를 사용해 개발자는 함수라고 하는 작업을 작성하고 순서대로 실행할 수 있습니다. 파이프라인 해석기는 예를 들어, 필드에 필요한 데이터를 가져오기 전에 권한 부여를 확인해야 하는 애플리케이션에 유용합니다.

파이프라인 해석기는 이전 매핑 템플릿, 이후 매핑 템플릿과 함수 목록으로 구성됩니다. 각 함수에는 데이터 원본에 대해 실행되는 요청 및 응답 매핑 템플릿이 있습니다. 파이프라인 해석기는 함수 목록에 실행을 위임하기 때문에 자신은 어떠한 데이터 원본에도 연결되지 않습니다. 단위 해석기와 함수는 데이터 원본에 대해 작업을 실행하는 기본 요소입니다. 자세한 내용은 해석기 매핑 템플릿 개요를 참조하세요.

파이프라인 해석기 생성

AWS AppSync 콘솔에서 스키마 페이지로 이동합니다.

다음 스키마를 저장합니다.

```
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
}
```

변형 유형의 signUp 필드에 파이프라인 해석기를 연결하겠습니다. 오른쪽에 있는 변형 유형에서 signUp 변형 필드 옆에 있는 연결을 선택합니다. 해석기 생성 페이지에서 작업을 클릭한 다음 런타임 업데이트를 클릭합니다. Pipeline Resolver를 선택하고 VTL을 선택한 다음 업데이트를 선택합니다. 이 페이지에는 사전 매핑 템플릿 텍스트 영역, 함수 섹션 및 사후 매핑 템플릿 텍스트 영역, 이렇게 3가지 섹션이 있어야 합니다.

파이프라인 해석기는 사용자를 등록해 먼저 입력된 이메일 주소를 확인한 다음 사용자를 시스템에 저장합니다. 이메일 검증을 validateEmail 함수 내에 캡슐화하고 사용자를 saveUser 함수 내에 저장할 것입니다. validateEmail 함수가 먼저 실행되고, 이메일이 유효한 경우 saveUser 함수가 실행됩니다.

다음과 같이 실행 흐름이 진행됩니다.

1. Mutation.signUp 해석기 요청 매핑 템플릿
2. validateEmail 함수
3. saveUser 함수
4. Mutation.signUp 해석기 응답 매핑 템플릿

API의 다른 해석기 내에서 validateEmail 함수를 다시 사용할 것입니다. GraphQL 필드 간에 변경되기 때문에 `$ctx.args`에는 액세스하지 않도록 하겠습니다. 대신 `$ctx.stash`를 사용해 `signUp(input: Signup)` 입력 필드 인수의 이메일 속성을 저장할 수 있습니다.

사전 매핑 템플릿:

```
## store email input field into a generic email key
$util.qr($ctx.stash.put("email", $ctx.args.input.email))
{}
```

콘솔은 사용할 기본 패스스루 사후 매핑 템플릿을 제공합니다.

```
$util.toJson($ctx.result)
```

생성 또는 저장을 선택하여 해석기를 업데이트합니다.

함수 생성

파이프라인 해석기 페이지의 함수 섹션에서 함수 추가를 클릭하고 새 함수 생성을 클릭합니다. 해석기 페이지를 거치지 않고 함수를 생성할 수 있는데, 이렇게 하려면 AWS AppSync 콘솔에서 함수 페이지로 이동합니다. 함수 생성 버튼을 선택합니다. 이메일이 유효하고 특정 도메인에서 전송되는지 확인하는 함수를 생성합니다. 이메일이 유효하지 않으면 함수에서 오류가 발생합니다. 이메일이 유효할 경우 제공된 입력을 전달합니다.

새 함수 페이지에서 작업을 선택한 다음 런타임 업데이트를 선택합니다. VTL을 선택하고 업데이트를 선택합니다. 없음 유형의 데이터 원본을 생성했는지 확인합니다. 데이터 원본 이름 목록에서 이 데이터 원본을 선택합니다. 함수 이름에 `validateEmail`을 입력합니다. 함수 코드 영역에서 다음 스니펫으로 모든 내용을 덮어씁니다.

```
#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}) }
}
```

이 내용을 응답 매핑 템플릿에 붙여넣습니다.

```
$util.toJson($ctx.result)
```

선택 항목을 검토한 다음 생성을 선택합니다. 방금 validateEmail 함수를 생성했습니다. 이 단계를 반복하여 다음 요청 및 응답 매핑 템플릿으로 saveUser 함수를 생성합니다(단순화를 위해 없음 데이터 원본을 사용하고 함수가 실행된 후 사용자가 시스템에 저장된 것으로 가장함).

요청 매핑 템플릿

```
## $ctx.prev.result contains the signup input values. We could have also
## used $ctx.args.input.
{
    "payload": $util.toJson($ctx.prev.result)
}
```

응답 매핑 템플릿:

```
## 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)
```

방금 saveUser 함수를 생성했습니다.

파이프라인 해석기에 함수 추가

파이프라인 해석기에 방금 생성한 함수가 자동으로 추가되어야 합니다. 그렇지 않거나 함수 페이지를 통해 함수를 생성한 경우, 해석기 페이지에서 함수 추가를 클릭하여 연결할 수 있습니다. 해석기에 validateEmail 및 saveUser 함수를 모두 추가합니다. validateEmail 함수는 saveUser 함수 앞에 배치해야 합니다. 함수를 더 추가할 때 위로 이동 및 아래로 이동 옵션을 사용해 함수 실행 순서를 다시 정리할 수 있습니다. 변경 사항을 검토한 후 저장을 선택합니다.

쿼리 실행

AWS AppSync 콘솔에서 쿼리 페이지로 이동합니다. 탐색기에서 변형을 사용하고 있는지 확인합니다. 그렇지 않다면 드롭다운 목록에서 `Mutation`을 선택한 다음 +를 선택합니다. 다음 쿼리를 입력합니다.

```
mutation {
  signUp(input: {
    email: "nadia@myvaliddomain.com"
```

```
    username: "nadia"
  }) {
    id
    email
  }
}
```

다음과 유사하게 반환되어야 합니다.

```
{
  "data": {
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "email": "nadia@myvaliddomain.com"
    }
  }
}
```

파이프라인 해석기를 사용하여 성공적으로 사용자를 등록했고, 입력한 이메일을 검증했습니다. 파이프라인 해석기에 대해 집중적으로 설명하는 자습서 전체를 살펴보려면 자습서: 파이프라인 해석기를 참조하십시오.

4단계: API 사용: CDK 예제

> **Tip**
>
> CDK를 사용하기 전에 CDK의 공식 설명서와 함께 AWS AppSync의 CDK 참조를 검토하는 것이 좋습니다.
> 또한 AWS CLI 및 NPM 설치가 시스템에서 작동하는지 확인하는 것이 좋습니다.

이 섹션에서는 DynamoDB 테이블에서 항목을 추가하고 가져올 수 있는 간단한 CDK 앱을 만들 것입니다. 이 예제는 스키마 디자인, 데이터 소스 연결, 해석기 구성(JavaScript) 섹션의 일부 코드를 사용하는 빠른 시작 예제입니다.

CDK 프로젝트 만들기

> **Warning**
>
> 환경에 따라 이러한 단계가 완전히 정확하지 않을 수 있습니다. 시스템에 필요한 유틸리티가 설치되어 있고, AWS 서비스와 상호 작용하는 방법이 있으며, 적절한 구성이 마련되어 있다고 가정합니다.

첫 번째 단계는 AWS CDK를 설치하는 것입니다. CLI에 다음 명령을 입력할 수 있습니다.

```
npm install -g aws-cdk
```

다음으로 프로젝트 디렉터리를 만든 다음 해당 디렉터리로 이동해야 합니다. 디렉터리를 만들고 탐색하기 위한 명령 세트의 예는 다음과 같습니다.

```
mkdir example-cdk-app
cd example-cdk-app
```

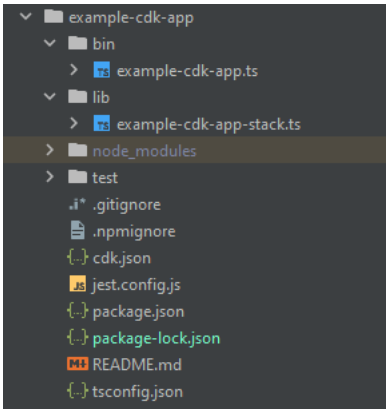
다음으로 앱을 만들어야 합니다. Amazon 서비스는 주로 TypeScript를 사용합니다. 프로젝트 디렉터리에서 다음 명령을 입력합니다.

```
cdk init app --language typescript
```

이렇게 하면 초기화 파일과 함께 CDK 앱이 설치됩니다.

```
Initializing a new git repository...
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Executing npm install...
✅All done!
```

프로젝트 구조는 다음과 같습니다.



다음과 같은 몇 가지 중요한 디렉토리가 있다는 것을 알 수 있습니다.

- `bin`: 초기 bin 파일이 앱을 생성합니다. 이 안내서에서는 이 내용을 다루지 않겠습니다.
- `lib`: lib 디렉토리에는 스택 파일이 들어 있습니다. 스택 파일은 개별 실행 단위로 생각할 수 있습니다. 구조는 스택 파일 안에 있을 것입니다. 기본적으로 앱이 배포될 때 AWS CloudFormation에서 스핀업되는 서비스를 위한 리소스입니다. 대부분의 코딩이 이루어지는 곳입니다.
- `node_modules`: 이 디렉터리는 NPM에서 생성하며 npm 명령을 사용하여 설치한 모든 패키지 종속성을 포함합니다.

초기 스택 파일에는 다음과 같은 내용이 포함될 수 있습니다.

```
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)
    // });
  }
}
```

앱에서 스택을 생성하기 위한 보일러플레이트 코드입니다. 이 예제의 코드 대부분은 이 클래스의 범위에 속합니다.

스택 파일이 앱의 디렉터리에 있는지 확인하려면 터미널에서 다음 명령어를 실행합니다.

```
cdk ls
```

스택 목록이 표시되어야 합니다. 그렇지 않으면 단계를 다시 실행하거나 공식 설명서에서 도움을 받아야 할 수 있습니다.

배포하기 전에 코드 변경사항을 빌드하려면 언제든지 터미널에서 다음 명령을 실행하면 됩니다.

```
npm run build
```

그리고 배포하기 전에 변경 사항을 확인하는 방법은 다음과 같습니다.

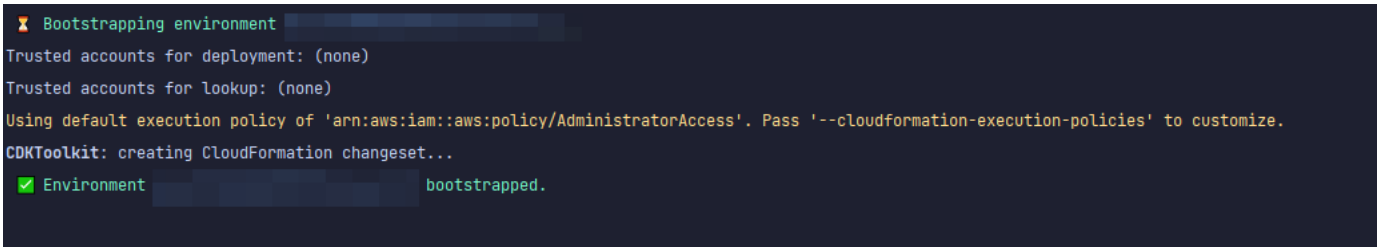
```
cdk diff
```

스택 파일에 코드를 추가하기 전에 부트스트랩을 수행합니다. 부트스트래핑을 통해 앱을 배포하기 전에 CDK용 리소스를 프로비저닝할 수 있습니다. 이 프로세스에 대한 자세한 내용은 여기에서 확인할 수 있습니다. 부트스트랩을 생성하기 위한 명령은 다음과 같습니다.

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

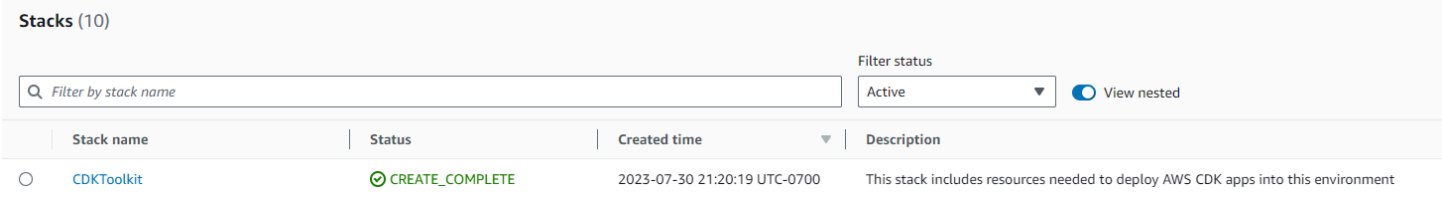
> **Tip**
>
> 이 단계를 수행하려면 계정에 여러 IAM 권한이 필요합니다. 권한이 없으면 부트스트랩이 거부됩니다. 이 경우 부트스트랩이 생성하는 S3 버킷과 같이 부트스트랩으로 인해 발생한 불완전한 리소스를 삭제해야 할 수도 있습니다.

부트스트랩은 여러 리소스를 가동합니다. 최종 메시지는 다음과 같습니다.

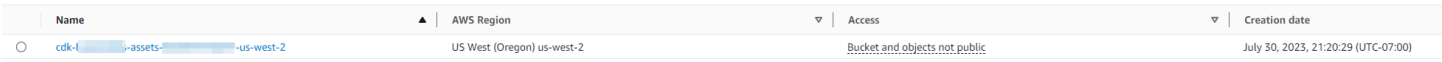


이 작업은 리전별로 계정당 한 번 수행되므로 자주 수행하지 않아도 됩니다. 부트스트랩의 기본 리소스는 AWS CloudFormation 스택과 Amazon S3 버킷입니다.

Amazon S3 버킷은 배포를 수행하는 데 필요한 권한을 부여하는 파일 및 IAM 역할을 저장하는 데 사용됩니다. 필요한 리소스는 부트스트랩 스택이라고 하는 AWS CloudFormation 스택에 정의되어 있으며, 일반적으로 이 스택의 이름은 `CDKToolkit`입니다. 다른 AWS CloudFormation 스택과 마찬가지로 배포가 완료되면 AWS CloudFormation 콘솔에 나타납니다.



버킷도 마찬가지입니다.



스택 파일에서 필요한 서비스를 가져오려면 다음 명령을 사용할 수 있습니다.

```
npm install aws-cdk-lib # V2 command
```

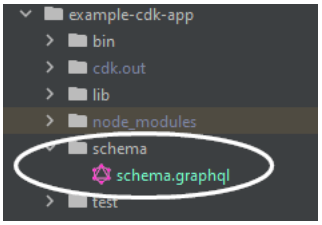
> **Tip**
>
> V2에 문제가 있는 경우 V1 명령을 사용하여 개별 라이브러리를 설치할 수 있습니다.
>
> ```
> npm install @aws-cdk/aws-appsync @aws-cdk/aws-dynamodb
> ```
>
> V1은 더 이상 사용되지 않으므로 권장하지 않습니다.

CDK 프로젝트 구현 - 스키마

이제 코드 구현을 시작할 수 있습니다. 먼저 스키마를 만들어야 합니다. 앱에서 간단히 `.graphql` 파일을 만들 수 있습니다.

```
mkdir schema
touch schema.graphql
```

이 예제에서는 `schema`라는 최상위 디렉터리에 `schema.graphql`이 포함되어 있습니다.



스키마 안에 간단한 예제를 포함시켜 보겠습니다.

```
input CreatePostInput {
    title: String
    content: String
}

type Post {
    id: ID!
    title: String
    content: String
}

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

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

스택 파일로 돌아가서 다음과 같은 가져오기 지시문이 정의되어 있는지 확인해야 합니다.

```
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';
```

클래스 내에서 GraphQL API를 만드는 코드를 추가하고 `schema.graphql` 파일에 연결합니다.

```
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'),
    });
  }
}
```

또한 GraphQL URL, API 키, 지역을 출력하는 코드를 추가할 예정입니다.

```
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
    });
  }
}
```

이제 앱 배포를 다시 사용하겠습니다.

```
cdk deploy
```

결과는 다음과 같습니다.

```
ExampleCdkAppStack: deploying... [1/1]
ExampleCdkAppStack: creating CloudFormation changeset...

 ✅ ExampleCdkAppStack

✨ Deployment time: 16.13s

Outputs:
ExampleCdkAppStack.GraphQLAPIKey = 
ExampleCdkAppStack.GraphQLAPIURL = https://                                    /graphql
ExampleCdkAppStack.StackRegion = us-west-2
Stack ARN:
arn:aws:cloudformation:

✨ Total time: 22s
```

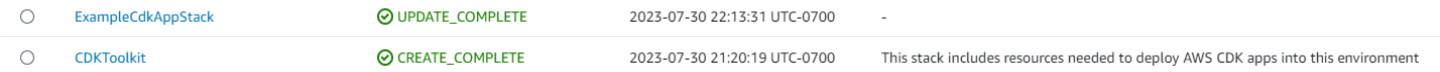
예제는 성공한 것 같지만 AWS AppSync 콘솔을 확인해 보겠습니다.

API가 생성된 것 같습니다. 이제 API에 연결된 스키마를 확인해 보겠습니다.

Schema

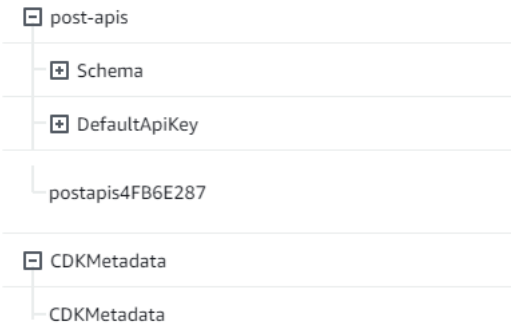
```
1  input CreatePostInput {
2    title: String
3    date: AWSDateTime
4  }
5
6  type Post {
7    id: ID!
8    title: String
9    date: AWSDateTime
10 }
11
12 type Mutation {
13   createPost(input: CreatePostInput!): Post
14 }
15
16 type Query {
17   getPost: [Post]
18 }
```

이 코드가 스키마 코드와 일치하는 것 같으니 성공입니다. 메타데이터 관점에서 이를 확인하는 또 다른 방법은 AWS CloudFormation 스택을 살펴보는 것입니다.



CDK 앱을 배포할 때 부트스트랩과 같은 리소스를 가동하기 위해 AWS CloudFormation을 거칩니다. 앱 내의 각 스택은 AWS CloudFormation 스택과 1:1로 매핑됩니다. 스택 코드로 돌아가면 클래스 이름

ExampleCdkAppStack에서 스택 이름을 가져왔습니다. GraphQL API 구조의 명명 규칙과 일치하는 리소스가 생성된 것을 확인할 수 있습니다.



CDK 프로젝트 구현 - 데이터 소스

다음으로 데이터 소스를 추가해야 합니다. 이 예제에서는 DynamoDB 테이블을 사용합니다. 스택 클래스 내에 새 테이블을 생성하는 코드를 몇 가지 추가해 보겠습니다.

```
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 || ''
```

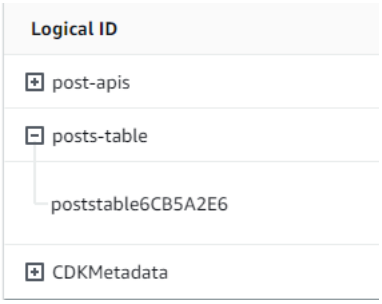
```
    });

    // Prints out the stack region to the terminal
    new cdk.CfnOutput(this, "Stack Region", {
      value: this.region
    });
  }
}
```

이제 다시 배포해 보겠습니다.

```
cdk deploy
```

DynamoDB 콘솔에서 새 테이블을 확인해야 합니다.

스택 이름이 올바르고 테이블 이름이 코드와 일치합니다. AWS CloudFormation 스택을 다시 확인하면 이제 새 테이블이 표시됩니다.



CDK 프로젝트 구현 - 해석기

이 예제에서는 두 개의 해석기를 사용합니다. 하나는 테이블을 쿼리하고 다른 하나는 테이블에 추가합니다. 파이프라인 해석기를 사용하고 있으므로 각각 함수가 하나씩 포함된 파이프라인 해석기 두 개를 선언해야 합니다. 쿼리에 다음 코드를 추가할 것입니다.

```
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
    });
```

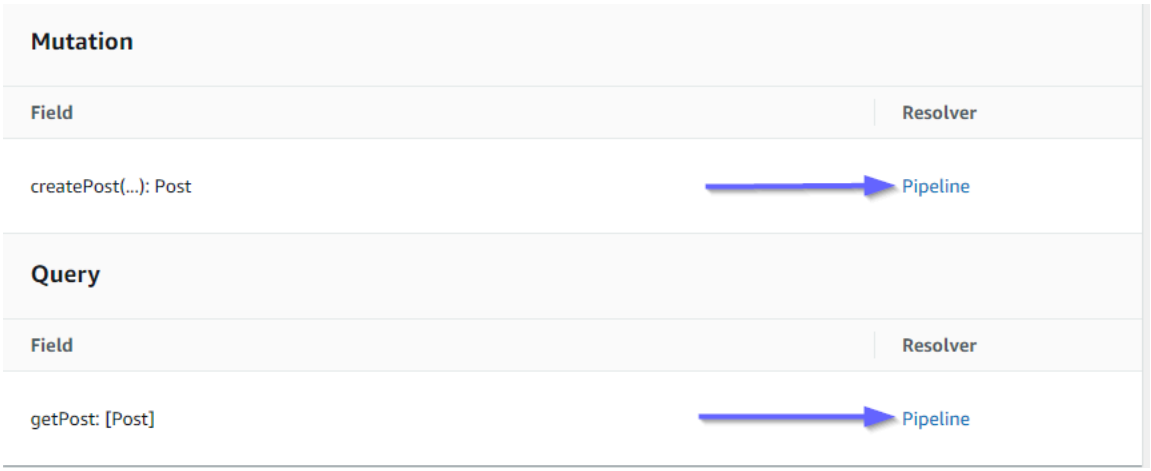
```
    // Prints out the AppSync GraphQL API key to the terminal
    new cdk.CfnOutput(this, "GraphQLAPIKey", {
      value: api.apiKey || ''
    });

    // Prints out the stack region to the terminal
    new cdk.CfnOutput(this, "Stack Region", {
      value: this.region
    });
  }
}
```

이 코드 조각에는 `func-add-post`라는 함수가 첨부되어 호출되는 `pipeline-resolver-create-posts` 파이프라인 리졸버를 추가했습니다. 테이블에 `Posts`를 추가하는 코드입니다. 다른 파이프라인 해석기는 테이블에 추가된 `Posts`를 검색하는 `func-get-post`라는 함수가 있는 `pipeline-resolver-get-posts`라고 불렀습니다.

이를 배포하여 AWS AppSync 서비스에 추가하겠습니다.

```
cdk deploy
```

AWS AppSync 콘솔에서 GraphQL API에 연결되었는지 확인해 보겠습니다.



맞는 것 같습니다. 코드에서 이 두 해석기는 모두 우리가 만든 GraphQL API에 연결되었습니다(해석기와 함수 모두에 있는 `api` 속성 값으로 표시됨). GraphQL API에서 해석기를 연결한 필드는 속성에도 지정되어 있습니다(각 해석기의 `typename` 및 `fieldname` 속성으로 정의됨).

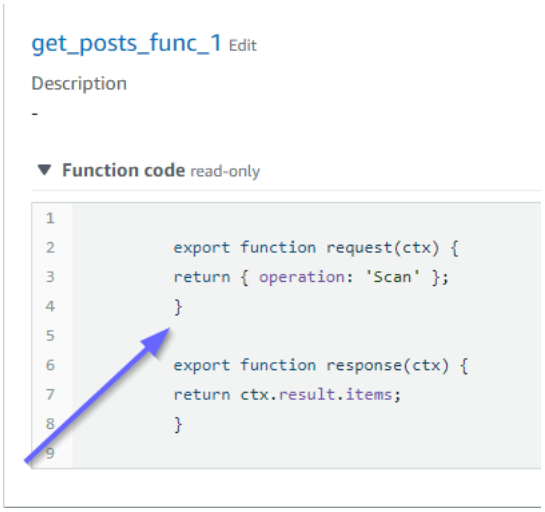
`pipeline-resolver-get-posts`부터 해석기의 콘텐츠가 올바른지 확인해 보겠습니다.

이전 핸들러와 이후 핸들러가 code 속성 값과 일치합니다. 또한 해석기에 연결된 함수 이름과 일치하는 add_posts_func_1이라는 함수가 있음을 알 수 있습니다.

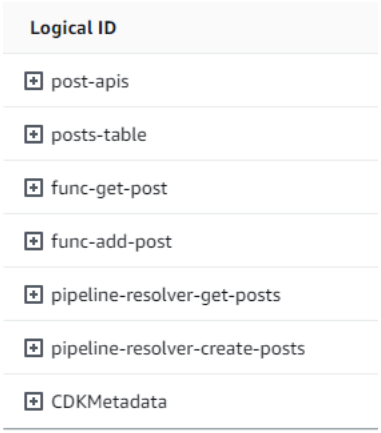
해당 함수의 코드 내용을 살펴보겠습니다.

이는 add_posts_func_1 함수의 code 속성과 일치합니다. 쿼리가 성공적으로 업로드되었으니 쿼리를 확인해 보겠습니다.

이것도 코드와 일치합니다. `get_posts_func_1`을 살펴보자면 다음과 같습니다.



모든 것이 제자리에 있는 것 같네요. 메타데이터 관점에서 이를 확인하기 위해 AWS CloudFormation에서 스택을 다시 확인할 수 있습니다.



이제 몇 가지 요청을 수행하여 이 코드를 테스트해야 합니다.

CDK 프로젝트 만들기 - 요청

AWS AppSync 콘솔에서 앱을 테스트하기 위해 쿼리 하나와 뮤테이션 하나를 만들었습니다.

```
query MyQuery {
  getPost {
    id
    date
    title
  }
}

mutation MyMutation {
  createPost(input: {date: "1970-01-01T12:30:00.000Z", title: "first post"}) {
    date
    id
    title
  }
}

```

MyMutation에는 1970-01-01T12:30:00.000Z 및 first post 인수가 있는 createPost 작업이 포함됩니다. 자동으로 생성된 id 값뿐만 아니라 우리가 전달한 date 및 title을 반환합니다. 뮤테이션을 실행하면 다음과 같은 결과가 나옵니다.

```
{
  "data": {
    "createPost": {
      "date": "1970-01-01T12:30:00.000Z",
      "id": "4dc1c2dd-0aa3-4055-9eca-7c140062ada2",
      "title": "first post"
    }
  }
}
```

DynamoDB 테이블을 빠르게 확인하면 테이블을 스캔할 때 테이블에 있는 항목을 확인할 수 있습니다.

id *(String)*	date	title
9f62c4dd-49d5-48d5-b835-143284c72fe0	1970-01-01T12:30:00.000Z	first post

AWS AppSync 콘솔로 돌아가서 쿼리를 실행하여 이 Post를 검색하면 다음과 같은 결과가 나타납니다.

```
{
  "data": {
    "getPost": [
      {
        "id": "9f62c4dd-49d5-48d5-b835-143284c72fe0",
        "date": "1970-01-01T12:30:00.000Z",
        "title": "first post"
```

```
      }
    ]
  }
}
```

실시간 데이터

AWS AppSync를 사용하면 구독을 활용하여 라이브 애플리케이션 업데이트, 푸시 알림 등을 구현할 수 있습니다. 클라이언트가 GraphQL 구독 작업을 간접적으로 호출하면 AWS AppSync에서 보안 WebSocket 연결을 자동으로 설정하고 유지 관리합니다. 그러면 AWS AppSync가 애플리케이션의 연결 및 조정 요구 사항을 지속적으로 관리하는 동안 애플리케이션은 데이터 원본의 데이터를 구독자에게 실시간으로 배포할 수 있습니다. 다음 섹션에서는 AWS AppSync의 구독이 작동하는 방식을 설명합니다.

GraphQL 스키마 구독 명령

AWS AppSync의 구독은 변형에 대한 응답으로 간접적으로 호출됩니다. 즉, 변형에 대한 GraphQL 스키마 명령을 지정하여 AWS AppSync에서 실시간으로 데이터 원본을 만들 수 있습니다.

AWS Amplify 클라이언트 라이브러리는 구독 연결 관리를 자동으로 처리합니다. 라이브러리는 순수 WebSocket을 클라이언트와 서비스 간의 네트워크 프로토콜로 사용합니다.

> **Note**
>
> 구독을 위해 연결 시 권한 부여를 제어하려면 필드 수준 권한 부여에 AWS Identity and Access Management(IAM), AWS Lambda, Amazon Cognito 자격 증명 풀 또는 Amazon Cognito 사용자 풀을 사용할 수 있습니다. 구독에 대한 세부적인 액세스 제어를 위해 해석기를 구독 필드에 연결하고 호출자의 자격 증명과 AWS AppSync 데이터 원본을 사용하여 로직을 수행할 수 있습니다. 자세한 내용은 인증 및 권한 부여 섹션을 참조하세요.

구독이 변형에서 트리거되고 변형 선택 세트가 구독자에게 전송됩니다.

다음 예에서는 GraphQL 구독을 사용하는 방법을 보여 줍니다. 데이터 원본이 Lambda, Amazon DynamoDB 또는 Amazon OpenSearch Service일 수 있으므로 데이터 원본을 지정하지 않습니다.

구독으로 시작하려면 다음과 같이 스키마에 구독 진입점을 추가해야 합니다.

```
schema {
```

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

블로그 게시물 사이트가 있고 새 블로그와 기존 블로그에 대한 변경 사항을 구독하려 한다고 가정해 보겠습니다. 이를 수행하려면 스키마에 다음 Subscription 정의를 추가합니다.

```
type Subscription {
    addedPost: Post
    updatedPost: Post
    deletedPost: Post
}
```

다음과 같은 변형이 있다고 추가로 가정해 보겠습니다.

```
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!
}
```

다음과 같이 알림을 받고자 하는 각 구독에 대해 `@aws_subscribe(mutations: ["mutation_field_1", "mutation_field_2"])` 명령을 추가하여 이러한 필드를 실시간으로 만들 수 있습니다.

```
type Subscription {
    addedPost: Post
    @aws_subscribe(mutations: ["addPost"])
    updatedPost: Post
    @aws_subscribe(mutations: ["updatePost"])
    deletedPost: Post
    @aws_subscribe(mutations: ["deletePost"])
}
```

`@aws_subscribe(mutations: ["",..,""])`는 변형 입력의 배열을 받기 때문에 구독을 시작하는 여러 변형을 지정할 수 있습니다. 클라이언트로부터 구독하는 경우, GraphQL 쿼리는 다음과 같습니다.

```
subscription NewPostSub {
```

```
    addedPost {
        __typename
        version
        title
        content
        author
        url
    }
}
```

이 구독 쿼리는 클라이언트 연결 및 도구에 필요합니다.

순수 WebSocket 클라이언트를 사용하는 경우 각 클라이언트가 자체 선택 세트를 정의할 수 있으므로 선택 세트 필터링은 클라이언트별로 수행됩니다. 이 경우 구독 선택 세트는 변형 선택 세트의 하위 집합이어야 합니다. 예를 들어 `addPost(...){id author title url version}` 변형과 연결된 `addedPost{author title}` 구독은 게시물의 작성자와 제목만 수신합니다. 다른 필드는 수신하지 않습니다. 그러나 변형의 선택 집합에 작성자가 없으면 구독자는 작성자 필드에 대한 `null` 값을 가져옵니다(작성자 필드가 스키마에서 필수/null이 아님으로 정의된 경우에는 오류를 가져옴).

순수 WebSocket을 사용할 때는 구독 선택 세트가 필수입니다. 구독에서 필드가 명시적으로 정의되지 않은 경우 AWS AppSync는 필드를 반환하지 않습니다.

이전 예제에서 구독에는 인수가 없었습니다. 스키마가 다음과 같은 모습이라고 가정합니다.

```
type Subscription {
    updatedPost(id:ID! author:String): Post
    @aws_subscribe(mutations: ["updatePost"])
}
```

이 경우 클라이언트는 구독을 다음과 같이 정의합니다.

```
subscription UpdatedPostSub {
    updatedPost(id:"XYZ", author:"ABC") {
        title
        content
    }
}
```

스키마에서 `subscription` 필드의 반환 형식은 해당 변형 필드의 반환 형식과 동일해야 합니다. 이전 예에서는 이러한 내용이 `addPost`와 `addedPost` 형식으로 반환된 `Post` 둘 다로 표시되었습니다.

클라이언트에 대한 구독을 설정하려면 클라이언트 애플리케이션 빌드 섹션을 참조하세요.

구독 인수 사용

GraphQL 구독 사용 시 중요한 부분은 인수를 사용하는 시기와 방법을 이해하는 것입니다. 발생한 변형에 대해 클라이언트에게 알리는 방법과 시기를 세밀하게 수정할 수 있습니다. 이렇게 하려면 'Todos'를 생성하는 빠른 시작 장의 샘플 스키마를 참조하세요. 이 샘플 스키마에는 다음 변형이 정의되어 있습니다.

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

기본 샘플에서 클라이언트는 인수 없이 onUpdateTodo subscription을 사용하여 Todo에 대한 업데이트를 구독할 수 있습니다.

```
subscription OnUpdateTodo {
  onUpdateTodo {
    description
    id
    name
    when
  }
}
```

사용자는 해당 인수를 사용하여 subscription을 필터링할 수 있습니다. 예를 들어 특정 ID를 사용하는 todo가 업데이트될 때만 subscription을 트리거하려면 ID 값을 지정합니다.

```
subscription OnUpdateTodo {
  onUpdateTodo(id: "a-todo-id") {
    description
    id
    name
    when
  }
}
```

또한 여러 인수를 전달할 수도 있습니다. 예를 들어, 다음 subscription은 특정 장소와 시간에 Todo 업데이트에 대한 알림을 받는 방법을 보여 줍니다.

```
subscription todosAtHome {
  onUpdateTodo(when: "tomorrow", where: "at home") {
    description
    id
    name
    when
    where
  }
}
```

모든 인수는 선택 사항입니다. subscription에 인수를 지정하지 않으면 애플리케이션에서 발생하는 모든 Todo 업데이트를 구독하게 됩니다. 하지만 ID 인수가 필요하도록 subscription의 필드 정의를 업데이트할 수 있습니다. 이렇게 하면 모든 todo의 응답이 아닌 특정 todo의 응답이 발생합니다.

```
onUpdateTodo(
  id: ID!,
  name: String,
  when: String,
  where: String,
  description: String
): Todo
```

인수 null 값에는 의미가 있습니다.

AWS AppSync에서 구독 쿼리를 수행하면 null 인수 값은 인수를 전체적으로 생략하는 것과는 다르게 결과를 필터링합니다.

todos를 생성할 수 있는 todos API 샘플로 돌아가 보겠습니다. 빠른 시작 장의 샘플 스키마를 참조하세요.

Todo 유형에 소유자를 설명하는 새 owner 필드를 포함하도록 스키마를 수정해 보겠습니다. owner 필드는 필수가 아니며 UpdateTodoInput에만 설정할 수 있습니다. 다음과 같은 스키마의 단순화된 버전을 참조하세요.

```
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"])
}
```

다음 구독은 모든 Todo 업데이트를 반환합니다.

```
subscription MySubscription {
  onUpdateTodo {
    description
    id
    name
    when
    where
  }
}
```

필드 인수 owner: null을 추가하도록 이전 구독을 수정하는 경우 이제 다른 질문을 하게 됩니다. 이제 이 구독은 소유자를 제공하지 않은 모든 Todo 업데이트에 대한 알림을 받을 수 있도록 클라이언트를 등록합니다.

```
subscription MySubscription {
  onUpdateTodo(owner: null) {
    description
    id
    name
    when
    where
  }
}
```

> **Note**
>
> 2022년 1월 1일부터는 WebSocket을 통한 MQTT를 더 이상 AWS AppSync API의 GraphQL 구독에 대한 프로토콜로 사용할 수 없습니다. 순수 WebSocket은 AWS AppSync에서 지원되는 유일한 프로토콜입니다.
> 2019년 11월 이후에 출시된 AWS AppSync SDK 또는 Amplify 라이브러리 기반 클라이언트는 기본적으로 순수 WebSocket을 자동으로 사용합니다. 클라이언트를 최신 버전으로 업그레이드하면 AWS AppSync의 순수 WebSocket 엔진을 사용할 수 있습니다.
> 순수 WebSocket은 더 큰 페이로드 크기(240KB), 더 다양한 클라이언트 옵션, 개선된 CloudWatch 메트릭을 제공합니다. 순수 WebSocket 사용 방법에 관한 추가 정보는 실시간 웹소켓 클라이언트 빌드 섹션을 참조하세요.

서버리스 WebSocket으로 구동되는 일반 pub-sub API 생성

일부 애플리케이션에는 클라이언트가 특정 채널 또는 주제를 청취하는 단순한 WebSocket API만 필요합니다. 특정 형태나 엄격한 형식의 요구 사항이 없는 일반 JSON 데이터는 순수하고 단순한 게시-구독(pub/sub) 패턴으로 이러한 채널 중 하나를 청취하는 클라이언트에게 푸시할 수 있습니다.

AWS AppSync를 사용하면 GraphQL에 대한 지식이 거의 또는 전혀 없어도 API 백엔드와 클라이언트 측 모두에서 GraphQL 코드를 자동으로 생성하여 단순한 pub-sub WebSocket API를 간단하게 구현할 수 있습니다.

pub-sub API 생성 및 구성

시작하려면 다음을 따르세요.

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
   - 대시보드에서 API 생성을 선택합니다.

2. 다음 화면에서 실시간 API 생성을 선택한 후 다음을 선택합니다.
3. pub/sub API에 대해 기억하기 쉬운 이름을 입력합니다.
4. 프라이빗 API 기능을 활성화할 수 있지만, 지금은 사용하지 않는 것이 좋습니다. 다음(Next)을 선택합니다.
5. WebSocket을 사용하여 작동하는 pub/sub API를 자동으로 생성하도록 선택할 수 있습니다. 마찬가지로 지금은 이 기능도 사용하지 않는 것이 좋습니다. 다음(Next)을 선택합니다.
6. API 생성을 선택한 다음 몇 분 정도 기다립니다. 사전 구성된 AWS AppSync pub/sub API가 AWS 계정에 새로 생성됩니다.

API는 AWS AppSync에 내장된 로컬 해석기(로컬 해석기 사용에 대한 자세한 내용은 AWS AppSync 개발자 안내서의 자습서: 로컬 해석기 참조)를 사용하여 여러 개의 임시 pub/sub 채널과 WebSocket 연결을 관리합니다. WebSocket 연결은 채널 이름만을 기준으로 구독한 클라이언트에 데이터를 자동으로 전달하고 필터링합니다. API 직접 호출은 API 키로 권한이 부여됩니다.

API가 배포되면 클라이언트 코드를 생성하고 이를 클라이언트 애플리케이션과 통합하기 위한 몇 가지 추가 단계가 제공됩니다. 이 안내서에서는 클라이언트를 빠르게 통합하는 방법의 예로 간단한 React 웹 애플리케이션을 사용합니다.

1. 먼저 로컬 컴퓨터에서 NPM을 사용하여 보일러플레이트 React 앱을 만듭니다.

```
$ npx create-react-app mypubsub-app
$ cd mypubsub-app
```

> **Note**
>
> 이 예제에서는 Amplify 라이브러리를 사용하여 클라이언트를 백엔드 API에 연결합니다. 하지만 Amplify CLI 프로젝트를 로컬에서 생성할 필요는 없습니다. 이 예제에서는 React를 클라이언트로 선택했지만 Amplify 라이브러리는 iOS, Android 및 Flutter 클라이언트도 지원하므로 서로 다른 런타임에서 동일한 기능을 제공합니다. 지원되는 Amplify 클라이언트는 AWS AppSync 실시간 WebSocket 프로토콜과 완벽하게 호환되는 내장형 WebSocket 기능을 포함하여 몇 줄의 코드만으로 AWS AppSync GraphQL API 백엔드와 상호 작용할 수 있는 간단한 추상화를 제공합니다.
>
> ```
> $ npm install @aws-amplify/api
> ```

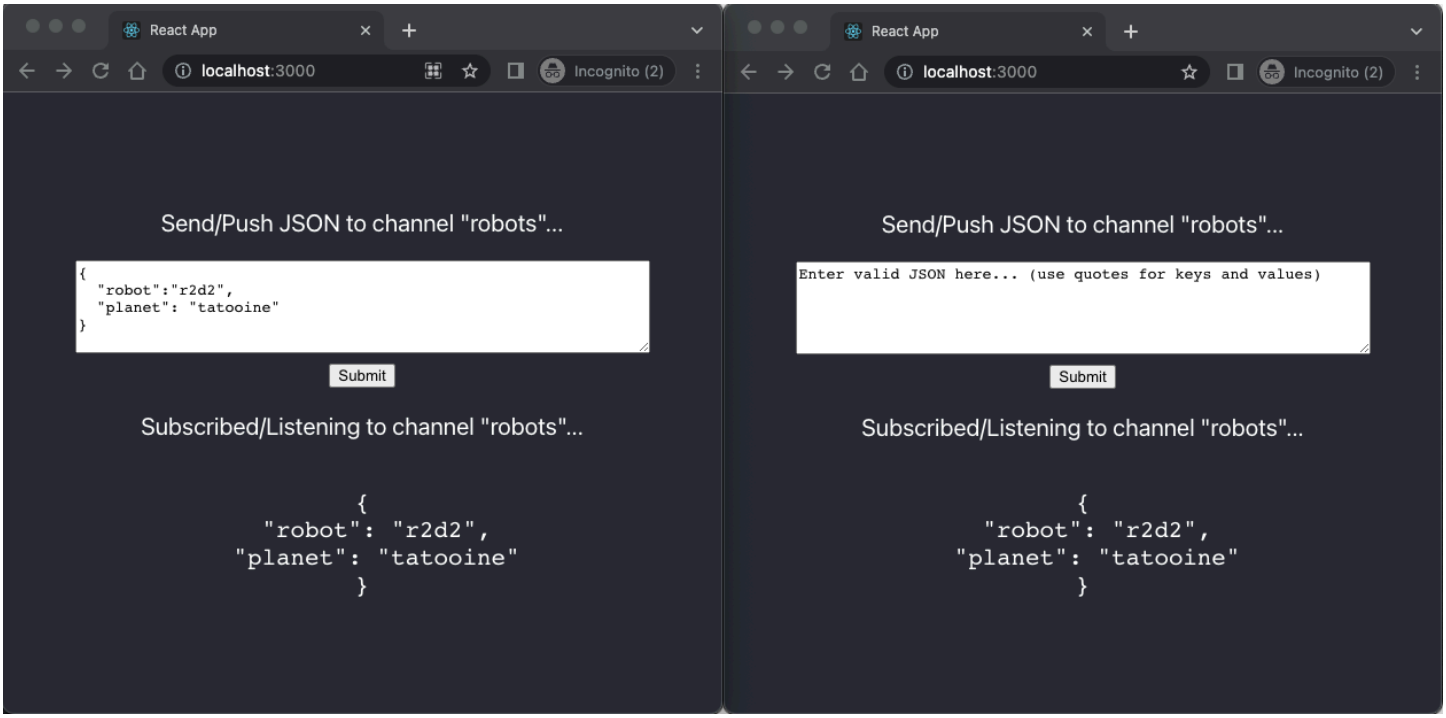
2. AWS AppSync 콘솔에서 JavaScript를 선택한 다음 다운로드를 선택하여 API 구성 세부 정보와 생성된 GraphQL 작업 코드가 포함된 단일 파일을 다운로드합니다.
3. 다운로드한 파일을 React 프로젝트의 `/src` 폴더에 복사합니다.
4. 다음으로, 기존 보일러플레이트 `src/App.js` 파일의 내용을 콘솔에서 사용할 수 있는 샘플 클라이언트 코드로 대체합니다.
5. 다음 명령을 사용하여 로컬에서 애플리케이션을 시작합니다.

```
$ npm start
```

6. 실시간 데이터 전송 및 수신을 테스트하려면 두 개의 브라우저 창을 열고 *`localhost:3000`*에 액세스하세요. 샘플 애플리케이션은 일반 JSON 데이터를 *`robots`*라는 하드 코딩된 채널로 전송하도록 구성되어 있습니다.
7. 브라우저 창 중 하나에서 텍스트 상자에 다음 JSON blob을 입력한 다음 제출을 클릭합니다.

```
{
  "robot":"r2d2",
  "planet": "tatooine"
}
```

두 브라우저 인스턴스 모두 *`robots`* 채널을 구독하며 게시된 데이터를 실시간으로 수신하여 웹 애플리케이션 하단에 표시합니다.



스키마, 해석기, 작업 등 필요한 모든 GraphQL API 코드가 자동으로 생성되어 일반 pub/sub 사용 사례를 지원합니다. 백엔드에서 데이터는 다음과 같은 GraphQL 변형을 사용하여 AWS AppSync의 실시간 엔드포인트에 게시됩니다.

```
mutation PublishData {
    publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
        data
        name
    }
}
```

구독자는 관련 GraphQL 구독을 통해 특정 임시 채널로 전송되는 게시된 데이터에 액세스합니다.

```
subscription SubscribeToData {
    subscribe(name:"channel") {
        name
        data
    }
}
```

기존 애플리케이션에 pub-sub API 구현

기존 애플리케이션에 실시간 기능을 구현해야 하는 경우 이 일반 pub/sub API 구성을 모든 애플리케이션 또는 API 기술에 쉽게 통합할 수 있습니다. 단일 API 엔드포인트를 사용하여 GraphQL을 통한 단일 네트워크 호출로 하나 이상의 데이터 원본에서 데이터에 안전하게 액세스하고 조작 및 결합할 때의 이점이 있지만, AWS AppSync의 실시간 기능을 활용하기 위해 기존 REST 기반 애플리케이션으로 변환하거나 처음부터 다시 빌드할 필요는 없습니다. 예를 들어, 실시간 및 pub/sub 목적으로만 기존 애플리케이션에서 일반 pub/sub API로 메시지나 이벤트를 보내고 받는 클라이언트가 있는 기존 CRUD 워크로드를 별도의 API 엔드포인트에 둘 수 있습니다.

향상된 구독 필터링

AWS AppSync에서는 추가 논리 연산자를 지원하는 필터를 사용하여 GraphQL API 구독 해석기에서 직접 백엔드의 데이터 필터링을 위한 비즈니스 로직을 정의하고 활성화할 수 있습니다. 클라이언트의 구독 쿼리에 정의된 구독 인수와는 다르게, 이러한 필터를 구성할 수 있습니다. 구독 인수 사용에 관한 자세한 정보는 구독 인수 사용 섹션을 참조하세요. 연산자 목록은 해석기 매핑 템플릿 유틸리티 참조 섹션을 참조하세요.

이 문서에서는 실시간 데이터 필터링을 다음과 같은 범주로 분류합니다.

- 기본 필터링 - 구독 쿼리의 클라이언트 정의 인수를 기반으로 필터링합니다.
- 향상된 필터링 - AWS AppSync 서비스 백엔드에서 중앙에 정의된 로직을 기반으로 필터링합니다.

다음 섹션에서는 향상된 구독 필터를 구성하는 방법을 설명하고 실제 사용 사례를 보여 줍니다.

GraphQL 스키마에서 구독 정의

향상된 구독 필터를 사용하려면 GraphQL 스키마에서 구독을 정의한 다음 필터링 확장을 사용하여 향상된 필터를 정의합니다. AWS AppSync에서 향상된 구독 필터링이 작동하는 방식을 설명하기 위해 티켓 관리 시스템 API를 정의하는 다음 GraphQL 스키마를 예로 사용합니다.

```
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
```

API용 NONE 데이터 원본을 만든 다음 이 데이터 원본을 사용하여 해석기를 createTicket 변형에 연결한다고 가정해 보겠습니다. 핸들러는 다음과 같을 수 있습니다.

```
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**
>
> 향상된 필터는 지정된 구독에서 GraphQL 해석기의 핸들러에 활성화됩니다. 자세한 내용은 해석기 참조를 참조하세요.

향상된 필터의 동작을 구현하려면 `extensions.setSubscriptionFilter()` 함수를 사용하여 구독하는 클라이언트가 관심을 가질 수 있는 GraphQL 변형의 게시된 데이터에 대해 평가된 필터 식을 정의해야 합니다. 필터링 확장에 대한 자세한 내용은 확장을 참조하세요.

다음 섹션에서는 필터링 확장을 사용하여 향상된 필터를 구현하는 방법을 설명합니다.

필터링 확장을 사용하여 향상된 구독 필터 생성

향상된 필터는 구독 해석기의 응답 핸들러에 JSON으로 작성됩니다. 필터는 `filterGroup`이라는 목록으로 그룹화할 수 있습니다. 필터는 각각 필드, 연산자 및 값이 있는 하나 이상의 규칙을 사용하여 정의됩니다. 향상된 필터를 설정하는 `onSpecialTicketCreated`에 대한 새 해석기를 정의해 보겠습니다. 하나의 필터는 AND 로직을 사용하여 평가되고, 필터 그룹의 여러 필터는 OR 로직을 사용하여 평가되는 여러 규칙을 구성할 수 있습니다.

```
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;
}
```

이전 예제에 정의된 필터를 기반으로 다음을 사용하여 티켓을 만들면 중요한 티켓이 구독하는 API 클라이언트에게 자동으로 푸시됩니다.

- `priority` 수준 `high` 또는 `medium`

  AND

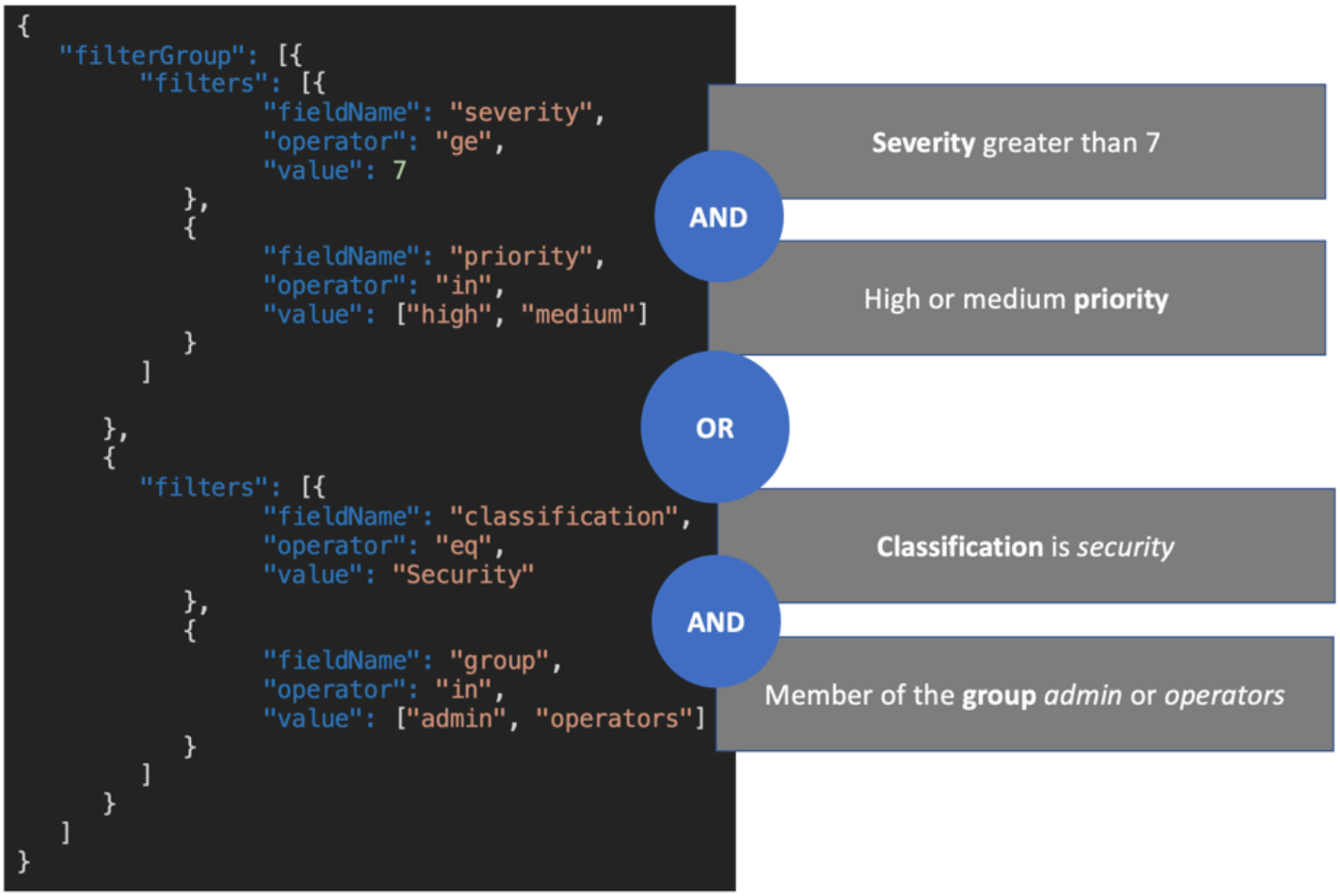
- `severity` 수준이 7보다 크거나 같음(ge)

또는

- `classification` 티켓이 `Security`로 설정됨

  AND

- `group` 할당이 `admin` 또는 `operators`로 설정됨



구독 해석기에 정의된 필터(향상된 필터링)는 구독 인수만을 기반으로 하는 필터링(기본 필터링)보다 우선합니다. 구독 인수 사용에 관한 자세한 정보는 구독 인수 사용을 참조하세요.

구독의 GraphQL 스키마에서 인수가 정의되고 필요한 경우 해당 인수가 해석기의 `extensions.setSubscriptionFilter()` 메서드에 규칙으로 정의된 경우에만 지정된 인수를 기반으로 필터링됩니다. 하지만 구독 해석기에 `extensions` 필터링 메서드가 없는 경우 클라이언트에 정의된 인수는 기본 필터링에만 사용됩니다. 기본 필터링과 향상된 필터링은 동시에 사용할 수 없습니다.

구독의 필터 확장 로직에 있는 context 변수를 사용하여 요청에 대한 컨텍스트 정보에 액세스할 수 있습니다. 예를 들어 권한 부여를 위해 Amazon Cognito 사용자 풀, OIDC 또는 Lambda 사용자 지정 권한 부여자를 사용하는 경우 구독이 설정되면 context.identity의 사용자에 대한 정보를 검색할 수 있습니다. 이 정보를 사용하여 사용자의 자격 증명을 기반으로 필터를 설정할 수 있습니다.

이제 onGroupTicketCreated에 대한 향상된 필터 동작을 구현한다고 가정해 보겠습니다. onGroupTicketCreated 구독에는 필수 group 이름이 인수로 필요합니다. 티켓을 만들면 티켓에 pending 상태가 자동으로 지정됩니다. 제공된 그룹에 속하는 새로 만들어진 티켓만 받도록 구독 필터를 설정할 수 있습니다.

```
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;
}
```

다음 예와 같이 변형을 사용하여 데이터를 게시하는 경우:

```
mutation CreateTicket {
  createTicket(input: {priority: medium, severity: 2, group: "aws"}) {
    id
    priority
    severity
    status
    group
    createdAt
  }
}
```

구독한 클라이언트는 createTicket 변형이 포함된 티켓이 생성되는 즉시 WebSocket을 통해 데이터가 자동으로 푸시되도록 수신 대기합니다.

```
subscription OnGroup {
```

```
  onGroupTicketCreated(group: "aws") {
    category
    status
    severity
    priority
    id
    group
    createdAt
    content
  }
}
```

필터링 로직이 향상된 필터링으로 AWS AppSync 서비스에 구현되어 클라이언트 코드를 간소화하기 때문에 클라이언트는 인수 없이 구독할 수 있습니다. 클라이언트는 정의된 필터 기준을 충족하는 경우에만 데이터를 수신합니다.

중첩된 스키마 필드에 대한 향상된 필터 정의

향상된 구독 필터링을 사용하여 중첩된 스키마 필드를 필터링할 수 있습니다. 위치 및 주소 유형을 포함하도록 이전 섹션의 스키마를 수정했다고 가정해 보겠습니다.

```
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
```

이 스키마에서는 . 구분 기호를 사용하여 중첩을 나타낼 수 있습니다. 다음 예제에서는 `location.address.country` 아래에 중첩된 스키마 필드에 대한 필터 규칙을 추가합니다. 티켓 주소가 USA로 설정되면 구독이 트리거됩니다.

```
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;
}
```

위의 예에서 `location`은 중첩 수준 1을, `address`는 중첩 수준 2, `country`는 중첩 수준 3을 나타내며, 모두 `.` 구분 기호로 구분됩니다.

`createTicket` 변형을 사용하여 이 구독을 테스트할 수 있습니다.

```
mutation CreateTicketInUSA {
  createTicket(input: {location: "{\"address\":{\"country\":\"USA\"}}"}) {
    category
    content
    createdAt
    group
    id
    location {
      address {
        country
      }
    }
    priority
    severity
    status
  }
}
```

클라이언트에서 향상된 필터 정의

GraphQL의 기본 필터링을 구독 인수와 함께 사용할 수 있습니다. 구독 쿼리에서 호출을 수행하는 클라이언트가 인수 값을 정의합니다. `extensions` 필터링 기능이 있는 AWS AppSync 구독 해석기에서 향상된 필터를 활성화하면 해석기에 정의된 백엔드 필터가 우선 순위를 갖습니다.

구독의 `filter` 인수를 사용하여 동적 클라이언트 정의 향상된 필터를 구성하세요. 이러한 필터를 구성할 때는 새 인수를 반영하도록 GraphQL 스키마를 업데이트해야 합니다.

```
...
type Subscription {
    onSpecialTicketCreated(filter: String): Ticket
```

```
        @aws_subscribe(mutations: ["createTicket"])
}
...
```

그러면 클라이언트는 다음 예와 같이 구독 쿼리를 전송할 수 있습니다.

```
subscription onSpecialTicketCreated($filter: String) {
     onSpecialTicketCreated(filter: $filter) {
        id
        group
        description
        priority
        severity
     }
 }
```

다음 예와 같이 쿼리 변수를 구성할 수 있습니다.

```
{"filter" : "{\"severity\":{\"le\":2}}"}
```

구독 응답 매핑 템플릿에서 `util.transform.toSubscriptionFilter()` 해석기 유틸리티를 구현하여 구독 인수에 정의된 필터를 각 클라이언트의 적용할 수 있습니다.

```
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;
}
```

이 전략을 통해 클라이언트는 향상된 필터링 로직과 추가 연산자를 사용하는 자체 필터를 정의할 수 있습니다. 지정된 클라이언트가 보안 WebSocket 연결에서 구독 쿼리를 간접적으로 호출하면 필터가 할당됩니다. `filter` 쿼리 변수 페이로드의 형식을 비롯하여 향상된 필터링을 위한 변환 유틸리티에 대한 자세한 내용은 JavaScript 해석기 개요를 참조하세요.

향상된 필터링 추가 제한 사항

다음은 향상된 필터에 추가 제한이 적용되는 몇 가지 사용 사례입니다.

- 향상된 필터는 최상위 객체 목록에 대한 필터링을 지원하지 않습니다. 이 사용 사례에서는 향상된 구독을 위해 변형의 게시된 데이터가 무시됩니다.
- AWS AppSync는 최대 다섯 가지 수준의 중첩을 지원합니다. 중첩 수준 5를 초과하는 스키마 필드에 대한 필터는 무시됩니다. 아래 GraphQL 응답을 예로 들어 보겠습니다. `venue.address.country.metadata.continent`의 `continent` 필드는 수준 5 중첩이므로 허용됩니다. 그러나 `venue.address.country.metadata.capital.financial`의 `financial`은 수준 6 중첩이므로 필터가 작동하지 않습니다.

```
{
    "data": {
        "onCreateFilterEvent": {
            "venue": {
                "address": {
                    "country": {
                        "metadata": {
                            "capital": {
                                "financial": "New York"
                            },
                            "continent" : "North America"
                        }
                    },
                    "state": "WA"
                },
                "builtYear": 2023
            },
            "private": false,
        }
    }
}
```

필터를 사용하여 WebSocket 연결 구독 취소

AWS AppSync에서는 특정 필터링 로직에 따라 연결된 클라이언트의 WebSocket 연결을 강제로 구독 취소하고 종료(무효화)할 수 있습니다. 이 기능은 그룹에서 사용자를 제거하는 경우와 같은 권한 부여 관련 시나리오에서 유용합니다.

구독 무효화는 변형에 정의된 페이로드에 대한 응답으로 발생합니다. 구독 연결을 무효화하는 데 사용되는 변형는 API의 관리 작업으로 취급하고 그에 따라 관리 사용자, 그룹 또는 백엔드 서비스로 사용을 제한하여 권한 범위를 지정하는 것이 좋습니다. 예를 들어 `@aws_auth(cognito_groups: ["Administrators"])` 또는 `@aws_iam`과 같은 스키마 권한 부여 지시문을 사용할 수 있습니다. 자세한 내용은 추가 권한 부여 모드 사용을 참조하세요.

무효화 필터는 향상된 구독 필터와 동일한 구문 및 로직을 사용합니다. 다음 유틸리티를 사용하여 이러한 필터를 정의하세요.

- `extensions.invalidateSubscriptions()` - 변형에 대한 GraphQL 해석기의 응답 핸들러에서 정의됩니다.
- `extensions.setSubscriptionInvalidationFilter()` - 변형에 연결된 구독의 GraphQL해석기의 응답 핸들러에서 정의됩니다.

무효화 필터링 확장에 대한 자세한 내용은 JavaScript 해석기 개요를 참조하세요.

구독 무효화 사용

AWS AppSync에서 구독 무효화가 작동하는 방식을 보려면 다음 GraphQL 스키마를 사용하세요.

```
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
}
```

removeUserFromGroup 변형 해석기 코드에서 무효화 필터를 정의하세요.

```
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 };
}
```

변형이 간접적으로 호출되면 subscriptionField에 정의된 구독을 취소하는 데 payload 객체에 정의된 데이터가 사용됩니다. 무효화 필터는 onGroupMessageCreated 구독의 응답 매핑 템플릿에도 정의되어 있습니다.

필터에 정의된 대로 구독한 클라이언트의 ID와 일치하는 ID가 extensions.invalidateSubscriptions() 페이로드에 포함된 경우 해당 구독은 구독 취소됩니다. 또한 WebSocket 연결도 종료됩니다. onGroupMessageCreated 구독의 구독 해석기 코드를 정의하세요.

```
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;
}
```

참고로 구독 응답 핸들러에는 구독 필터와 무효화 필터를 동시에 정의할 수 있습니다.

예를 들어 클라이언트 A가 다음 구독 요청을 사용하여 ID가 *group-1*인 그룹에 ID가 *user-1*인 새로운 사용자를 구독한다고 가정해 보겠습니다.

```
onGroupMessageCreated(userId : "user-1", groupId: :"group-1"){...}
```

AWS AppSync에서는 구독 해석기를 실행하여 이전 onGroupMessageCreated 응답 매핑 템플릿에 정의된 대로 구독 및 무효화 필터를 생성합니다. 클라이언트 A의 경우 구독 필터는 데이터가 *group-1*로만 전송될 수 있게 하며, 무효화 필터는 *user-1* 및 *group-1* 모두에 대해 정의됩니다.

이제 클라이언트 B가 다음 구독 요청을 사용하여 ID가 *group-2*인 그룹에 ID가 *user-2*인 사용자를 구독한다고 가정해 보겠습니다.

```
onGroupMessageCreated(userId : "user-2", groupId: :"group-2"){...}
```

AWS AppSync에서는 구독 해석기를 실행하여 구독 및 무효화 필터를 생성합니다. 클라이언트 B의 경우 구독 필터는 데이터가 *group-2*로만 전송될 수 있게 하며, 무효화 필터는 *user-2* 및 *group-2* 모두에 대해 정의됩니다.

다음으로, 다음 예와 같이 변형 요청을 사용하여 ID가 *message-1*인 새로운 그룹 메시지를 생성한다고 가정해 보겠습니다.

```
createGroupMessage(id: "message-1", groupId :
      "group-1", message: "test message"){...}
```

정의된 필터와 일치하는 구독 클라이언트는 WebSocket을 통해 다음 데이터 페이로드를 자동으로 수신합니다.

```
{
  "data": {
    "onGroupMessageCreated": {
      "id": "message-1",
      "groupId": "group-1",
      "message": "test message",
    }
  }
}
```

필터링 기준이 정의된 구독 필터와 일치하므로 클라이언트 A가 메시지를 수신합니다. 하지만 사용자가 *group-1*에 속하지 않으므로 클라이언트 B는 메시지를 수신하지 않습니다. 또한 요청이 구독 해석기에 정의된 구독 필터와 일치하지 않습니다.

마지막으로, 다음 변형 요청을 사용하여 *user-1*이 *group-1*에서 제거되었다고 가정해 보겠습니다.

```
removeUserFromGroup(userId: "user-1", groupId : "group-1"){...}
```

변형은 `extensions.invalidateSubscriptions()` 해석기 응답 핸들러 코드에 정의된 대로 구독 무효화를 시작합니다. 그런 다음 AWS AppSync은 클라이언트 A를 구독 취소하고 WebSocket 연결을 종료합니다. 변형에 정의된 무효화 페이로드가 해당 사용자 또는 그룹과 일치하지 않으므로 클라이언트 B는 영향을 받지 않습니다.

AWS AppSync에서 연결을 무효화하면 클라이언트는 구독 취소를 확인하는 메시지를 받습니다.

```
{
  "message": "Subscription complete."
}
```

구독 무효화 필터에 컨텍스트 변수 사용

향상된 구독 필터와 마찬가지로 구독 무효화 필터 확장의 `context` 변수를 사용하여 특정 데이터에 액세스할 수 있습니다.

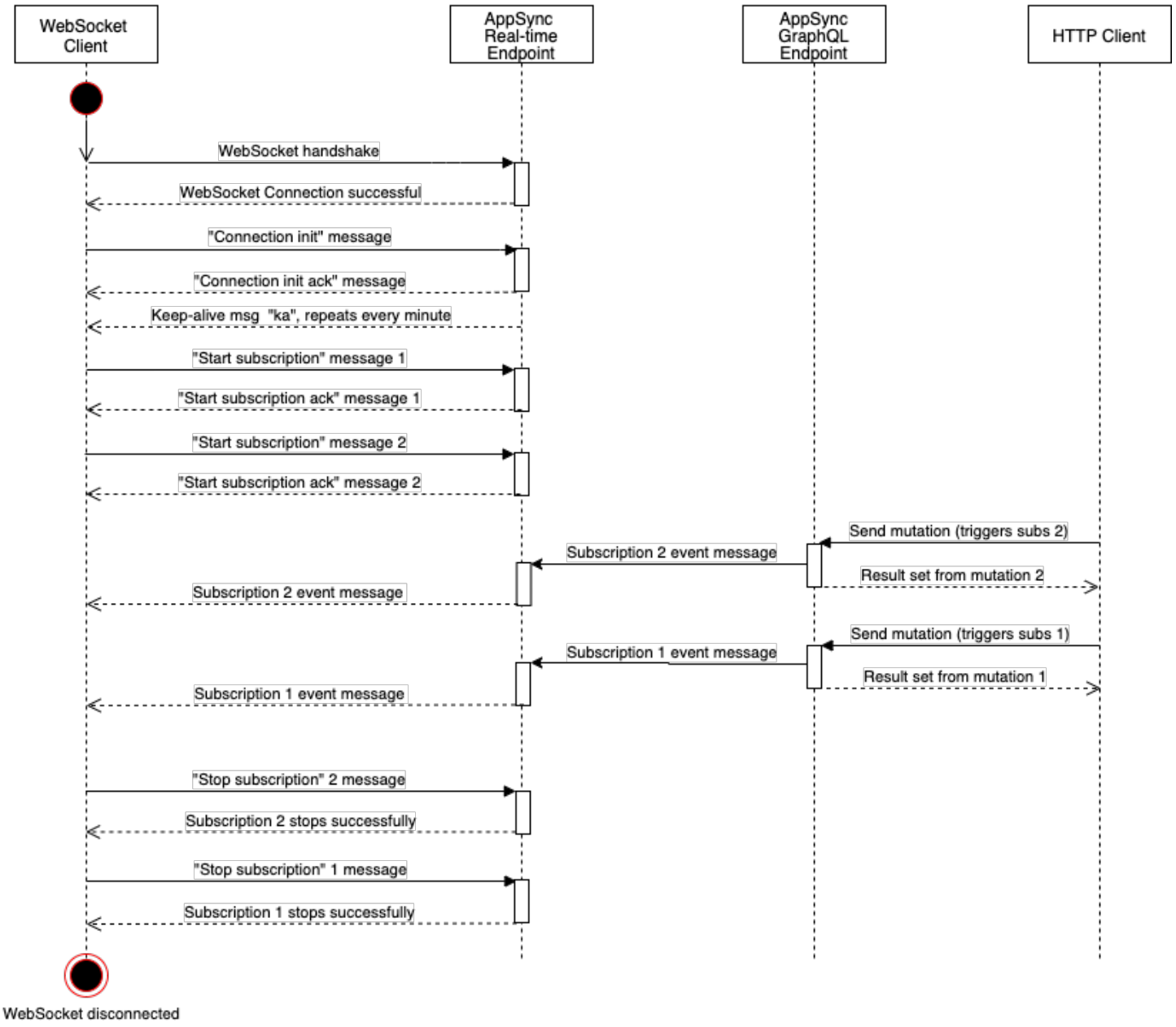
예를 들어, 이메일 주소를 변형의 무효화 페이로드로 구성한 다음 Amazon Cognito 사용자 풀 또는 OpenID Connect를 통해 권한이 부여된 구독 사용자의 이메일 특성 또는 클레임과 일치시킬 수 있습니다. `extensions.setSubscriptionInvalidationFilter()` 구독 무효 검증기에 정의된 무효화 필터는 변형의 `extensions.invalidateSubscriptions()` 페이로드에 의해 설정된 이메일 주소가 `context.identity.claims.email`에 있는 사용자의 JWT 토큰에서 검색된 이메일 주소와 일치하는지 확인하여 무효화를 시작합니다.

실시간 웹소켓 클라이언트 빌드

다음 섹션에서는 AWS AppSync의 실시간 기능을 뒷받침하는 아키텍처를 보여 줍니다.

GraphQL 구독을 위한 실시간 WebSocket 클라이언트 구현

다음 시퀀스 다이어그램과 단계는 WebSocket 클라이언트, HTTP 클라이언트 및 AWS AppSync 간의 실시간 구독 워크플로우를 보여 줍니다.



1. 클라이언트는 AWS AppSync 실시간 엔드포인트와 WebSocket 연결을 설정합니다. 네트워크 오류가 있는 경우 클라이언트는 지터 지수 백오프를 수행해야 합니다. 자세한 내용은 AWS 아키텍처 블로그의 지수 백오프 및 지터를 참조하세요.
2. WebSocket 연결이 성공적으로 설정되면 클라이언트는 `connection_init` 메시지를 전송합니다.
3. 클라이언트는 AWS AppSync의 `connection_ack` 메시지를 대기합니다. 이 메시지에는 `"ka"`(연결 유지) 메시지의 최대 대기 시간(밀리초)인 `connectionTimeoutMs` 파라미터가 포함됩니다.
4. AWS AppSync는 `"ka"` 메시지를 주기적으로 전송합니다. 클라이언트는 각 `"ka"` 메시지를 수신한 시간을 추적합니다. 클라이언트가 `connectionTimeoutMs`밀리초 이내에 `"ka"` 메시지를 수신하지 못하면 클라이언트는 연결을 종료해야 합니다.
5. 클라이언트는 `start` 구독 메시지를 전송하여 구독을 등록합니다. 단일 WebSocket 연결은 서로 다른 인증 모드에 있더라도 여러 구독을 지원합니다.
6. 클라이언트는 성공적인 구독을 확인하기 위해 AWS AppSync에서 `start_ack` 메시지를 전송할 때까지 대기합니다. 오류가 있으면 AWS AppSync는 `"type": "error"` 메시지를 반환합니다.
7. 클라이언트는 해당 변형이 호출된 후 전송되는 구독 이벤트를 수신합니다. 쿼리와 변형은 일반적으로 `https://`를 통해 AWS AppSync GraphQL 엔드포인트로 전송됩니다. 구독은 보안 WebSocket(`wss://`)을 사용하여 AWS AppSync 실시간 엔드포인트를 통해 흐릅니다.
8. 클라이언트는 `stop` 구독 메시지를 전송하여 구독을 등록 취소합니다.
9. 모든 구독을 등록 취소하고 WebSocket을 통해 전송되는 메시지가 없는지 확인한 후, 클라이언트는 WebSocket 연결에서 연결을 해제할 수 있습니다.

WebSocket 연결을 설정하기 위한 핸드셰이크 세부 정보

AWS AppSync와 연결하고 성공적인 핸드셰이크를 시작하려면 WebSocket 클라이언트에 다음이 필요합니다.

- AWS AppSync 실시간 엔드포인트
- `header` 및 `payload` 파라미터가 포함된 쿼리 문자열:
  - `header`: AWS AppSync 엔드포인트 및 권한 부여와 관련된 정보가 포함됩니다. 이 항목은 문자열화된 JSON 객체의 base64로 인코딩된 문자열입니다. JSON 객체 내용은 인증 모드에 따라 다릅니다.
  - `payload`: `payload`의 Base64로 인코딩된 문자열입니다.

WebSocket 클라이언트는 이러한 요구 사항과 함께 `graphql-ws`를 WebSocket 프로토콜로 사용하여 쿼리 문자열이 있는 API 실시간 엔드포인트가 포함된 URL에 연결할 수 있습니다.

GraphQL 엔드포인트에서 실시간 엔드포인트 검색

AWS AppSync GraphQL 엔드포인트와 AWS AppSync 실시간 엔드포인트는 프로토콜과 도메인에서 약간 다릅니다. AWS Command Line Interface(AWS CLI) 명령 `aws appsync get-graphql-api`를 사용하여 GraphQL 엔드포인트를 검색할 수 있습니다.

AWS AppSync GraphQL 엔드포인트:

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync 실시간 엔드포인트:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

애플리케이션은 쿼리와 변형에 대한 HTTP 클라이언트를 사용하여 AWS AppSync GraphQL 엔드포인트(`https://`)에 연결할 수 있습니다. 애플리케이션은 구독에 대한 WebSocket 클라이언트를 사용하여 AWS AppSync 실시간 엔드포인트(`wss://`)에 연결할 수 있습니다.

사용자 지정 도메인 이름을 사용하면 하나의 도메인을 사용하여 두 엔드포인트와 모두 상호 작용할 수 있습니다. 예를 들어 `api.example.com`을 사용자 지정 도메인으로 구성하면 다음 URL을 사용하여 GraphQL 및 실시간 엔드포인트와 상호 작용할 수 있습니다.

AWS AppSync 사용자 지정 도메인 GraphQL 엔드포인트:

```
https://api.example.com/graphql
```

AWS AppSync 사용자 지정 도메인 실시간 엔드포인트:

```
wss://api.example.com/graphql/realtime
```

AWS AppSync API 권한 부여 모드를 기반으로 한 헤더 파라미터 형식

연결 쿼리 문자열에 사용되는 `header` 객체의 형식은 AWS AppSync API 권한 부여 모드에 따라 다릅니다. 객체의 `host` 필드는 AWS AppSync GraphQL 엔드포인트를 참조합니다. 이 엔드포인트는 실시간 엔드포인트에 대해 `wss://` 호출이 수행되더라도 연결의 유효성을 검사하는 데 사용됩니다. 핸드셰이크를 시작하고 인증된 연결을 설정하려면 `payload`가 비어 있는 JSON 객체여야 합니다.

API 키

API 키 헤더

헤더 콘텐츠

- `"host": <string>`: AWS AppSync GraphQL 엔드포인트의 호스트 또는 사용자 지정 도메인 이름입니다.
- `"x-api-key": <string>`: AWS AppSync API용으로 구성된 API 키입니다.

예

```
{
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com",
    "x-api-key":"da2-12345678901234567890123456"
}
```

페이로드 콘텐츠

```
{}
```

요청 URL

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJob3N0IjoiZXhhbXBsZTEyMzQ1Njc4OTAwMDAuYXBwc3luYy1hcGkudXMtZWFzdC0xLmFtYXpvbmF3cy5jb20i
```

Amazon Cognito 사용자 풀 및 OpenID Connect(OIDC)

Amazon Cognito 및 OIDCheader

헤더 콘텐츠:

- `"Authorization": <string>`: JWT ID 토큰입니다. 헤더는 보유자 체계를 사용할 수 있습니다.
- `"host": <string>`: AWS AppSync GraphQL 엔드포인트의 호스트 또는 사용자 지정 도메인 이름입니다.

예제:

```
{
```

```
 "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"
}
```

페이로드 콘텐츠:

```
{}
```

요청 URL:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

IAM

IAM 헤더

헤더 콘텐츠

- `"accept": "application/json, text/javascript"`: 상수 `<string>` 파라미터입니다.
- `"content-encoding": "amz-1.0"`: 상수 `<string>` 파라미터입니다.
- `"content-type": "application/json; charset=UTF-8"`: 상수 `<string>` 파라미터입니다.
- `"host": <string>`: AWS AppSync GraphQL 엔드포인트의 호스트입니다.
  - `"x-amz-date": <string>`: 타임스탬프는 협정 세계시(UTC)이고 YYYYMMDD'T'HHMMSS'Z' ISO 8601 형식이어야 합니다. 예를 들어, 20150830T123600Z는 유효한 타임스탬프입니다. 타임스탬프에 밀리초를 포함하지 마십시오. 자세한 정보는 AWS 일반 참조의 서명 버전 4에서 날짜 처리를 참조하세요.
  - `"X-Amz-Security-Token": <string>`: 임시 보안 자격 증명을 사용할 때 필요한 AWS 세션 토큰입니다. 자세한 내용은 IAM 사용 설명서의 AWS 리소스에서 임시 자격 증명 사용을 참조하세요.

- `"Authorization": <string>`: AWS AppSync 엔드포인트에 대한 Signature Version 4(SigV4) 서명 정보입니다. 서명 프로세스에 대한 자세한 내용은 AWS 일반 참조의 작업 4: HTTP 요청에 서명 추가를 참조하세요.

SigV4 서명 HTTP 요청에는 표준 URL이 포함되며, 이 URL은 `/connect`가 추가된 AWS AppSync GraphQL 엔드포인트입니다. 서비스 엔드포인트 AWS 리전은 AWS AppSync API를 사용하는 리전과 동일하며 서비스 이름은 'appsync'입니다. 서명할 HTTP 요청은 다음과 같습니다.

```
{
  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",
  }
}
```

예

```
{
  "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=XXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
 Signature=83EXAMPLEbcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}
```

페이로드 콘텐츠

```
{}
```

요청 URL

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyEXAMPLEHQiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoE
```

사용자 지정 도메인을 사용하여 요청에 서명하려면 다음을 따르세요.

```
{
  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",
  }
}
```

예

```
{
  "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"
}
```

페이로드 콘텐츠

```
{}
```

요청 URL

```
wss://api.example.com/graphql?
header=eyEXAMPLEHQiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoE
```

> **Note**
>
> 하나의 WebSocket 연결에는 여러 개의 구독이 있을 수 있습니다(각 구독이 서로 다른 인증 모드를 사용하더라도 상관없음). 이 연결을 구현하는 한 가지 방법은 첫 번째 구독에 대한 WebSocket 연결을 만든 후 마지막 구독이 등록 취소되면 이 연결을 닫는 것입니다. 마지막 구독이 등록 취소된 직후에 앱이 구독되는 경우 WebSocket 연결을 닫기 전에 몇 초 동안 대기하

여 이 연결을 최적화할 수 있습니다. 모바일 앱 예제의 경우, 한 화면에서 다른 화면으로 변경할 때 탑재 해제 이벤트에서 구독을 중지하고 탑재 이벤트에서 다른 구독을 시작합니다.

Lambda 권한 부여

Lambda 권한 부여 헤더

헤더 콘텐츠

- `"Authorization": <string>`: authorizationToken으로 전달되는 값입니다.
- `"host": <string>`: AWS AppSync GraphQL 엔드포인트의 호스트 또는 사용자 지정 도메인 이름입니다.

예

```
{

 "Authorization":"M0UzQzM1MkQtMkI0Ni00OTZCLUI1NkQtMUM0MTQ0QjVBRTczCkI1REEzRTIxLTk5NzItNDJENi1BQ
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

페이로드 콘텐츠

```
{}
```

요청 URL

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

실시간 WebSocket 작업

AWS AppSync를 사용하여 성공적인 WebSocket 핸드셰이크를 시작한 후, 클라이언트는 다른 작업을 위해 AWS AppSync에 연결하기 위한 후속 메시지를 전송해야 합니다. 이러한 메시지에는 다음 데이터가 필요합니다.

- `type`: 작업의 유형입니다.
- `id`: 구독의 고유 식별자입니다. UUID를 이러한 목적으로 사용하는 것이 좋습니다.

- payload: 작업 유형에 따라 연결된 페이로드입니다.

type 필드만 필수 필드이며, id 및 payload 필드는 선택 사항입니다.

이벤트 순서

구독 요청을 성공적으로 시작, 설정, 등록 및 처리하려면 클라이언트는 다음 순서를 따라야 단계를 완료해야 합니다.

1. 연결 초기화(connection_init)
2. 연결 승인(connection_ack)
3. 구독 등록(start)
4. 구독 승인(start_ack)
5. 구독 처리(data)
6. 구독 등록 취소(stop)

연결 초기화 메시지

핸드셰이크에 성공한 후 클라이언트는 AWS AppSync 실시간 엔드포인트와 통신을 시작하기 위해 connection_init 메시지를 전송해야 합니다. 이 단계가 없으면 다른 모든 메시지가 무시됩니다. 메시지는 다음 JSON 객체를 다음과 같이 문자열로 변환하여 얻은 문자열입니다.

```
{ "type": "connection_init" }
```

연결 승인 메시지

connection_init 메시지를 전송한 후 클라이언트는 connection_ack 메시지를 대기해야 합니다. connection_ack를 수신하기 전에 전송된 모든 메시지는 무시됩니다. 메시지는 다음과 같은 내용이어야 합니다.

```
{
  "type": "connection_ack",
  "payload": {
    // Time in milliseconds waiting for ka message before the client should terminate
 the WebSocket connection
    "connectionTimeoutMs": 300000
  }
```

```
}
```

연결 유지 메시지

클라이언트는 연결 승인 메시지 외에도 연결 유지 메시지를 주기적으로 수신합니다. 클라이언트가 연결 제한 시간 내에 연결 유지 메시지를 받지 못하면 클라이언트는 연결을 종료해야 합니다. AWS AppSync는 연결이 자동으로 종료될 때까지(24시간 후) 이러한 메시지를 계속 전송하고 등록된 구독에 서비스를 제공합니다. 연결 유지 메시지는 하트비트이며 클라이언트가 승인할 필요가 없습니다.

```
{ "type": "ka" }
```

구독 등록 메시지

클라이언트가 connection_ack 메시지를 수신하고 나면 AWS AppSync에 구독 등록 메시지를 보낼 수 있습니다. 이 메시지 유형은 다음 필드가 포함된 문자열화된 JSON 객체입니다.

- `"id": <string>`: 구독의 ID입니다. 이 ID는 각 구독에 고유해야 합니다. 그렇지 않으면 서버에서 구독 ID가 중복됨을 나타내는 오류를 반환합니다.
- `"type": "start"`: 상수 `<string>` 파라미터입니다.
- `"payload": <Object>`: 구독에 관련된 정보가 포함된 객체입니다.
  - `"data": <string>`: GraphQL 쿼리 및 변수를 포함하는 문자열화된 JSON 객체입니다.
    - `"query": <string>`: GraphQL 작업입니다.
    - `"variables": <Object>`: 쿼리에 대한 변수가 포함된 객체입니다.
  - `"extensions": <Object>`: 권한 부여 객체가 포함된 객체입니다.
- `"authorization": <Object>`:권한 부여에 필요한 필드가 포함된 객체입니다.

구독 등록을 위한 권한 부여 개체

AWS AppSync API 권한 부여 모드를 기반으로 한 헤더 파라미터 형식 섹션의 동일한 규칙이 권한 부여 객체에도 적용됩니다. 유일한 예외는 IAM입니다. 이 경우 SigV4 서명 정보가 약간 다릅니다. 자세한 내용은 IAM 예제를 참조하십시오.

Amazon Cognito 사용자 풀 사용 예:

```
{
  "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"
}
```

IAM 사용 예:

```
{
  "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"
}
```

사용자 지정 도메인 이름을 사용하는 예:

```
{
  "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"
}
```

SigV4 서명은 URL에 /connect를 추가할 필요가 없으며 JSON 문자열화된 GraphQL 작업은 data를 대체합니다. 다음은 SigV4 서명 요청의 예입니다.

```
{
  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",
  }
}
```

구독 승인 메시지

구독 시작 메시지를 전송한 후 클라이언트는 AWS AppSync가 `start_ack` 메시지를 전송할 때까지 대기해야 합니다. `start_ack` 메시지는 구독이 성공했음을 나타냅니다.

구독 승인 예:

```
{
  "type": "start_ack",
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

오류 메시지

연결 초기화 또는 구독 등록이 실패하거나 서버에서 구독이 종료되는 경우 서버는 클라이언트에 오류 메시지를 전송합니다.

- `"type": "error"`: 상수 `<string>` 파라미터입니다.
- `"id": <string>`: 관련된 경우 해당 등록 구독의 ID입니다.
- `"payload" <Object>`: 해당 오류 정보가 포함된 객체입니다.

예제:

```
{
  "type": "error",
  "payload": {
    "errors": [
      {
        "errorType": "LimitExceededError",
        "message": "Rate limit exceeded"
      }
    ]
  }
}
```

데이터 메시지 처리

클라이언트가 변형을 제출하면 AWS AppSync는 관심 있는 모든 구독자를 식별하고 "start" 구독 작업에 사용된 해당 구독 id를 사용하여 각각에게 "type":"data" 메시지를 전송합니다. 클라이언트는 데이터 메시지를 수신할 때 클라이언트가 해당 구독과 일치시킬 수 있도록 전송하는 구독 id를 추적해야 합니다.

- "type": "data": 상수 <string> 파라미터입니다.
- "id": <string>: 해당 등록 구독의 ID입니다.
- "payload" <Object>: 구독 정보가 포함된 객체입니다.

예제:

```
{
  "type": "data",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": {
      "onCreateMessage": {
        "__typename": "Message",
        "message": "test"
      }
    }
  }
}
```

구독 등록 취소 메시지

앱이 구독 이벤트 수신 대기를 중지하려는 경우 클라이언트는 다음과 같은 문자열화된 JSON 객체가 있는 메시지를 전송해야 합니다.

- "type": "stop": 상수 <string> 파라미터입니다.
- "id": <string>: 등록 취소할 구독의 ID입니다.

예제:

```
{
  "type":"stop",
```

```
  "id":"ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

AWS AppSync는 다음과 같은 문자열화된 JSON 객체를 사용하여 확인 메시지를 반송합니다.

- `"type": "complete"`: 상수 `<string>` 파라미터입니다.
- `"id": <string>`: 등록되지 않은 구독의 ID입니다.

클라이언트는 확인 메시지를 받은 후 이 특정 구독에 대한 메시지를 더 이상 받지 않습니다.

예제:

```
{
  "type":"complete",
  "id":"eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

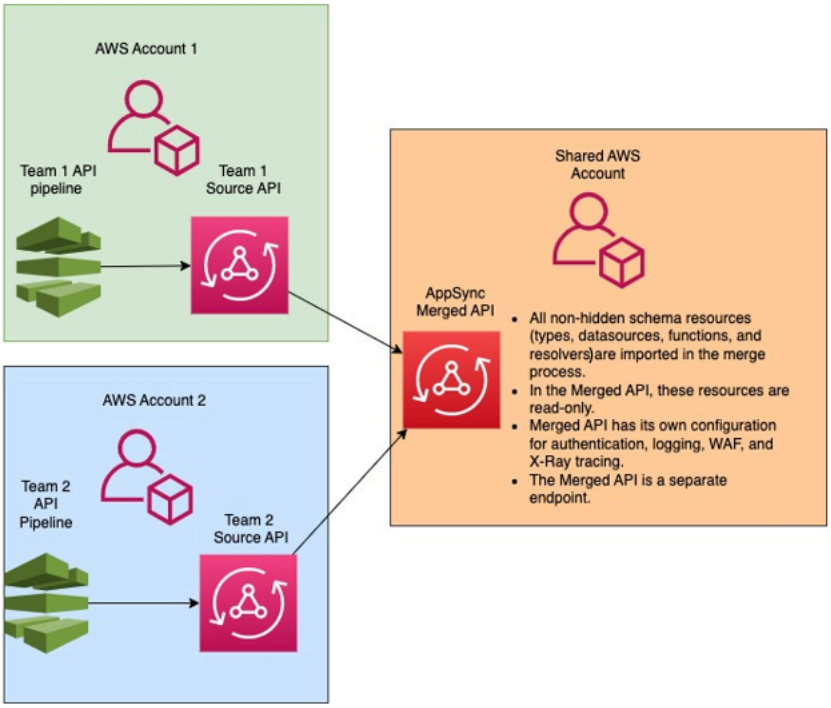
WebSocket 연결 해제

연결을 해제하기 전에 클라이언트는 데이터 손실을 방지하기 위해 WebSocket 연결을 통해 현재 진행 중인 작업이 없는지 확인하는 데 필요한 논리를 갖춰야 합니다. WebSocket에서 연결 해제하기 전에 모든 구독을 등록 취소해야 합니다.

병합된 API

조직 내에서 GraphQL의 사용이 확대됨에 따라 API 사용 편의성과 API 개발 속도 간의 상충 지점이 발생할 수 있습니다. 한편으로 조직은 AWS AppSync와 GraphQL을 채택하여 개발자에게 단일 네트워크 호출로 하나 이상의 데이터 도메인에서 데이터에 안전하게 액세스하고 데이터를 조작 및 결합하는 데 사용할 수 있는 유연한 API를 제공함으로써 애플리케이션 개발을 단순화합니다. 반면, 단일 GraphQL API 엔드포인트로 결합된 여러 데이터 도메인을 담당하는 조직 내 팀은 개발 속도를 높이기 위해 서로 독립적으로 API 업데이트를 생성, 관리 및 배포할 수 있는 기능을 원할 수 있습니다.

이러한 긴장을 해소하기 위해 AWS AppSync 병합 API 기능을 사용하면 서로 다른 데이터 도메인의 팀이 독립적으로 AWS AppSync API(예: GraphQL 스키마, 해석기, 데이터 원본, 함수)를 만들고 배포한 다음 이를 하나의 병합된 API로 결합할 수 있습니다. 이를 통해 조직은 사용이 간편한 교차 도메인 API를 유지 관리하고 해당 API에 기여하는 여러 팀이 빠르고 독립적으로 API 업데이트를 수행할 수 있습니다.



조직은 병합된 API를 사용하여 여러 독립 소스 AWS AppSync API의 리소스를 하나의 AWS AppSync 병합 API 엔드포인트로 가져올 수 있습니다. 이를 위해 AWS AppSync를 사용하면 소스 AWS AppSync 소스 API 목록을 만든 다음 스키마, 형식, 데이터 원본, 해석기, 함수 등 소스 API와 관련된 모든 메타데이터를 새로운 AWS AppSync 병합 API에 병합할 수 있습니다.

병합 중에 소스 API 데이터 콘텐츠의 불일치로 인해 병합 충돌이 발생할 수 있습니다(예: 여러 스키마 결합 시 형식 이름 충돌). 소스 API의 정의가 충돌하지 않는 간단한 사용 사례의 경우 소스 API 스키마를 수정할 필요가 없습니다. 이렇게 생성된 병합된 API는 단순히 원래 소스 AWS AppSync API에서 모든 형식, 해석기, 데이터원본스 및 기능을 가져옵니다. 충돌이 발생하는 복잡한 사용 사례의 경우 사용자/팀은 다양한 방법을 통해 충돌을 해결해야 합니다. AWS AppSync에서는 사용자에게 병합 충돌을 줄일 수 있는 몇 가지 도구와 예를 제공합니다.

AWS AppSync에서 구성된 후속 병합은 소스 API의 변경 사항을 연관된 병합 API에 전파합니다.

병합된 API 및 페더레이션

GraphQL 커뮤니티에는 GraphQL 스키마를 결합하고 공유 그래프를 통해 팀 협업을 가능하게 하는 다양한 솔루션과 패턴이 있습니다. AWS AppSync 병합된 API는 소스 API를 별도의 병합된 API로 결합하는 빌드 타임 방식을 스키마 구성에 채택합니다. 또 다른 방법으로는 런타임 라우터를 여러 소스 API 또는 하위 그래프에 계층화하는 방식이 있습니다. 이 방식에서는 라우터가 요청을 수신하고, 메타데이

터로 유지 관리하는 결합된 스키마를 참조하고, 요청 계획을 구성한 다음, 기본 하위 그래프/서버에 요청 요소를 배포합니다. 다음 표에서는 AWS AppSync 병합 API 빌드 타임 방식을 GraphQL 스키마 구성에 대한 라우터 기반 런타임 방식과 비교합니다.

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/deserializati on, 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 associate d 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

* AWS AppSync 병합 API는 AWS AppSync 소스 API와만 연결할 수 있습니다. AWS AppSync 하위 그래프 및 AWS AppSync가 아닌 하위 그래프에 스키마 구성 지원이 필요한 경우 하나 이상의 AWS AppSync GraphQL 또는 병합된 API를 라우터 기반 솔루션에 연결할 수 있습니다. 예를 들어 Apollo Federation v2에서 라우터 기반 아키텍처를 사용하여 AWS AppSync API를 하위 그래프로 추가하는 방법에 대해서는 블로그 AWS AppSync와 Apollo GraphQL 페더레이션을 참조하세요.

주제

- 병합된 API 충돌 해결
- 스키마 구성
- 권한 부여 모드 구성
- 실행 역할 구성
- AWS RAM을 사용하여 교차 계정 병합 API 구성
- 병합
- 병합된 API에 대한 추가 지원
- 병합된 API 제한 사항
- 병합된 API 생성

병합된 API 충돌 해결

AWS AppSync에서는 병합 충돌 발생 시 문제를 해결하는 데 도움이 되는 몇 가지 도구와 예를 제공합니다.

병합된 API 스키마 지시문

AWS AppSync에서는 소스 API 간의 충돌을 줄이거나 해결하는 데 사용할 수 있는 몇 가지 GraphQL 지시문을 도입했습니다.

- @canonical: 이 지시문은 이름과 데이터가 비슷한 형식/필드의 우선 순위를 설정합니다. 둘 이상의 소스 API에 동일한 GraphQL 형식 또는 필드가 있는 경우 API 중 하나가 형식 또는 필드에

canonical로 주석을 달아 병합 중에 우선 순위가 지정되도록 할 수 있습니다. 다른 소스 API에서 이 지시문으로 주석을 달지 않은 충돌하는 형식/필드는 병합될 때 무시됩니다.

- @hidden: 이 지시문은 특정 형식/필드를 캡슐화하여 병합 프로세스에서 제거합니다. 팀에서 내부 클라이언트만 특정 형식의 데이터에 액세스할 수 있도록 소스 API에서 특정 형식이나 작업을 제거하거나 숨기고 싶을 수 있습니다. 이 지시문이 연결되면 형식이나 필드가 병합된 API에 병합되지 않습니다.
- @renamed: 이 지시문은 형식/필드의 이름을 변경하여 이름 충돌을 줄입니다. 서로 다른 API에 동일한 형식 또는 필드 이름이 사용되는 경우가 있습니다. 그러나 이들 모두 병합된 스키마에서 사용할 수 있어야 합니다. 모든 항목을 병합된 API에 포함하는 간단한 방법은 필드 이름을 비슷하지만 다른 이름으로 바꾸는 것입니다.

제공되는 유틸리티 스키마 지시문을 표시하려면 다음 예를 고려해 보세요.

이 예에서는 두 개의 소스 API를 병합한다고 가정합니다. 게시물(예: 댓글 섹션 또는 소셜 미디어 게시물)을 생성하고 검색하는 두 개의 스키마가 제공됩니다. 형식과 필드가 매우 유사하다고 가정하면 병합 작업 중에 충돌이 발생할 가능성이 높습니다. 아래 스니펫은 각 스키마의 형식과 필드를 보여 줍니다.

Source1.graphql이라는 첫 번째 파일은 사용자가 `putPost` 변형을 사용하여 `Posts`를 생성할 수 있게 하는 GraphQL 스키마입니다. 각 `Post`에는 제목과 ID가 포함되어 있습니다. ID는 `User` 또는 게시자의 정보(이메일 및 주소)와 `Message` 또는 페이로드(콘텐츠)를 참조하는 데 사용됩니다. `User` 형식에는 @canonical 태그로 주석이 달립니다.

```
# 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
}
```

Source2.graphql이라는 두 번째 파일은 Source1.graphql과 매우 유사한 작업을 수행하는 GraphQL 스키마입니다. 하지만 각 형식의 필드는 서로 다르다는 점에 유의하세요. 이 두 스키마를 병합하면 이러한 차이로 인해 병합 충돌이 발생합니다.

또한 Source2.graphql에는 이러한 충돌을 줄이기 위한 몇 가지 지시문이 포함되어 있다는 점도 참고하세요. `Post` 형식에는 병합 작업 중에 자체적으로 난독화할 수 있도록 @hidden 태그가 주석으로 달려 있습니다. `Message` 형식에는 다른 `Message` 형식과 이름이 충돌하는 경우 형식 이름을 `ChatMessage`로 수정할 수 있도록 @renamed 태그가 주석으로 달려 있습니다.

```
# 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")
}
```

병합이 발생하면 다음과 같이 MergedSchema.graphql 파일이 생성됩니다.

```
# 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
}
```

병합에서 몇 가지 사항이 발생했습니다.

- @canonical 주석으로 인해 Source1.graphql의 `User` 형식이 Source2.graphql의 `User` 형식보다 우선 순위로 지정되었습니다.
- Source1.graphql의 `Message` 형식이 병합에 포함되었습니다. 하지만 Source2.graphql의 `Message`에 이름 충돌이 발생했습니다. @renamed 주석으로 인해 병합에 함께 포함되었지만 대체 이름인 `ChatMessage`가 사용되었습니다.
- Source1.graphql의 `Post` 형식은 포함되었지만, Source2.graphql의 `Post` 형식은 포함되지 않았습니다. 일반적으로는 이 형식에서 충돌이 발생하지만 Source2.graphql의 `Post` 형식에 @hidden 주석이 있으므로 해당 데이터가 난독화되어 병합에 포함되지 않았습니다. 그 결과 충돌은 발생하지 않았습니다.
- 두 파일의 내용을 모두 포함하도록 `Query` 형식이 업데이트되었습니다. 그러나 지시문으로 인해 `GetMessage` 쿼리 하나의 이름이 `GetChatMessage`로 변경되었습니다. 이를 통해 이름이 같은 두 쿼리 간의 이름 충돌이 해결되었습니다.

충돌하는 유형에 지시문이 추가되지 않는 경우도 있습니다. 여기서, 병합된 형식에는 해당 형식의 모든 소스 정의에 있는 모든 필드의 조합이 포함됩니다. 예를 들어 다음 예제를 고려해 보십시오.

Source1.graphql이라는 이 스키마를 사용하면 `Posts`를 생성하고 검색할 수 있습니다. 구성은 이전 예제와 비슷하지만 정보가 더 적습니다.

```
# 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
}
```

Source2.graphql이라는 이 스키마를 사용하면 `Reviews`(예: 영화 평점 또는 음식점 리뷰)를 생성하고 검색할 수 있습니다. `Reviews`는 ID 값이 동일한 `Post`와 연결됩니다. 여기에는 전체 리뷰 게시물의 제목, 게시물 ID, 페이로드 메시지가 모두 포함됩니다.

병합하면 두 Post 형식 간에 충돌이 발생합니다. 이 문제를 해결할 주석이 없으므로 기본 동작은 충돌하는 형식에 대해 조합 작업을 수행하는 것입니다.

```
# 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
}
```

병합이 발생하면 다음과 같이 MergedSchema.graphql 파일이 생성됩니다.

```
# 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
}
```

병합에서 몇 가지 사항이 발생했습니다.

- `Mutation` 형식은 충돌이 없었고 병합되었습니다.
- `Post` 형식 필드는 조합 작업을 통해 결합되었습니다. 이 둘의 조합으로 어떻게 단일 `id`, `title` 및 단일 `reviews`가 생성했는지를 주목해 보세요.
- `Review` 형식은 충돌이 없었고 병합되었습니다.
- `Query` 형식은 충돌이 없었고 병합되었습니다.

공유 형식의 해석기 관리

위 예제에서 Source1.graphql이 `PostDatasource`라는 DynamoDB 데이터 원본을 사용하는 `Query.getPost`에 유닛 해석기를 구성한 경우를 고려해 보세요. 이 해석기는 `Post` 형식의 `id` 및 `title`을 반환합니다. 이제 Source2.graphql에서 두 함수를 실행하는 `Post.reviews`에 파이프라인 해석기를 구성했다고 가정해 보겠습니다. `Function1`에는 사용자 지정 권한 부여 검사를 수행할 수 있는 None 데이터 원본이 연결되어 있습니다. `Function2`에는 `reviews` 테이블을 쿼리하기 위해 DynamoDB 데이터 원본이 연결되어 있습니다.

```
query GetPostQuery {
    getPost(id: "1") {
        id,
        title,
        reviews
    }
}
```

클라이언트가 병합된 API 엔드포인트에 대해 위 쿼리를 실행하면 AWS AppSync 서비스는 먼저 `Source1`의 `Query.getPost`에 대해 유닛 해석기를 실행하여 `PostDatasource`를 호출하고 DynamoDB에서 데이터를 반환합니다. 그런 다음 `Function1`이 사용자 지정 권한 부여 로직을 수행하고 `Function2`가 `$context.source`에서 발견된 `id`에 제공된 리뷰를 반환하는 `Post.reviews` 파이프라인 해석기를 실행합니다. 서비스는 요청을 단일 GraphQL 실행으로 처리하며 이 단순한 요청에는 단일 요청 토큰만 필요합니다.

공유 형식의 해석기 충돌 관리

Source2의 필드 해석기 외에도 한 번에 여러 필드를 제공하기 위해 Query.getPost에 해석기도 구현하는 다음과 같은 경우를 생각해 보세요. Source1.graphql은 다음과 같을 수 있습니다.

```
# This snippet represents a file called Source1.graphql

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

type Query {
    getPost(id: ID!): Post
}
```

Source2.graphql은 다음과 같을 수 있습니다.

```
# This snippet represents a file called Source2.graphql

type Post  {
  id: ID!
  content: String!
  contentHash: String!
  author: String!
}

type Query {
    getPost(id: ID!): Post
}
```

이 두 스키마를 병합하려고 하면 병합 오류가 발생합니다. AWS AppSync 병합된 API는 여러 소스 해석기를 동일한 필드에 연결할 수 없기 때문입니다. 이 충돌을 해결하려면 Source2.graphql이 Post 형식에서 소유한 필드를 정의하는 별도의 형식을 추가해야 하는 필드 해석기 패턴을 구현할 수 있습니다. 다음 예제에서는 PostInfo라는 형식을 추가하는데, 여기에는 Source2.graphql에서 해석할 내용 및 작성자 필드가 포함되어 있습니다. Source1.graphql은 Query.getPost에 연결된 해석기를 구현하고, Source2.graphql은 이제 Post.postInfo에 해석기를 연결하여 모든 데이터가 성공적으로 검색될 수 있도록 합니다.

```
type Post  {
```

```
  id: ID!
  postInfo: PostInfo
}

type PostInfo {
   content: String!
   contentHash: String!
   author: String!
}

type Query {
    getPost(id: ID!): Post
}
```

이러한 충돌을 해결하려면 소스 API 스키마를 다시 작성하고 잠재적으로 클라이언트도 쿼리를 변경해야 하지만, 이 접근 방식의 장점은 병합된 해석기의 소유권이 소스 팀 간에 명확하게 유지된다는 것입니다.

스키마 구성

두 당사자는 병합된 API를 생성하기 위한 스키마 구성을 담당합니다.

- 병합된 API 소유자 - 병합된 API 소유자는 병합된 API의 권한 부여 로직과 고급 설정(예: 로깅, 추적, 캐싱, WAF 지원)을 구성해야 합니다.
- 관련 소스 API 소유자 - 관련 API 소유자는 병합된 API를 구성하는 스키마, 해석기, 데이터 원본을 구성해야 합니다.

병합된 API의 스키마는 관련 소스 API의 스키마에서 생성되므로 읽기 전용입니다. 즉, 스키마 변경은 소스 API에서 시작해야 합니다. AWS AppSync 콘솔에서 스키마 창 위의 드롭다운 목록을 사용하여 병합된 API에 포함된 소스 API의 병합된 스키마와 개별 스키마 사이를 전환할 수 있습니다.

권한 부여 모드 구성

여러 권한 부여 모드를 사용하여 병합된 API를 보호할 수 있습니다. AWS AppSync의 권한 부여 모드에 대해 자세히 알아보려면 권한 부여 및 인증을 참조하세요.

병합된 API와 함께 사용할 수 있는 권한 부여 모드는 다음과 같습니다.

- API 키: 가장 간단한 권한 부여 전략입니다. 모든 요청에는 `x-api-key` 요청 헤더 아래에 API 키가 포함되어야 합니다. 만료된 API 키는 만료일 이후 60일 동안 보관됩니다.

- AWS Identity and Access Management(IAM): AWS IAM 권한 부여 전략은 sigv4로 서명된 모든 요청을 승인합니다.
- Amazon Cognito 사용자 풀: Amazon Cognito 사용자 풀을 통해 사용자에게 권한을 부여하여 보다 세밀하게 제어할 수 있도록 합니다.
- AWS Lambda Authorizers: 사용자 지정 로직을 사용하여 AWS AppSync API에 대한 액세스를 인증하고 권한을 부여할 수 있는 서버리스 함수입니다.
- OpenID Connect: 이 권한 부여 유형은 OIDC 호환 서비스에서 제공하는 OpenID Connect(OIDC) 토큰을 강제로 적용합니다. 애플리케이션에서는 OIDC 공급자가 액세스 제어를 위해 정의한 사용자 및 권한을 활용합니다.

병합된 API의 권한 부여 모드는 병합된 API 소유자가 구성합니다. 병합 작업 시 병합된 API는 소스 API에 구성된 기본 권한 부여 모드를 자체 기본 권한 부여 모드 또는 보조 권한 부여 모드로 포함해야 합니다. 그렇지 않으면 호환되지 않고 충돌이 발생하여 병합 작업이 실패합니다. 소스 API에서 다중 인증 지시문을 사용하는 경우 병합 프로세스에서 이러한 지시문을 통합 엔드포인트에 자동으로 병합할 수 있습니다. 소스 API의 기본 권한 부여 모드가 병합된 API의 기본 권한 부여 모드와 일치하지 않는 경우 소스 API의 형식에 대한 권한 부여 모드가 일관되도록 하기 위해 이러한 인증 지시문을 자동으로 추가합니다.

실행 역할 구성

병합된 API를 생성할 때는 서비스 역할을 정의해야 합니다. AWS 서비스 역할은 AWS 서비스가 사용자를 대신하여 작업을 수행하는 데 사용되는 AWS Identity and Access Management(IAM) 역할입니다.

이러한 맥락에서 병합된 API는 소스 API에 구성된 데이터 원본의 데이터에 액세스하는 해석기를 실행해야 합니다. 이를 위한 필수 서비스 역할은 `mergedApiExecutionRole`이며, `appsync:SourceGraphQL` IAM 권한을 통해 병합된 API에 포함된 소스 API에서 요청을 실행할 수 있는 명시적 액세스 권한이 있어야 합니다. GraphQL 요청을 실행하는 동안 AWS AppSync 서비스는 이 서비스 역할을 맡고 `appsync:SourceGraphQL` 작업을 수행할 역할을 승인합니다.

AWS AppSync에서는 IAM 권한 부여 모드가 IAM API에 작동하는 방식과 같이 요청 내의 특정 최상위 필드에 대해 이 권한을 허용하거나 거부할 수 있습니다. 최상위 수준이 아닌 필드의 경우 AWS AppSync에서는 소스 API ARN 자체에 대한 권한을 정의하도록 요구합니다. 병합된 API의 최상위 수준이 아닌 특정 필드에 대한 액세스를 제한하려면 Lambda 내에서 사용자 지정 로직을 구현하거나 @hidden 지시문을 사용하여 병합된 API에서 소스 API 필드를 숨기는 것이 좋습니다. 역할이 소스 API 내에서 모든 데이터 작업을 수행하도록 허용하려면 아래 정책을 추가할 수 있습니다. 첫 번째 리소스 항목은 모든 최상위 필드에 대한 액세스를 허용하고 두 번째 항목은 소스 API 리소스 자체에서 권한을 부여하는 하위 해석기를 포함합니다.

```
{
    "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"]
    }]
}
```

특정 최상위 필드로만 액세스를 제한하려면 다음과 같은 정책을 사용할 수 있습니다.

```
{
    "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"]
    }]
}
```

또한 AWS AppSync 콘솔 API 생성 마법사를 사용하여 서비스 역할을 생성해 병합된 API가 병합된 API와 동일한 계정에 있는 소스 API에 구성된 리소스에 액세스할 수 있도록 허용할 수도 있습니다. 소스 API가 병합된 API와 동일한 계정에 있지 않은 경우 먼저 AWS Resource Access Manager(AWS RAM)를 사용하여 리소스를 공유해야 합니다.

AWS RAM을 사용하여 교차 계정 병합 API 구성

병합된 API를 생성할 때 선택적으로 AWS Resource Access Manager(AWS RAM)를 통해 공유된 다른 계정의 소스 API를 연결할 수 있습니다. AWS RAM은 AWS 계정 간, 조직 또는 조직 단위(OU) 내, IAM 역할 및 사용자와 리소스를 안전하게 공유하는 데 도움이 됩니다.

AWS AppSync는 단일 병합 API에서 여러 계정의 소스 API를 구성하고 액세스할 수 있도록 하기 위해 AWS RAM과 통합됩니다. AWS RAM을 사용하면 리소스 공유를 생성하거나 각 리소스에 대해 공유될 리소스 및 권한 집합의 컨테이너를 생성할 수 있습니다. AWS RAM에서 리소스 공유에 AWS AppSync API를 추가할 수 있습니다. 리소스 공유 내에서 AWS AppSync는 RAM의 AWS AppSync API와 연결할 수 있는 세 가지 권한 집합을 제공합니다.

1. AWSRAMPermissionAppSyncSourceApiOperationAccess: 다른 권한이 지정되지 않은 경우 AWS RAM에 AWS AppSync API를 공유할 때 추가되는 기본 권한 집합입니다. 이 권한 집합은 병합된 API 소유자와 소스 AWS AppSync API를 공유하는 데 사용됩니다. 이 권한 집합에는 소스 API의 appsync:AssociateMergedGraphqlApi에 대한 권한과 런타임 시 소스 API 리소스에 액세스하는 데 필요한 appsync:SourceGraphQL 권한이 포함됩니다.
2. AWSRAMPermissionAppSyncMergedApiOperationAccess: 이 권한 집합은 소스 API 소유자와 병합된 API를 공유할 때 구성해야 합니다. 이 권한 집합은 대상 보안 주체가 소유한 모든 소스 API를 병합된 API에 연결하고 병합된 API의 소스 API 연결을 읽고 업데이트하는 기능을 포함하여 병합된 API를 구성할 수 있는 기능을 소스 API에 제공합니다.
3. AWSRAMPermissionAppSyncAllowSourceGraphQLAccess: 이 권한 집합은 AWS AppSync API에서 appsync:SourceGraphQL 권한이 사용될 수 있도록 합니다. 병합된 API 소유자와 소스 API를 공유하는 데 사용되도록 의도된 권한 집합입니다. 소스 API 작업 액세스를 위한 기본 권한 집합과 달리 이 권한 집합에는 런타임 권한 appsync:SourceGraphQL만 포함됩니다. 사용자가 병합된 API 작업 액세스를 소스 API 소유자와 공유하기로 선택한 경우, 병합된 API 엔드포인트를 통해 런타임 액세스 권한을 가지려면 소스 API의 이 권한을 병합된 API 소유자에게 공유해야 합니다.

AWS AppSync에서는 고객 관리 권한도 지원합니다. 제공된 AWS 관리형 권한 중 하나가 작동하지 않는 경우 고객 관리형 권한을 직접 생성할 수 있습니다. 고객 관리형 권한은 AWS RAM을 사용하여 공유되는 리소스가 어떤 조건에서 어떤 작업을 수행할 수 있는지 정확하게 지정하여 작성하고 유지 관리하는 관리형 권한입니다. AWS AppSync를 사용하면 권한을 직접 생성할 때 다음 작업 중에서 선택할 수 있습니다.

1. appsync:AssociateSourceGraphqlApi
2. appsync:AssociateMergedGraphqlApi
3. appsync:GetSourceApiAssociation
4. appsync:UpdateSourceApiAssociation
5. appsync:StartSchemaMerge
6. appsync:ListTypesByAssociation
7. appsync:SourceGraphQL

AWS RAM에서 소스 API 또는 병합된 API를 적절하게 공유하고 필요한 경우 리소스 공유 초대를 수락했다면 병합된 API에서 소스 API 연결을 생성하거나 업데이트할 때 AWS AppSync 콘솔에 표시됩니다. AWS AppSync가 제공하는 ListGraphqlApis 작업을 호출하고 OTHER_ACCOUNTS 소유자 필터

를 사용하여 설정된 권한에 관계없이 AWS RAM을 사용하여 공유된 모든 AWS AppSync API를 계정에 나열할 수도 있습니다.

> **Note**
>
> AWS RAM을 통해 공유하려면 AWS RAM의 호출자에게 공유 중인 API에서 `appsync:PutResourcePolicy` 작업을 수행할 권한이 있어야 합니다.

병합

병합 관리

병합된 API는 통합된 AWS AppSync 엔드포인트에서 팀 협업을 지원하는 것이 목적입니다. 팀은 백엔드에서 격리된 자체 소스 GraphQL API를 독립적으로 발전시킬 수 있으며, AWS AppSync 서비스는 단일 병합 API 엔드포인트로의 리소스 통합을 관리하여 협업의 어려움을 줄이고 개발 리드 타임을 줄일 수 있습니다.

자동 병합

소스 API를 변경한 후 AWS AppSync 병합 API에 연결된 소스 API를 병합된 API에 자동으로 병합(자동 병합)하도록 구성할 수 있습니다. 이렇게 하면 소스 API의 변경 사항이 백그라운드에서 병합된 API 엔드포인트로 항상 전달됩니다. 소스 API 스키마의 모든 변경 사항은 병합된 API의 기존 정의와 병합 충돌을 일으키지 않는 한 병합된 API에서 업데이트됩니다. 소스 API의 업데이트가 해석기, 데이터 원본 또는 함수를 업데이트하는 경우 가져온 리소스도 업데이트됩니다. 자동으로 해결(자동 해결)할 수 없는 새로운 충돌이 발생하면 병합 작업 중에 지원되지 않는 충돌로 인해 병합된 API 스키마 업데이트가 거부됩니다. 콘솔에서 상태가 `MERGE_FAILED`인 각 소스 API 연결에 대한 오류 메시지를 확인할 수 있습니다. 다음과 같이 AWS SDK를 사용하거나 AWS CLI를 사용하여 지정된 소스 API 연결에 대한 `GetSourceApiAssociation` 작업을 호출하여 오류 메시지를 검사할 수도 있습니다.

```
aws appsync get-source-api-association --merged-api-identifier <Merged API ARN> --
association-id <SourceApiAssociation id>
```

그러면 다음 형식의 결과가 생성됩니다.

```
{
    "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."
    }
}
```

수동 병합

소스 API의 기본 설정은 수동 병합입니다. 병합된 API가 마지막으로 업데이트된 이후 소스 API에서 발생한 변경 사항을 병합하기 위해 소스 API 소유자는 AWS AppSync 콘솔에서 또는 AWS SDK 및 AWS CLI에서 사용할 수 있는 `StartSchemaMerge` 작업을 통해 수동 병합을 간접적으로 호출할 수 있습니다.

병합된 API에 대한 추가 지원

구독 구성

GraphQL 스키마 구성에 대한 라우터 기반 접근 방식과 달리 AWS AppSync 병합 API는 GraphQL 구독에 대한 기본 지원을 제공합니다. 관련 소스 API에 정의된 모든 구독 작업은 수정 없이 병합된 API에 자동으로 병합되어 작동합니다. AWS AppSync에서 서버리스 WebSocket 연결을 통한 구독을 지원하는 방법에 대해 자세히 알아보려면 실시간 데이터를 참조하세요.

관측성 구성

AWS AppSync 병합 API는 Amazon CloudWatch를 통해 내장된 로깅, 모니터링 및 메트릭을 제공합니다. 또한 AWS AppSync는 AWS X-Ray를 통한 추적을 위한 기본 지원을 제공합니다.

사용자 지정 도메인 구성

AWS AppSync 병합 API는 병합된 API의 GraphQL 및 실시간 엔드포인트와 함께 사용자 지정 도메인을 사용하기 위한 기본 지원을 제공합니다.

캐싱 구성

AWS AppSync 병합된 API는 요청 수준 또는 해석기 수준 응답을 선택적으로 캐싱하고 응답 압축을 위한 기본 지원을 제공합니다. 자세히 알아보려면 캐싱 및 압축을 참조하세요.

프라이빗 API 구성

AWS AppSync 병합된 API는 병합된 API의 GraphQL 및 실시간 엔드포인트에 대한 액세스를 구성 가능한 VPC 엔드포인트에서 발생하는 트래픽으로 제한하는 프라이빗 API에 대한 기본 지원을 제공합니다.

방화벽 규칙 구성

AWS AppSync 병합된 API는 AWS WAF에 대한 기본 지원을 제공하며, 이를 통해 웹 애플리케이션 방화벽 규칙을 정의하여 API를 보호할 수 있습니다.

감사 로그 구성

AWS AppSync 병합된 API는 감사 로그를 구성하고 관리할 수 있는 AWS CloudTrail에 대한 기본 지원을 제공합니다.

병합된 API 제한 사항

병합된 API를 개발하기 전에 다음 규칙을 고려해야 합니다.

1. 병합된 API는 다른 병합된 API의 소스 API가 될 수 없습니다.
2. 소스 API는 둘 이상의 병합된 API와 연결할 수 없습니다.
3. 병합된 API 스키마 문서의 기본 크기 제한은 10MB입니다.
4. 병합된 API와 연결할 수 있는 소스 API의 수는 기본적으로 10개입니다. 그러나 병합된 API에서 10개 이상의 소스 API가 필요하다면 제한 증가를 요청할 수 있습니다.

병합된 API 생성

콘솔에서 병합된 API를 만드는 방법

1. AWS Management Console 에 로그인한 다음 AWS AppSync 콘솔을 엽니다.
   - 대시보드에서 API 생성을 선택합니다.
2. 병합된 API를 선택한 후 다음을 선택합니다.
3. API 세부 정보 지정 페이지에서 다음 정보를 입력합니다.
   a. API 세부 정보에서 다음 정보를 입력합니다.

i. 병합된 API의 API 이름을 지정합니다. 이 필드에서 GraphQL API에 레이블을 지정하여 다른 GraphQL API와 쉽게 구분할 수 있습니다.

ii. 연락처 세부 정보를 지정합니다. 이 필드는 선택 사항이며 GraphQL API에 이름 또는 그룹을 연결합니다. 이 필드는 다른 리소스에 연결되거나 다른 리소스에 의해 생성되지 않으며 API 이름 필드와 매우 유사하게 작동합니다.

b. 서비스 역할에서 병합된 API에 IAM 실행 역할을 연결해야 AWS AppSync에서 런타임 시 리소스를 안전하게 가져오고 사용할 수 있습니다. 새 서비스 역할 생성 및 사용을 선택하여 AWS AppSync에서 사용할 정책과 리소스를 지정할 수 있습니다. 기존 서비스 역할 사용을 선택한 다음 드롭다운 목록에서 역할을 선택하여 기존 IAM 역할을 가져올 수도 있습니다.

c. 프라이빗 API 구성에서 프라이빗 API 기능을 활성화할 수 있습니다. 단, 병합된 API를 생성한 후에는 이 선택을 변경할 수 없습니다. 프라이빗 API에 대한 자세한 내용은 AWS AppSync 프라이빗 API 사용을 참조하세요.

완료했으면 다음을 선택합니다.

4. 다음으로 병합된 API의 기반으로 사용할 GraphQL API를 추가해야 합니다. 소스 API 선택 페이지에서 다음 정보를 입력합니다.

a. AWS 계정의 API 테이블에서 소스 API 추가를 선택합니다. GraphQL API 목록의 각 항목에는 다음 데이터가 포함됩니다.

i. 이름: GraphQL API의 API 이름 필드입니다.

ii. API ID: GraphQL API의 고유한 ID 값입니다.

iii. 기본 인증 모드: GraphQL API의 기본 권한 부여 모드입니다. AWS AppSync의 권한 부여 모드에 대한 자세한 내용은 권한 부여 및 인증을 참조하세요.

iv. 추가 인증 모드: GraphQL API에 구성된 보조 권한 부여 모드입니다.

v. API의 이름 필드 옆에 있는 확인란을 선택하여 병합된 API에서 사용할 API를 선택합니다. 그런 다음 소스 API 추가를 선택합니다. 선택한 GraphQL API는 AWS 계정의 API 테이블에 표시됩니다.

b. 다른 AWS 계정의 API 테이블에서 소스 API 추가를 선택합니다. 이 목록에 있는 GraphQL API는 AWS Resource Access Manager(AWS RAM)을 통해 리소스를 공유하는 다른 계정에서 가져옵니다. 이 테이블의 GraphQL API를 선택하는 프로세스는 이전 섹션의 프로세스와 동일합니다. AWS RAM을 통해 리소스를 공유하는 방법에 대한 자세한 내용은 AWS Resource Access Manager이란?을 참조하세요.

완료했으면 다음을 선택합니다.

c. 기본 인증 모드를 추가합니다. 자세한 내용은 권한 부여 및 인증을 참조하세요. 다음을 선택합니다.

d. 입력 내용을 검토한 다음 API 생성을 선택합니다.

RDS 내부 검사

AWS AppSync는 기존 관계형 데이터베이스에서 API를 쉽게 구축할 수 있습니다. 내부 검사 유틸리티는 데이터베이스 테이블에서 모델을 검색하고 GraphQL 유형을 제안할 수 있습니다. AWS AppSync 콘솔의 API 생성 마법사는 Aurora MySQL 또는 PostgreSQL 데이터베이스에서 API를 즉시 생성할 수 있습니다. 데이터를 읽고 쓸 수 있는 유형과 JavaScript 해석기를 자동으로 생성합니다.

AWS AppSync는 Amazon RDS 데이터 API를 통한 Amazon Aurora 데이터베이스와의 직접 통합을 제공합니다. Amazon RDS Data API는 지속적인 데이터베이스 연결을 요구하는 대신 SQL 문 실행을 위해 AWS AppSync가 연결하는 안전한 HTTP 엔드포인트를 제공합니다. 이를 사용하여 Aurora에서 MySQL 및 PostgreSQL 워크로드를 위한 관계형 데이터베이스 API를 생성할 수 있습니다.

AWS AppSync로 관계형 데이터베이스용 API를 구축하면 다음과 같은 몇 가지 이점이 있습니다.

- 데이터베이스는 액세스 포인트를 데이터베이스 자체에서 분리하므로 클라이언트에 직접 노출되지 않습니다.
- 다양한 애플리케이션의 요구 사항에 맞게 특별히 구축된 API를 구축할 수 있으므로 프런트엔드에서 사용자 지정 비즈니스 로직이 필요하지 않습니다. 이는 Backend-For-Frontend(BFF) 패턴과 일치합니다.
- 다양한 권한 부여 모드를 사용하여 AWS AppSync 계층에서 권한 부여 및 액세스 제어를 구현하여 액세스를 제어할 수 있습니다. 웹 서버 호스팅 또는 프록시 연결과 같이 데이터베이스에 연결하는 데 추가 컴퓨팅 리소스가 필요하지 않습니다.
- 구독을 통해 실시간 기능을 추가할 수 있으며, AppSync를 통해 이루어진 데이터 변경은 연결된 클라이언트에 자동으로 푸시됩니다.
- 클라이언트는 443과 같은 일반 포트를 사용하여 HTTPS를 통해 API에 연결할 수 있습니다.

AWS AppSync는 기존 관계형 데이터베이스에서 API를 쉽게 구축할 수 있습니다. 내부 검사 유틸리티는 데이터베이스 테이블에서 모델을 검색하고 GraphQL 유형을 제안할 수 있습니다. AWS AppSync 콘솔의 API 생성 마법사는 Aurora MySQL 또는 PostgreSQL 데이터베이스에서 API를 즉시 생성할 수 있습니다. 데이터를 읽고 쓸 수 있는 유형과 JavaScript 해석기를 자동으로 생성합니다.

AWS AppSync는 해석기에서 SQL 문 작성을 단순화하는 통합 JavaScript 유틸리티를 제공합니다. 동적 값이 있는 정적 문에는 AWS AppSync의 `sql` 태그 템플릿을 사용하거나 프로그래밍 방식으로 문을 작성하는 데 `rds` 모듈 유틸리티를 사용할 수 있습니다. 자세한 내용은 RDS용 해석기 함수 참조 데이터 소스 및 내장 모듈 섹션을 참조하십시오.

내부 검사 기능 사용(콘솔)

자세한 자습서 및 시작하기 안내서는 자습서: 데이터 API를 사용하는 Aurora PostgreSQL Serverless 섹션을 참조하십시오.

AWS AppSync 콘솔을 사용하면 데이터 API로 구성된 기존 Aurora 데이터베이스에서 단 몇 분 만에 AWS AppSync GraphQL API를 생성할 수 있습니다. 그러면 데이터베이스 구성을 기반으로 운영 스키마가 빠르게 생성됩니다. API를 있는 그대로 사용하거나 API를 기반으로 기능을 추가할 수 있습니다.

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.

   - 대시보드에서 API 생성을 선택합니다.

2. API 옵션에서 GraphQL API를 선택하고 Amazon Aurora 클러스터로 시작을 누른 다음 다음을 선택합니다.

   a. API 이름을 입력합니다. 이는 콘솔에서 API의 식별자로 사용됩니다.

   b. 연락처 세부 정보에는 API 관리자를 식별할 연락처를 입력하면 됩니다. 이 필드는 선택 사항입니다.

   c. 프라이빗 API 구성에서 프라이빗 API 기능을 활성화할 수 있습니다. 프라이빗 API는 구성된 VPC 엔드포인트(VPCE)에서만 액세스할 수 있습니다. 자세한 내용은 프라이빗 API를 참조하세요.

      이 예에서는 이 기능을 활성화하지 않는 것이 좋습니다. 입력 내용을 검토한 후 다음을 선택합니다.

3. 데이터베이스 페이지에서 데이터베이스 선택을 선택합니다.

   a. 클러스터에서 데이터베이스를 선택해야 합니다. 첫 번째 단계는 클러스터가 있는 리전을 선택하는 것입니다.

   b. 드롭다운 목록에서 Aurora 클러스터를 선택합니다. 리소스를 사용하기 전에 해당 데이터 API를 생성하고 활성화해야 합니다.

   c. 다음으로 데이터베이스에 대한 보안 인증을 서비스에 추가해야 합니다. 이는 주로 AWS Secrets Manager를 사용하여 수행됩니다. 암호가 존재하는 리전을 선택합니다. 암호 정보를 검색하는 방법에 대한 자세한 내용은 보안 암호 찾기 또는 보안 암호 검색을 참조하십시오.

d. 드롭다운 목록에서 암호를 추가합니다. 단, 사용자에게 데이터베이스에 대한 읽기 권한이 있어야 합니다.

4. 가져오기를 선택합니다.

AWS AppSync는 데이터베이스 검사를 시작하여 테이블, 열, 프라이머리 키, 색인을 검색합니다. 검색된 테이블이 GraphQL API에서 지원될 수 있는지 확인합니다. 새 행 생성을 지원하려면 테이블에 여러 열을 사용할 수 있는 프라이머리 키가 필요합니다. AWS AppSync가 다음과 같이 테이블 열을 유형 필드에 매핑합니다.

데이터 유형	필드 유형
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. 테이블 검색이 완료되면 데이터베이스 섹션에 사용자 정보가 채워집니다. 새 데이터베이스 테이블 섹션에서 테이블의 데이터가 이미 채워져 스키마의 유형으로 변환되었을 수 있습니다. 필수 데이터 중 일부가 보이지 않는 경우 테이블 추가를 선택하고 나타나는 모달에서 해당 유형의 확인란을 클릭한 다음 추가를 선택하여 해당 데이터를 확인할 수 있습니다.

   데이터베이스 테이블 섹션에서 유형을 제거하려면 제거하려는 유형 옆의 확인란을 클릭한 다음 제거를 선택합니다. 제거된 유형은 나중에 다시 추가하려는 경우 테이블 추가 모달에 배치됩니다.

참고로 AWS AppSync는 테이블 이름을 유형 이름으로 사용하지만 이름을 바꿀 수 있습니다. 예를 들어 *movies*와 같은 복수형 테이블 이름을 *Movie*라는 유형 이름으로 변경할 수 있습니다. 데이터베이스 테이블 섹션에서 유형의 이름을 바꾸려면 이름을 바꾸려는 유형의 확인란을 클릭한 다음 유형 이름 열에서 연필 아이콘을 클릭합니다.

선택에 따라 스키마의 내용을 미리 보려면 스키마 미리 보기를 선택합니다. 이 스키마는 비워 둘 수 없으므로 한 개 이상의 테이블을 유형으로 변환해야 합니다. 또한 이 스키마의 크기는 1MB를 초과할 수 없습니다.

- 서비스 역할에서 이 가져오기를 위한 새 서비스 역할을 만들지 아니면 기존 역할을 사용할지를 선택합니다.

6. 다음을 선택합니다.
7. 그런 다음 읽기 전용 API(쿼리만 해당)를 만들지 아니면 데이터 읽기 및 쓰기를 위한 API(쿼리 및 변형 해당)를 만들지 선택합니다. 후자는 변형에 의해 트리거되는 실시간 구독도 지원합니다.
8. 다음을 선택합니다.
9. 선택 사항을 검토한 다음 API 생성을 선택합니다. AWS AppSync는 API를 생성하고 해석기를 쿼리와 변형에 연결합니다. 생성된 API는 완벽하게 작동하며 필요에 따라 확장할 수 있습니다.

내부 검사 기능 사용(API)

`StartDataSourceIntrospection` 내부 검사 API를 사용하여 데이터베이스에서 프로그래밍 방식으로 모델을 검색할 수 있습니다. 명령에 대한 자세한 내용은 `StartDataSourceIntrospection` API 사용 섹션을 참조하십시오.

`StartDataSourceIntrospection`을 사용하려면 Aurora 클러스터 Amazon 리소스 이름(ARN), 데이터베이스 이름 및 AWS Secrets Manager 암호 ARN을 입력합니다. 이 명령은 내부 검사 프로세스를 시작합니다. `GetDataSourceIntrospection` 명령을 사용하여 결과를 검색할 수 있습니다. 검색된 모델의 스토리지 정의 언어(SDL) 문자열을 명령으로 반환할지 여부를 지정할 수 있습니다. 이는 검색된 모델에서 직접 SDL 스키마 정의를 생성하는 데 유용합니다.

예를 들어, 단순 `Todos` 테이블에 다음과 같은 데이터 정의 언어(DDL) 문이 있는 경우

```
create table if not exists public.todos
(
id serial constraint todos_pk primary key,
description text,
due timestamp,
```

```
"createdAt" timestamp default now()
);
```

내부 검사는 다음과 같이 시작합니다.

```
aws appsync start-data-source-introspection \
  --rds-data-api-config resourceArn=<cluster-arn>,secretArn=<secret-
arn>,databaseName=database
```

그런 다음 GetDataSourceIntrospection 명령을 사용하여 결과를 검색합니다.

```
aws appsync get-data-source-introspection \
  --introspection-id a1234567-8910-abcd-efgh-identifier \
  --include-models-sdl
```

그 결과 다음 결과를 반환합니다.

```
{
    "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
    }
```

```
}
```

클라이언트 애플리케이션 빌드

모든 AWS AppSync GraphQL 클라이언트를 사용하여 GraphQL API에 연결할 수 있지만 Amplify 클라이언트를 사용하는 것이 좋습니다. Amplify는 GraphQL API를 위한 강력한 형식의 클라이언트 SDK를 자동 생성할 뿐만 아니라 클라이언트 애플리케이션에서 실시간 데이터 및 향상된 GraphQL 쿼리 기능을 지원합니다. 웹 애플리케이션의 경우 Amplify는 클라이언트를 JavaScript 생성할 수 있습니다. 교차 플랫폼 또는 모바일 환경을 대상으로 하는 애플리케이션을 위해 Amplify는 Android, iOS 및 React Native를 지원합니다. 이러한 플랫폼의 클라이언트 코드 생성에 대해 더 알아보려면 Amplify 설명서를 참조하세요. 다음은 React 애플리케이션으로 여정을 시작하는 데 도움이 되는 가이드입니다. JavaScript

> **Note**
>
> 시작하기 전에 npm과 Amazon CLI를 모두 설치하고 구성해야 합니다. Amplify v6 클라이언트를 사용하는 경우 이 가이드를 따르십시오.

시작하려면

1. 로컬 컴퓨터에서 프로젝트 디렉터리로 이동합니다. 아래 명령을 사용하여 Amplify 라이브러리를 설치합니다.

   ```
   npm install aws-amplify
   ```

2. 구성 파일을 다운로드하여 프로젝트 폴더에 저장합니다. 구성 파일에는 일반적으로 일부 설정 (엔드포인트, 지역, 권한 부여 모드 등) 이 정의된 `config` 변수가 포함됩니다. 예를 들어 다음과 같이 보일 수 있습니다.

   ```
   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. 코드에서 Amplify 라이브러리 및 Amplify를 설정하기 위한 구성을 가져와 다음과 같이 하십시오.

```
import { Amplify } from 'aws-amplify';
import config from './aws-exports.js';

Amplify.configure(config);
```

또는 API 구성의 스니펫을 사용하여 Amplify를 직접 설정할 수도 있습니다.

```
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. Amplify 도구 체인을 사용하면 스키마를 기반으로 작업을 자동 생성할 수 있으므로 수동 스크립팅에 드는 수고를 덜 수 있습니다. 애플리케이션의 루트 디렉터리에서 다음 CLI 명령을 사용합니다.

```
npx @aws-amplify/cli codegen add --apiId <id goes here> --region <region goes here>
```

그러면 API의 스키마가 다운로드되고 기본적으로 폴더에 클라이언트 헬퍼 코드가 생성됩니다. `src/graphql` API를 배포할 때마다 다음 명령을 다시 실행하여 업데이트된 GraphQL 문 및 유형을 생성할 수 있습니다.

```
npx @aws-amplify/cli codegen
```

5. 이제 안드로이드, 스위프트, 플러터 등을 위한 모델을 생성할 수 있습니다. JavaScript DataStore 다음 명령어를 사용하여 스키마를 다운로드하세요.

```
aws appsync get-introspection-schema --api-id <id goes here> --region <region goes
 here> --format SDL schema.graphql
```

그런 다음 애플리케이션의 루트 디렉터리에서 다음 명령을 실행합니다.

```
npx @aws-amplify/cli codegen models \
--model-schema schema.graphql \
--target [android|ios|flutter|javascript|typescript] \
--output-dir ./
```

해석기 튜토리얼(JavaScript)

데이터 소스 및 해석기는 AWS AppSync가 GraphQL 요청을 변환하고 AWS 리소스에서 정보를 가져오는 방법입니다. AWS AppSync는 자동 프로비저닝과 특정 데이터 소스 유형과의 연결을 지원합니다. AWS AppSync는 AWS Lambda, Amazon DynamoDB, 관계형 데이터베이스(Amazon Aurora Serverless), Amazon OpenSearch Service, HTTP 엔드포인트와 같은 데이터 소스를 지원합니다. 기존 AWS 리소스와 함께 GraphQL API를 사용하거나 데이터 소스 및 해석기를 빌드할 수 있습니다. 이 섹션은 작업 세부 사항과 튜닝 옵션을 더 잘 이해할 수 있도록 일련의 자습서에서 이 프로세스를 안내합니다.

주제

- 자습서: DynamoDB JavaScript 해석기
- 자습서: Lambda 해석기
- 자습서: 로컬 해석기
- 튜토리얼: GraphQL 해석기 결합
- 자습서: Amazon OpenSearch Service 해석기
- 자습서: DynamoDB 트랜잭션 해석기
- 자습서: DynamoDB 배치 해석기
- 자습서: HTTP 해석기
- 자습서: 데이터 API를 사용하는 Aurora PostgreSQL

자습서: DynamoDB JavaScript 해석기

이 튜토리얼에서는 Amazon DynamoDB 테이블을 AWS AppSync로 가져오고 자체 애플리케이션에서 활용할 수 있는 JavaScript 파이프라인 해석기를 사용하여 모든 기능을 갖춘 GraphQL API를 구축하도록 연결합니다.

AWS AppSync 콘솔을 사용하여 Amazon DynamoDB 리소스를 프로비저닝하고, 해석기를 생성하고, 이를 데이터 소스에 연결합니다. 또한 GraphQL 명령문을 통해 Amazon DynamoDB 데이터베이스를 읽고 쓸 수 있으며 실시간 데이터를 구독할 수 있습니다.

GraphQL 명령문을 Amazon DynamoDB 작업으로 변환하고 응답을 다시 GraphQL로 변환하기 위해서는 특정 단계를 완료해야 합니다. 이 자습서에서는 여러 가지 실제 시나리오와 데이터 액세스 패턴을 통해 구성 프로세스를 설명합니다.

GraphQL API 생성

AWS AppSync에서 GraphQL API를 생성하려면

1. AppSync 콘솔을 열고 API 생성을 선택합니다.
2. 처음부터 디자인을 선택하고 다음을 선택합니다.
3. API `PostTutorialAPI`의 이름을 지정한 후 다음을 선택합니다. 나머지 옵션은 기본값으로 설정한 채 검토 페이지로 건너뛰고 `Create`를 선택합니다.

AWS AppSync 콘솔이 새로운 GraphQL API를 생성합니다. 기본적으로 API 키 인증 모드를 사용합니다. 콘솔에서 GraphQL API의 나머지 부분을 설정하고 이 자습서의 나머지 부분에서 이 API에 대한 쿼리를 실행할 수 있습니다.

기본 Post API 정의

GraphQL API가 있으므로 게시 데이터의 기본적인 생성, 검색 및 삭제를 허용하는 기본 스키마를 설정할 수 있습니다.

스키마에 데이터를 추가하려면

1. API에서 스키마 탭을 선택합니다.
2. `Post` 객체를 추가하고 가져오기 위한 `Post` 유형과 작업 `addPost`를 정의하는 스키마를 만들 것입니다. 스키마 창에서 파일 콘텐츠를 다음 코드로 바꿉니다.

```
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. 스키마 저장(Save Schema)을 선택합니다.

Amazon DynamoDB 테이블 설정

AWS AppSync 콘솔은 Amazon DynamoDB 테이블에 자체 리소스를 저장하는 데 필요한 AWS 리소스를 프로비저닝하는 데 도움이 될 수 있습니다. 이 단계에서는 Amazon DynamoDB 테이블을 생성하여 게시물을 저장합니다. 또한 나중에 사용할 보조 인덱스도 설정합니다.

Amazon DynamoDB 테이블을 생성하려면

1. 스키마 페이지에서 리소스 생성을 선택합니다.
2. 기존 유형 사용을 선택하고 `Post` 유형을 선택합니다.
3. 보조 인덱스 섹션에서 인덱스 추가를 선택합니다.
4. 인덱스 이름을 `author-index`로 지정합니다.
5. `Primary key`를 `author`로 설정하고 `Sort` 키를 `None`으로 설정합니다.
6. GraphQL 자동 생성을 비활성화합니다. 이 예에서는 해석기를 직접 생성합니다.
7. 생성을 선택합니다.

이제 `PostTable`이라는 새 데이터 소스가 생겼습니다. 사이드 탭의 데이터 소스를 방문하면 이를 확인할 수 있습니다. 이 데이터 소스를 사용하여 쿼리와 뮤테이션을 Amazon DynamoDB 테이블에 연결합니다.

addPost 해석기 설정(Amazon DynamoDB PutItem)

AWS AppSync에서 Amazon DynamoDB 테이블을 인식하면 해석기를 정의해 이 테이블을 개별 쿼리 및 뮤테이션에 연결할 수 있습니다. 생성한 첫 번째 해석기는 JavaScript를 사용하는 `addPost` 파이프라인 해석기로, 이 해석기를 사용하면 Amazon DynamoDB 테이블에 게시물을 작성할 수 있습니다. 파이프라인 해석기에는 다음 구성 요소가 있습니다.

- 해석기를 연결할 GraphQL 스키마의 위치. 이 경우 `createPost` 형식의 `Mutation` 필드에서 해석기를 설정합니다. 이 해석기는 호출자가 뮤테이션 `{ addPost(...){...} }`를 호출하면 간접적으로 호출됩니다.
- 이 해석기에 사용할 데이터 원본. 이 경우 앞서 정의한 DynamoDB 데이터 소스를 사용하려고 합니다. 따라서 `post-table-for-tutorial` DynamoDB 테이블에 항목을 추가할 수 있습니다.
- 요청 핸들러. 요청 핸들러는 호출자로부터 들어오는 요청을 처리하여 AWS AppSync가 DynamoDB에 대해 수행할 명령으로 변환하는 함수입니다.
- 응답 핸들러. 응답 핸들러의 작업은 DynamoDB의 응답을 처리하여 GraphQL에 필요한 것으로 변환하는 것입니다. 이러한 템플릿은 DynamoDB의 데이터 모양과 GraphQL의 `Post` 형식이 다른 경우 유용하지만 이 경우 모양이 서로 동일하기 때문에 데이터를 그냥 전달하면 됩니다.

해석기를 설정하려면

1. API에서 스키마 탭을 선택합니다.
2. 해석기 창에서 `Mutation` 유형 아래의 `addPost` 필드를 찾은 다음 연결을 선택합니다.
3. 데이터 소스를 선택한 다음 생성을 선택합니다.
4. 코드 편집기에서 코드를 다음 코드 조각으로 바꿉니다.

```
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. Save를 선택합니다.

> **Note**
>
> 이 코드에서는 DynamoDB 요청을 쉽게 생성할 수 있는 DynamoDB 모듈 유틸리티를 사용합니다.

AWS AppSync에는 `util.autoId()`라는 자동 ID 생성 유틸리티가 함께 제공되며, 이 유틸리티는 새 게시물의 ID를 생성하는 데 사용됩니다. ID를 지정하지 않으면 유틸리티에서 ID를 자동으로 생성합니다.

```
const key = { id: ctx.args.id ?? util.autoId() }
```

JavaScript에 사용할 수 있는 유틸리티에 대한 자세한 내용은 해석기 및 함수를 위한 JavaScript 런타임 기능을 참조하세요.

게시물 추가를 위한 API 호출

이제 해석기가 구성되었으므로 AWS AppSync는 들어오는 `addPost` 뮤테이션을 Amazon DynamoDB `PutItem` 작업으로 변환할 수 있습니다. 이제 변형을 실행해 테이블에 데이터를 입력할 수 있습니다.

작업을 실행하려면

1. API에서 쿼리 탭을 선택합니다.
2. 쿼리 창에 다음 뮤테이션을 추가합니다.

```
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. 실행(주황색 재생 버튼)을 선택한 다음 addPost를 선택합니다. 새로 생성한 게시물의 결과가 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

다음 설명은 어떤 일이 발생했는지 보여줍니다.

1. AWS AppSync에서 addPost 변형 요청을 수신했습니다.
2. AWS AppSync는 해석기의 요청 핸들러를 실행합니다. ddb.put 함수는 다음과 같은 PutItem 요청을 생성합니다.

```
{
  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 AppSync는 이 값을 사용하여 Amazon DynamoDB `PutItem` 요청을 생성하고 실행합니다.
4. AWS AppSync에서 `PutItem` 요청의 결과를 받아 다시 GraphQL 형식으로 변환했습니다.

```
{
    "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. 응답 핸들러는 결과를 즉시 반환합니다(`return ctx.result`).
6. 최종 결과는 GraphQL 응답에서 확인할 수 있습니다.

getPost 해석기 설정(Amazon DynamoDB GetItem)

이제 Amazon DynamoDB 테이블에 데이터를 추가할 수 있으므로 DynamoDB 테이블에서 데이터를 검색할 수 있도록 쿼리를 설정해야 합니다. 이렇게 하기 위해 다른 해석기를 설정합니다.

해석기를 추가하려면

1. API에서 스키마 탭을 선택합니다.
2. 오른쪽의 해석기 창에서 `Query` 유형 아래의 `getPost` 필드를 찾은 다음 연결을 선택합니다.
3. 데이터 소스를 선택한 다음 생성을 선택합니다.
4. 코드 편집기에서 코드를 다음 코드 조각으로 바꿉니다.

```
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. 해석기를 저장합니다.

> **Note**
>
> 이 해석기에서는 응답 핸들러에 화살표 함수 표현식을 사용합니다.

게시물 가져오기를 위한 API 호출

이제 해석기가 설정되었으므로 AWS AppSync는 들어오는 `getPost` 쿼리를 Amazon DynamoDB `GetItem` 작업으로 변환하는 방법을 알고 있습니다. 이제 앞에서 생성한 게시물을 검색하는 쿼리를 실행할 수 있습니다.

쿼리를 실행하려면

1. API에서 쿼리 탭을 선택합니다.
2. 쿼리 창에서 다음 코드를 추가하고 게시물을 만든 후 복사한 ID를 사용합니다.

```
query getPost {
  getPost(id: "123") {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. 실행(주황색 재생 버튼)을 선택한 다음 `getPost`를 선택합니다. 새로 생성한 게시물의 결과가 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다.
4. Amazon DynamoDB에서 가져온 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

또는 다음 예를 사용합니다.

```
query getPost {
  getPost(id: "123") {
    id
    author
    title
  }
}
```

getPost 쿼리에 id, author, title만 필요한 경우, 요청 함수를 변경하여 프로젝션 표현식을 사용하여 DynamoDB 테이블에서 원하는 속성만 지정하면 DynamoDB에서 AWS AppSync로 불필요한 데이터 전송을 방지할 수 있습니다. 예를 들어 요청 함수는 아래 코드 조각과 비슷할 수 있습니다.

```
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
```

getPost와 함께 selectionSetList를 사용하여 expression을 나타낼 수도 있습니다.

```
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
```

updatePost 뮤테이션 생성(Amazon DynamoDB UpdateItem)

이제 Amazon DynamoDB에서 `Post` 객체를 생성하고 검색할 수 있습니다. 다음으로, 객체 업데이트를 위한 새 변형을 설정합니다. 모든 필드를 지정해야 하는 `addPost` 뮤테이션에 비해 이 뮤테이션을 사용하면 변경하려는 필드만 지정할 수 있습니다. 또한 수정하려는 버전을 지정할 수 있는 새 `expectedVersion` 인수도 도입되었습니다. 객체의 최신 버전을 수정하고 있는지 확인하는 조건을 설정해야 합니다. 이 작업은 `UpdateItem` Amazon DynamoDB operation.sc를 사용하여 수행합니다.

해석기를 업데이트하려면

1. API에서 스키마 탭을 선택합니다.
2. 스키마 창에서 다음과 같이 `Mutation` 형식을 수정하여 새로운 `updatePost` 변형을 추가할 수 있습니다.

```
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. 스키마 저장(Save Schema)을 선택합니다.

4. 오른쪽의 해석기 창에서 Mutation 유형에 새로 생성된 updatePost 필드를 찾은 다음 연결을 선택합니다. 아래 코드 조각을 사용하여 새 해석기를 만듭니다.

```
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. 변경한 내용을 모두 저장합니다.

이 해석기는 ddb.update를 사용하여 Amazon DynamoDB UpdateItem 요청을 생성합니다. 전체 항목을 쓰는 대신 특정 속성을 업데이트하도록 Amazon DynamoDB에 요청합니다. Amazon DynamoDB 업데이트 표현식을 사용하여 요청합니다.

ddb.update 함수는 키와 업데이트 객체를 인수로 사용합니다. 그런 다음 들어오는 인수의 값을 확인합니다. 값이 null로 설정되면 DynamoDB remove 작업을 사용하여 DynamoDB 항목에서 값을 제거해야 한다는 신호를 보냅니다.

또한 새로운 condition 섹션이 있습니다. 조건 표현식을 사용하면 작업을 수행하기 전에 Amazon DynamoDB에 이미 있는 객체의 상태를 기반으로 AWS AppSync 및 Amazon DynamoDB에 요청에 성공해야 할지 여부를 지시할 수 있습니다. 이 경우에는 Amazon DynamoDB에 현재 있는 항목의 version 필드가 expectedVersion 인수와 정확하게 일치하는 경우에만 UpdateItem 요청에 성공

하면 됩니다. 항목이 업데이트되면 `version`의 값을 증가시키겠습니다. 작업 기능 `increment`를 사용하면 이 작업을 쉽게 수행할 수 있습니다.

조건 표현식에 대한 자세한 내용은 조건 표현식 설명서를 참조하세요.

`UpdateItem` 요청에 대한 자세한 내용은 UpdateItem 설명서 및 DynamoDB 모듈 설명서를 참조하세요.

업데이트 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB UpdateExpressions 문서를 참조하십시오.

게시물 업데이트를 위한 API 호출

새 해석기를 사용해 `Post` 객체를 업데이트해 보겠습니다.

객체를 업데이트하려면

1. API에서 쿼리 탭을 선택합니다.
2. 쿼리 창에 다음 뮤테이션을 추가합니다. 또한 앞서 적어둔 값을 갖도록 `id` 인수를 업데이트해야 합니다.

```
mutation updatePost {
  updatePost(
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. 실행(주황색 재생 버튼)을 선택한 다음 `updatePost`를 선택합니다.
4. Amazon DynamoDB에서 업데이트된 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

이 요청에서는 AWS AppSync 및 Amazon DynamoDB에 `title` 및 `content` 필드만 업데이트하도록 요청했습니다. `version` 필드를 증분하는 것을 제외하고 다른 모든 필드는 그대로 둡니다. `title` 속성을 새 값으로 설정하고 게시물에서 `content` 속성을 제거했습니다. `author`, `url`, `ups` 및 `downs` 필드는 그대로 남아 있습니다. 뮤테이션 요청을 다시 실행해도 요청은 정확하게 그대로 유지됩니다. 다음과 유사한 응답이 나타납니다.

```
{
  "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)"
    }
  ]
}
```

조건 표현식이 `false`로 평가되었으므로 요청에 실패합니다.

1. 요청을 처음 실행했을 때 Amazon DynamoDB에 있는 게시물의 `version` 필드 값은 1이었고, 이는 `expectedVersion` 인수와 일치했습니다. 요청에 성공하자 Amazon DynamoDB의 `version` 필드가 2로 증가했습니다.
2. 요청을 두 번째로 실행했을 때 Amazon DynamoDB에 있는 게시물의 `version` 필드 값은 2이었고, 이는 `expectedVersion` 인수와 일치하지 않았습니다.

이러한 패턴을 일반적으로 낙관적 잠금이라고 합니다.

투표 뮤테이션 생성(Amazon DynamoDB UpdateItem)

`Post` 유형에는 추천과 비추천을 기록할 수 있는 `ups` 및 `downs` 필드가 포함되어 있습니다. 하지만 현재로서는 API를 통해 아무것도 할 수 없습니다. 게시물을 추천 및 비추천하도록 하는 뮤테이션을 추가해 보겠습니다.

뮤테이션을 추가하려면

1. API에서 스키마 탭을 선택합니다.
2. 스키마 창에서 `Mutation` 유형을 수정하고 `DIRECTION` 열거형을 추가하여 새 투표 뮤테이션을 추가합니다.

```
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. 스키마 저장(Save Schema)을 선택합니다.
4. 오른쪽의 해석기 창에서 Mutation 유형에 새로 생성된 vote 필드를 찾은 다음 연결을 선택합니다. 코드를 생성하고 다음 코드 조각으로 대체하여 새 해석기를 만듭니다.

```
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. 변경한 내용을 모두 저장합니다.

게시물을 추천 또는 비추천하는 API 호출

이제 새 해석기가 설정되었으므로 AWS AppSync는 들어오는 upvotePost 쿼리 또는 downvote 뮤테이션을 Amazon DynamoDB UpdateItem 작업으로 변환하는 방법을 알고 있습니다. 이제 앞서 생성한 게시물을 추천 또는 비추천하는 변형을 실행할 수 있습니다.

뮤테이션을 추가하려면

1. API에서 쿼리 탭을 선택합니다.

2. 쿼리 창에 다음 뮤테이션을 추가합니다. 또한 앞서 적어둔 값을 갖도록 id 인수를 업데이트해야 합니다.

```
mutation votePost {
  vote(id:123, direction: UP) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. 실행(주황색 재생 버튼)을 선택한 다음 votePost를 선택합니다.
4. Amazon DynamoDB에서 업데이트된 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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. 실행을 몇 번 더 선택합니다. 쿼리를 실행할 때마다 ups 및 version 필드가 1씩 증가하는 것이 보여야 합니다.
6. 쿼리를 변경하여 다른 DIRECTION으로 호출합니다.

```
mutation votePost {
  vote(id:123, direction: DOWN) {
    id
```

```
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

7. 실행(주황색 재생 버튼)을 선택한 다음 votePost를 선택합니다.

   이번에는 쿼리를 실행할 때마다 downs 및 version 필드가 1씩 증가하는 것이 보여야 합니다.

deletePost 해석기 설정(Amazon DynamoDB DeleteItem)

다음으로 게시물을 삭제하기 위한 뮤테이션을 생성해야 합니다. 이 작업은 DeleteItem Amazon DynamoDB 작업을 사용하여 수행합니다.

뮤테이션을 추가하려면

1. 스키마에서 스키마 탭을 선택합니다.
2. 스키마 창에서 다음과 같이 Mutation 유형을 수정하여 새로운 deletePost 뮤테이션을 추가합니다.

```
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. 이번에는 expectedVersion 필드를 선택 사항으로 설정했습니다. 다음으로 스키마 저장을 선택합니다.
4. 오른쪽의 해석기 창에서 Mutation 유형에 새로 생성된 delete 필드를 찾은 다음 연결을 선택합니다. 다음 코드를 사용하여 새 해석기를 생성합니다.

```
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**
>
> 참고: expectedVersion 인수는 선택적 인수입니다. 호출자가 요청에서 expectedVersion 인수를 설정하면, 요청 핸들러에서는 항목이 이미 삭제된 경우 또는 Amazon DynamoDB에 있는 version 속성이 expectedVersion과 정확하게 일치하는 경우, DeleteItem 요청이 완료되게 하는 조건을 추가합니다. 그냥 두면 DeleteItem 요청

에 아무런 조건 표현식도 지정되지 않습니다. 이는 `version`의 값이나 해당 항목이 Amazon DynamoDB에 존재하는지 여부에 관계없이 완료됩니다.
항목을 삭제하고 있더라도 항목이 아직 삭제되지 않은 경우 삭제된 항목을 반환할 수 있습니다.

`DeleteItem` 요청에 대한 자세한 내용은 DeleteItem 설명서를 참조하세요.

게시물 삭제를 위한 API 호출

이제 해석기가 설정되었으므로 AWS AppSync는 들어오는 `delete` 뮤테이션을 Amazon DynamoDB `DeleteItem` 작업으로 변환하는 방법을 알고 있습니다. 이제 변형을 실행해 테이블에 데이터를 삭제할 수 있습니다.

뮤테이션을 추가하려면

1. API에서 쿼리 탭을 선택합니다.
2. 쿼리 창에 다음 뮤테이션을 추가합니다. 또한 앞서 적어둔 값을 갖도록 `id` 인수를 업데이트해야 합니다.

```
mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. 실행(주황색 재생 버튼)을 선택한 다음 `deletePost`를 선택합니다.
4. Amazon DynamoDB에서 해당 게시물이 삭제됩니다. AWS AppSync에서는 Amazon DynamoDB에서 삭제된 항목의 값을 반환합니다. 해당 값은 쿼리 창 오른쪽에 있는 결과 창에 표시되어야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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. deletePost에 대한 호출이 Amazon DynamoDB에서 실제로 삭제된 항목인 경우에만 값이 반환됩니다. 실행을 다시 선택합니다.
6. 호출에는 계속 성공하지만 반환되는 값은 없습니다.

```
{
  "data": {
    "deletePost": null
  }
}
```

7. 이제 게시물을 삭제해 보는데, 이번에는 expectedValue를 지정해 보겠습니다. 지금까지 작업한 게시물 하나를 방금 삭제했기 때문에 먼저 새 게시물을 생성해야 합니다.
8. 쿼리 창에 다음 뮤테이션을 추가합니다.

```
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. 실행(주황색 재생 버튼)을 선택한 다음 addPost를 선택합니다.
10. 새로 생성한 게시물의 결과가 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 곧 필요할 것이기 때문에 새로 생성한 객체의 id를 기록해 둡니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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. 이제 expectedVersion의 값이 잘못된 게시물을 삭제해 보겠습니다. 쿼리 창에 다음 뮤테이션을 추가합니다. 또한 앞서 적어둔 값을 갖도록 id 인수를 업데이트해야 합니다.

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

12. 실행(주황색 재생 버튼)을 선택한 다음 deletePost를 선택합니다. 다음 결과가 반환됩니다.

```
{
  "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)"
    }
  ]
}
```

13조건 표현식이 `false`로 평가되었으므로 요청에 실패합니다. Amazon DynamoDB에 있는 게시물의 `version` 값이 인수에 지정된 `expectedValue`와 일치하지 않습니다. 객체의 현재 값은 GraphQL 응답에서 `data` 섹션의 `errors` 필드에 반환됩니다. 이 요청을 다시 실행하되 `expectedVersion`을 수정합니다.

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
```

```
    version
  }
}
```

14.실행(주황색 재생 버튼)을 선택한 다음 deletePost를 선택합니다.

이번에는 요청에 성공하고 Amazon DynamoDB에서 삭제된 값이 반환됩니다.

```
{
  "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.실행을 다시 선택합니다. 호출에 계속해서 성공하지만 이번에는 게시물이 이미 Amazon DynamoDB에서 삭제되었으므로 반환되는 값이 없습니다.

```
{ "data": { "deletePost": null } }
```

allPost 해석기 설정(Amazon DynamoDB 스캔)

지금까지 API는 살펴보고자 하는 각 게시물의 id를 아는 경우에만 유용했습니다. 테이블의 게시물을 모두 반환하는 새 해석기를 추가해 보겠습니다.

뮤테이션을 추가하려면

1. API에서 스키마 탭을 선택합니다.
2. 스키마 창에서 다음과 같이 Query 형식을 수정하여 새로운 allPost 쿼리를 추가할 수 있습니다.

```
type Query {
    allPost(limit: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
```

```
}
```

3. 새로운 PaginationPosts 형식을 다음과 같이 추가합니다.

```
type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}
```

4. 스키마 저장(Save Schema)을 선택합니다.
5. 오른쪽의 해석기 창에서 Query 유형에 새로 생성된 allPost 필드를 찾은 다음 연결을 선택합니다. 다음 코드를 사용하여 새 해석기를 생성합니다.

```
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 };
}
```

   이 해석기의 요청 핸들러에는 두 개의 선택적 인수가 필요합니다.

   - limit - 단일 호출에서 반환할 최대 항목 수를 지정합니다.
   - nextToken - 다음 결과 세트를 검색하는 데 사용됩니다(nextToken 값의 출처는 나중에 설명).
6. 해석기의 모든 변경 사항을 저장합니다.

Scan 요청에 대한 자세한 내용은 스캔 참조 문서를 참조하세요.

모든 게시물을 스캔하는 API 호출

이제 해석기가 설정되었으므로 AWS AppSync는 들어오는 allPost 쿼리를 Amazon DynamoDB Scan 작업으로 변환하는 방법을 알고 있습니다. 이제 테이블을 스캔해 게시물을 모두 검색할 수 있습니다. 지금까지 작업했던 데이터를 모두 삭제했기 때문에 이 요청을 수행하기 전에 몇 가지 데이터로 테이블을 채워야 합니다.

데이터를 추가 및 쿼리하려면

1. API에서 쿼리 탭을 선택합니다.
2. 쿼리 창에 다음 뮤테이션을 추가합니다.

```
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. 실행(주황색 재생 버튼)을 선택합니다.
4. 이제 테이블을 스캔해 보겠습니다. 한 번에 5개 결과를 반환합니다. 쿼리 창에 다음 쿼리를 추가합니다.

```
query allPost {
  allPost(limit: 5) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

5. 실행(주황색 재생 버튼)을 선택한 다음 `allPost`를 선택합니다.

   처음 5개 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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. 5개 결과가 반환되고 다음 결과 집합을 가져오는데 사용할 수 있는 nextToken도 있습니다. 이전 결과 집합에서 allPost을 포함하도록 nextToken 쿼리를 업데이트합니다.

```
query allPost {
  allPost(
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
      author
    }
```

```
    nextToken
  }
}
```

7. 실행(주황색 재생 버튼)을 선택한 다음 allPost를 선택합니다.

나머지 4개 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 게시물이 모두 9개이며 남은 게시물이 없기 때문에 이 결과 집합에는 nextToken이 없습니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

allPostsByAuthor 해석기 설정(Amazon DynamoDB 쿼리)

Amazon DynamoDB에서 모든 게시물을 스캔하는 것 이외에 Amazon DynamoDB를 쿼리해 특정 작성자가 생성한 게시물을 검색할 수 있습니다. 앞서 이미 생성한 Amazon DynamoDB 테이블에는

특정 작성자가 생성한 게시물을 모두 검색하는 Query 작업에 사용할 수 있는 author-index라는 GlobalSecondaryIndex가 있습니다.

쿼리를 추가하려면

1. API에서 스키마 탭을 선택합니다.
2. 스키마 창에서 다음과 같이 Query 형식을 수정하여 새로운 allPostsByAuthor 쿼리를 추가할 수 있습니다.

```
type Query {
    allPostsByAuthor(author: String!, limit: Int, nextToken: String): PaginatedPosts!
    allPost(limit: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

   참고: 여기서는 allPost 쿼리에 사용한 것과 동일한 PaginatedPosts 유형을 사용합니다.

3. 스키마 저장(Save Schema)을 선택합니다.
4. 오른쪽의 해석기 창에서 Query 유형에 새로 생성된 allPostsByAuthor 필드를 찾은 다음 연결을 선택합니다. 아래 코드 조각을 사용하여 해석기를 만듭니다.

```
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 };
}
```

   allPost 해석기와 마찬가지로 이 해석기에는 두 개의 선택적 인수가 있습니다.

- limit - 단일 호출에서 반환할 최대 항목 수를 지정합니다.

- nextToken - 다음 결과 세트를 검색합니다(nextToken의 값은 이전 호출에서 가져올 수 있음).

5. 해석기의 모든 변경 사항을 저장합니다.

Query 요청에 대한 자세한 내용은 쿼리 참조 문서를 참조하세요.

작성자별 모든 게시물을 쿼리하는 API 호출

이제 해석기가 설정되었으므로 AWS AppSync는 들어오는 allPostsByAuthor 뮤테이션을 author-index 인덱스에 대한 DynamoDB Query 작업으로 변환하는 방법을 알고 있습니다. 이제 테이블을 쿼리해 특정 작성자별로 게시물을 모두 검색할 수 있습니다.

하지만 그 전에, 지금까지 모든 게시물의 작성자가 동일하므로 테이블을 몇 개의 게시물로 더 채워 보겠습니다.

데이터를 추가하고 쿼리하려면

1. API에서 쿼리 탭을 선택합니다.
2. 쿼리 창에 다음 뮤테이션을 추가합니다.

```
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. 실행(주황색 재생 버튼)을 선택한 다음 addPost를 선택합니다.
4. 이제 Nadia가 작성한 게시물을 모두 반환하도록 테이블을 쿼리하겠습니다. 쿼리 창에 다음 쿼리를 추가합니다.

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
```

```
  }
}
```

5. 실행(주황색 재생 버튼)을 선택한 다음 allPostsByAuthor를 선택합니다. Nadia가 작성한 모든 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

6. Query에 대한 페이지 매김은 Scan을 수행할 때와 동일하게 작동합니다. 예를 들어, AUTHORNAME의 모든 게시물을 살펴보겠습니다. 이 경우 한 번에 5개 결과를 반환합니다.
7. 쿼리 창에 다음 쿼리를 추가합니다.

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

8. 실행(주황색 재생 버튼)을 선택한 다음 allPostsByAuthor를 선택합니다. AUTHORNAME가 작성한 모든 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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. 다음과 같이 이전 쿼리에서 반환된 값으로 nextToken 인수를 업데이트합니다.

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
```

```
      title
    }
    nextToken
  }
}
```

10.실행(주황색 재생 버튼)을 선택한 다음 `allPostsByAuthor`를 선택합니다. `AUTHORNAME`이 작성한 나머지 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

집합 사용

지금까지 `Post` 유형은 플랫 키/값 객체였습니다. 또한 해석기를 사용하여 집합, 목록 및 맵 등과 같은 복잡한 객체를 모델링할 수 있습니다. 태그를 포함하도록 `Post` 형식을 업데이트해 보겠습니다. 게시물에는 태그가 0개 이상 있을 수 있는데, 태그는 DynamoDB에 문자열 집합으로 저장됩니다. 또한 태그

를 추가 및 제거하는 몇 가지 변형과 특정 태그가 지정된 게시물을 스캔하는 새 쿼리를 설정해 볼 것입니다.

데이터를 설정하려면

1. API에서 스키마 탭을 선택합니다.
2. 스키마 창에서 다음과 같이 Post 형식을 수정하여 새로운 tags 필드를 추가할 수 있습니다.

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
}
```

3. 스키마 창에서 다음과 같이 Query 형식을 수정하여 새로운 allPostsByTag 쿼리를 추가할 수 있습니다.

```
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. 스키마 창에서 다음과 같이 Mutation 유형을 수정하여 새로운 addTag 및 removeTag 뮤테이션을 추가할 수 있습니다.

```
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. 스키마 저장(Save Schema)을 선택합니다.
6. 오른쪽의 해석기 창에서 Query 유형에 새로 생성된 allPostsByTag 필드를 찾은 다음 연결을 선택합니다. 아래 코드 조각을 사용하여 해석기를 만듭니다.

```
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. 해석기의 모든 변경 사항을 저장합니다.
8. 이제 아래 코드 조각을 사용하여 Mutation 필드 addTag에 대해 동일한 작업을 수행합니다:

> **Note**
>
> DynamoDB 유틸리티는 현재 집합 작업을 지원하지 않지만, 직접 요청을 작성하여 집합과 상호 작용할 수 있습니다.

```
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. 해석기의 모든 변경 사항을 저장합니다.

10.아래 코드 조각을 사용하여 Mutation 필드 removeTag에 대해 이 작업을 한 번 더 반복합니다.

```
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.해석기의 모든 변경 사항을 저장합니다.

태그를 사용하는 API 호출

이제 해석기를 설정했으므로, AWS AppSync에서는 수신되는 `addTag`, `removeTag` 및 `allPostsByTag` 요청을 DynamoDB `UpdateItem` 및 `Scan` 작업으로 변환하는 방법을 알고 있습니다. 실행해 보기 위해 이전에 생성한 게시물 중 선택해 보겠습니다. 예를 들어, `Nadia`에서 작성한 게시물을 사용해 보겠습니다.

태그를 사용하려면

1. API에서 쿼리 탭을 선택합니다.
2. 쿼리 창에 다음 쿼리를 추가합니다.

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

3. 실행(주황색 재생 버튼)을 선택한 다음 `allPostsByAuthor`를 선택합니다.
4. Nadia의 모든 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
```

```
    }
  }
}
```

5. 세상에서 가장 귀여운 강아지라는 제목이 붙은 게시물을 사용해 봅시다. 나중에 사용할 것이기 때문에 이 게시물의 id를 기록해 둡니다. 이제 dog 태그를 추가해 보겠습니다.
6. 쿼리 창에 다음 뮤테이션을 추가합니다. 또한 앞서 적어둔 값을 갖도록 id 인수를 업데이트해야 합니다.

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

7. 실행(주황색 재생 버튼)을 선택한 다음 addTag를 선택합니다. 다음과 같이 새 태그로 게시물이 업데이트됩니다.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

8. 태그를 더 추가할 수 있습니다. tag 인수를 puppy로 변경하도록 뮤테이션을 업데이트합니다.

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

9. 실행(주황색 재생 버튼)을 선택한 다음 addTag를 선택합니다. 다음과 같이 새 태그로 게시물이 업데이트됩니다.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

10. 태그를 삭제할 수도 있습니다. 쿼리 창에 다음 뮤테이션을 추가합니다. 또한 앞서 적어둔 값을 갖도록 id 인수를 업데이트해야 합니다.

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

11. 실행(주황색 재생 버튼)을 선택한 다음 removeTag를 선택합니다. 게시물이 업데이트되고 puppy 태그가 삭제됩니다.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

12또한 태그가 지정된 게시물을 모두 검색할 수도 있습니다. 쿼리 창에 다음 쿼리를 추가합니다.

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

13.실행(주황색 재생 버튼)을 선택한 다음 `allPostsByTag`를 선택합니다. dog 태그가 지정된 게시물이 다음과 같이 모두 반환됩니다.

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
            "dog",
            "puppy"
          ]
        }
      ],
      "nextToken": null
    }
  }
}
```

결론

이 자습서에서는 AWS AppSync 및 GraphQL을 사용하여 DynamoDB의 `Post` 객체를 조작할 수 있는 API를 구축했습니다.

정리를 위해 콘솔에서 AWS AppSync GraphQL API를 삭제할 수 있습니다.

DynamoDB 테이블과 연결된 역할을 삭제하려면 데이터 소스 테이블에서 데이터 소스를 선택하고 편집을 클릭합니다. 새 역할 생성 또는 기존 역할 사용에서 역할 값을 기록해 둡니다. IAM 콘솔로 이동하여 역할을 삭제합니다.

DynamoDB 테이블을 삭제하려면 데이터 소스 목록에서 테이블 이름을 클릭합니다. 그러면 테이블을 삭제할 수 있는 DynamoDB 콘솔로 이동합니다.

자습서: Lambda 해석기

AWS AppSync와 함께 AWS Lambda를 사용하여 모든 GraphQL 필드를 확인할 수 있습니다. 예를 들어, GraphQL 쿼리는 Amazon Relational Database Service(Amazon RDS) 인스턴스로 호출을 보내고, GraphQL 뮤테이션은 Amazon Kinesis 스트림에 쓸 수 있습니다. 이 단원에서는 GraphQL 필드 작업의 호출에 따라 비즈니스 로직을 수행하는 Lambda 함수의 작성 방법을 설명합니다.

Lambda 함수 생성

다음 예는 블로그 게시물 애플리케이션의 일부로 블로그 게시물에 대해 다양한 연산을 수행하는 `Node.js`(런타임: Node.js 18.x)로 작성된 Lambda 함수를 보여줍니다. 단, 코드는 확장명이.mis인 파일 이름에 저장해야 합니다.

```
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)
  }
}
```

Lambda 함수는 ID별로 게시물을 검색하고, 게시물을 추가하고, 게시물 목록을 가져오고, 지정된 게시물에 대해 관련 게시물을 가져오는 작업을 처리합니다.

> **Note**
>
> 참고: `event.field`에 대해 `switch` 명령문을 사용하면 Lambda 함수가 현재 해석 중인 필드를 확인할 수 있습니다.

AWS 관리 콘솔을 사용하여 이 Lambda 함수를 생성합니다.

Lambda용 데이터 소스 구성

Lambda 함수를 만든 후 AWS AppSync 콘솔에서 GraphQL API로 이동한 다음 데이터 소스 탭을 선택합니다.

데이터 소스 생성을 선택하고 친숙한 데이터 소스 이름(예: **Lambda**)을 입력한 다음 데이터 소스 유형에서 AWS Lambda 함수를 선택합니다. 리전에서 함수와 동일한 리전을 선택합니다. 함수 ARN의 경우, Lambda 함수의 Amazon Resource Name(ARN)을 선택합니다.

Lambda 함수를 선택한 후 새 AWS Identity and Access Management(IAM) 역할을 만들거나(AWS AppSync에서 적절한 권한을 할당) 다음 인라인 정책이 있는 기존 역할을 선택할 수 있습니다.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/LAMBDA_FUNCTION"
        }
    ]
}
```

또한 다음과 같이 IAM 역할에 대해 AWS AppSync와 신뢰 관계를 설정해야 합니다:

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

GraphQL 스키마 생성

이제 데이터 소스가 Lambda 함수에 연결되었으며 GraphQL 스키마를 생성해 보겠습니다.

AWS AppSync 콘솔의 스키마 편집기에서 스키마가 다음 스키마와 일치하는지 확인합니다.

```
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]
}
```

해석기 구성

이제 Lambda 데이터 소스와 유효한 GraphQL 스키마를 등록했으며, 해석기를 사용하여 GraphQL 필드를 Lambda 데이터 소스에 연결할 수 있습니다.

AWS AppSync AppSync(APPSYNC_JS) 런타임을 사용하는 해석기를 생성하고 Lambda 함수와 상호 작용합니다. JavaScript를 사용하여 AWS AppSync 해석기 및 함수를 작성하는 방법에 대해 자세히 알아보려면 해석기 및 함수를 위한 JavaScript 런타임 기능을 참조하세요.

Lambda 매핑 템플릿에 대한 자세한 내용은 Lambda용 JavaScript 해석기 함수 참조를 참조하세요.

이 단계에서는 `getPost(id:ID!): Post`, `allPosts: [Post]`, `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`,

`Post.relatedPosts: [Post]` 필드에 대한 해석기를 Lambda 함수에 연결합니다. AWS AppSync 콘솔의 스키마 편집기에서 해석기 창의 `getPost(id:ID!): Post` 필드 옆에 있는 연결을 선택합니다. Lambda 데이터 소스를 선택합니다. 그런 다음, 다음 코드를 제공합니다.

```
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;
}
```

이 해석기 코드는 Lambda 함수를 호출할 때 필드 이름, 인수 목록, 소스 객체에 대한 컨텍스트를 Lambda 함수에 전달합니다. Save를 선택합니다.

이제 첫 번째 해석기를 성공적으로 연결했습니다. 나머지 필드에 대해 이 작업을 반복합니다.

GraphQL API 테스트

Lambda 함수가 GraphQL 해석기에 연결되었으므로 콘솔이나 클라이언트 애플리케이션을 사용하여 뮤테이션 및 쿼리를 실행할 수 있습니다.

AWS AppSync 콘솔에서 왼쪽에 있는 쿼리를 선택한 후, 다음 코드를 붙여 넣습니다.

addPost 변형

```
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 쿼리

```
query GetPost {
    getPost(id: "2") {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

allPosts 쿼리

```
query AllPosts {
    allPosts {
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {
            id
            title
        }
    }
}
```

오류 반환

필드 해상도를 지정하면 오류가 발생할 수 있습니다. AWS AppSync를 사용하면 다음 소스에서 오류를 발생시킬 수 있습니다.

- 해석기 응답 핸들러
- Lambda 함수

해석기 응답 핸들러에서

의도적인 오류를 발생시키려면 `util.error` 유틸리티 메서드를 사용하면 됩니다. `errorMessage`, `errorType` 및 선택적 `data` 값을 인수로 사용합니다. `data`는 오류가 발생한 경우 외부 데이터를 클라이언트로 반환할 때 유용합니다. GraphQL 최종 응답에서 `errors`에 `data` 객체가 추가됩니다.

다음 예시는 `Post.relatedPosts: [Post]` 해석기 응답 핸들러에서 이를 사용하는 방법을 보여줍니다.

```
// the Post.relatedPosts response handler
export function response(ctx) {
    util.error("Failed to fetch relatedPosts", "LambdaFailure", ctx.result)
    return ctx.result;
}
```

다음과 유사한 GraphQL 응답이 산출됩니다.

```
{
    "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"
                }
            ]
        }
    ]
}
```

여기서 `allPosts[0].relatedPosts`는 오류로 인해 null이며 `errorMessage`, `errorType` 및 `data`가 `data.errors[0]` 객체 안에 있습니다.

Lambda 함수에서

또한 AWS AppSync는 Lambda 함수에서 발생하는 오류를 이해합니다. Lambda 프로그래밍 모델은 handled 오류를 발생시킵니다. Lambda 함수에서 오류가 발생하는 경우 AWS AppSync에서 현재 필드가 해석되지 않습니다. Lambda에서 반환되는 오류 메시지만 응답에 설정됩니다. 현재는 Lambda 함수에서 오류를 발생시켜 관련 없는 데이터를 클라이언트에 다시 전달할 수 없습니다.

> **Note**
>
> Lambda 함수가 unhandled 오류를 발생시키는 경우 AWS AppSync는 Lambda에서 설정한 오류 메시지를 사용합니다.

다음 Lambda 함수는 오류를 발생시킵니다.

```
export const handler = async (event) => {
  console.log('Received event {}', JSON.stringify(event, 3))
  throw new Error('I always fail.')
}
```

응답 핸들러에 오류가 수신되었습니다. `util.appendError`로 응답에 오류를 추가하여 GraphQL 응답으로 다시 보낼 수 있습니다. 이렇게 하려면 AWS AppSync 함수 응답 핸들러를 다음과 같이 변경합니다.

```
// the lambdaInvoke response handler
export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  }
  return result;
}
```

다음과 유사한 GraphQL 응답이 반환됩니다.

```
{
  "data": {
    "allPosts": null
  },
  "errors": [
    {
      "path": [
        "allPosts"
      ],
      "data": null,
      "errorType": "Lambda:Unhandled",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "I fail. always"
    }
  ]
```

```
}
```

고급 사용 사례: 일괄 처리

이 예제의 Lambda 함수에는 해당 게시물에 대한 관련 게시물 목록을 반환하는 `relatedPosts` 필드가 있습니다. 예제 쿼리에서 Lambda 함수의 `allPosts` 필드 호출은 5개 게시물을 반환합니다. 각각의 반환된 게시물에 대해 `relatedPosts`를 해석하도록 지정했으므로 `relatedPosts` 필드 작업이 5회 호출됩니다.

```
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
        }
    }
}
```

이 특정 예제에서는 큰 문제가 되지 않을 수 있지만, 이러한 복합적이고 과도한 가져오기로 인해 애플리케이션이 빠르게 손상될 수 있습니다.

동일한 쿼리에서 반환된 관련 Posts에 대해 `relatedPosts`를 다시 가져오려 할 경우 호출 횟수가 크게 증가할 것입니다.

```
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
            }
        }
    }
}
```

이 매우 단순한 쿼리에서는 AWS AppSync가 Lambda 함수를 1 + 5 + 25 = 31번 호출합니다.

이는 상당히 일반적인 문제로 대개 N+1 문제(이 경우에 N = 5)라고도 하며 이로 인해 애플리케이션의 지연 시간 및 비용이 증가될 수 있습니다.

이 문제를 해결하는 한 가지 방법은 유사한 필드 해석기 요청을 하나로 묶는 것입니다. 이 예제에서는 지정된 단일 게시물에 대한 관련 게시물 목록을 해석하는 Lambda 함수 하나 대신에, 해당 게시물 배치에 대한 관련 게시물 목록을 해석합니다.

이를 설명하기 위해 일괄 처리를 처리하도록 `relatedPosts`의 해석기를 업데이트해 보겠습니다.

```
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;
}
```

이제 이 코드는 해석 중인 fieldName이 relatedPosts일 때 작업을 Invoke에서 BatchInvoke로 변경합니다. 이제 일괄 처리 구성 섹션에서 함수에 대한 일괄 처리를 활성화합니다. 설정된 최대 일괄 처리 크기를 5로 설정합니다. Save를 선택합니다.

이 변경으로 Lambda 함수는 relatedPosts 해결 시 다음을 입력으로 받습니다.

```
[
    {
        "field": "relatedPosts",
        "source": {
            "id": 1
        }
    },
    {
        "field": "relatedPosts",
        "source": {
            "id": 2
        }
    },
    ...
]
```

BatchInvoke가 요청에 지정되었으면 Lambda 함수가 요청 목록을 수신하고 결과 목록을 반환합니다.

특히, 결과 목록이 요청 페이로드 항목의 순서와 크기 면에서 일치해야 하므로 AWS AppSync가 그에 맞게 결과를 일치시킬 수 있습니다.

이 일괄 처리 예제에서 Lambda 함수는 다음과 같이 결과의 배치를 반환합니다.

```
[
    [{"id":"2","title":"Second book"}, {"id":"3","title":"Third book"}],   //
 relatedPosts for id=1
    [{"id":"3","title":"Third book"}]                                      //
 relatedPosts for id=2
]
```

Lambda 코드를 업데이트하여 relatedPosts에 대한 일괄 처리를 처리할 수 있습니다.

```
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)
  }
}
```

개별 오류 반환

앞의 예에서는 Lambda 함수에서 단일 오류가 반환되거나 응답 핸들러에서 오류를 일으킬 수 있음을 보았습니다. 일괄 처리된 호출의 경우, Lambda 함수에서 오류가 발생하면 전체 배치가 실패로 플래그 지정됩니다. 이는 데이터 스토어와의 연결 중단 같은 취소 불가능한 오류가 발생하는 특정 시나리오를 잘 설명할 수 있습니다. 하지만 배치의 일부 항목이 성공하고 다른 항목이 실패하는 경우, 오류 및 유효한 데이터를 모두 반환할 수 있습니다. AWS AppSync에서는 배치의 원래 크기와 일치하는 요소를 나열하기 위해 배치 응답이 필요하므로 유효한 데이터와 오류를 구분할 수 있는 데이터 구조를 사용자가 정의해야 합니다.

예를 들어 Lambda 함수가 관련 게시물의 일괄 처리를 반환해야 하는 경우 각 객체에 선택적 data, errorMessage 및 errorType 필드가 있는 `Response` 객체 목록을 반환하도록 선택할 수 있습니다. errorMessage 필드가 있는 경우 오류가 발생했음을 의미합니다.

다음 코드는 Lambda 함수를 업데이트하는 방법을 보여줍니다.

```
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)
  }
}
```

relatedPosts 해석기 코드 업데이트:

```
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
  }
}
```

이제 응답 핸들러는 Invoke 작업에 대해 Lambda 함수가 반환한 오류를 확인하고, BatchInvoke 작업에서 개별 항목에 대해 반환한 오류를 확인한 다음, 마지막으로 fieldName을 확인합니다. relatedPosts의 경우 함수는 result.data를 반환합니다. 다른 모든 필드의 경우 함수가 result를 반환합니다. 그 예로, 아래 쿼리를 살펴보겠습니다.

```
query AllPosts {
  allPosts {
    id
    title
    content
    url
```

```
    ups
    downs
    relatedPosts {
      id
    }
    author
  }
}
```

이 쿼리는 다음과 유사한 GraphQL 응답을 반환합니다.

```
{
  "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"
    }
  ]
}
```

최대 배치 크기 구성

해석기에서 최대 일괄 처리 크기를 구성하려면 AWS Command Line Interface(AWS CLI)에서 다음 명령을 사용합니다.

```
$ 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**
>
> 요청 매핑 템플릿을 제공할 때는 `BatchInvoke` 작업을 사용하여 일괄 처리를 사용해야 합니다.

자습서: 로컬 해석기

AWS AppSync를 사용하면 지원되는 데이터 소스(AWS Lambda, Amazon DynamoDB 또는 Amazon OpenSearch Service)를 사용하여 다양한 작업을 수행할 수 있습니다. 하지만 특정 시나리오에서는 지원되는 데이터 원본을 호출할 필요가 없을 수도 있습니다.

이러한 경우에는 로컬 해석기가 유용합니다. 로컬 해석기는 원격 데이터 소스를 호출하는 것이 아니라 단순히 요청 핸들러의 결과를 응답 핸들러로 전달할 뿐입니다. 필드 해석은 AWS AppSync에서 이루어집니다.

로컬 해석기는 다양한 상황에서 유용합니다. 가장 널리 알려진 사용 사례는 데이터 원본 호출을 트리거하지 않고 알림을 게시하는 경우입니다. 이 사용 사례를 시연하기 위해 사용자가 메시지를 게시하고 구독할 수 있는 게시/구독 애플리케이션을 만들어 보겠습니다. 이 예제에서는 Subscriptions를 활용하므로, Subscriptions에 대해 잘 모르는 경우 실시간 데이터 자습서에 따라 실행할 수도 있습니다.

게시/구독 앱 만들기

먼저 GraphQL API를 생성할 때 처음부터 디자인 옵션을 선택하고 선택적 세부 정보를 구성하여 빈 GraphQL API를 생성합니다.

게시/구독 애플리케이션에서 클라이언트는 메시지를 구독하고 게시할 수 있습니다. 게시된 각 메시지에는 이름과 데이터가 포함됩니다. 스키마에 다음을 추가합니다.

```
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"])
}
```

다음으로 Mutation.publish 필드에 해석기를 연결해 보겠습니다. 스키마 창 옆의 해석기 창에서 Mutation 유형을 찾고 publish(...): Channel 필드를 찾은 다음 연결을 클릭합니다.

없음 새 데이터 소스를 생성하고 이름을 PageDataSource로 지정합니다. 해석기에 연결합니다.

다음 코드 조각을 사용하여 해석기 구현을 추가합니다.

```
export function request(ctx) {
  return { payload: ctx.args };
}

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

해석기를 만들고 변경한 내용을 저장했는지 확인합니다.

메시지 전송 및 구독

클라이언트에서 메시지를 수신하도록 하려면 먼저 클라이언트가 받은 편지함을 구독해야 합니다.

쿼리 창에서 SubscribeToData 구독을 실행합니다.

```
subscription SubscribeToData {
    subscribe(name:"channel") {
        name
        data
    }
}
```

구독자는 publish 뮤테이션이 호출될 때마다 메시지를 수신하지만 메시지가 channel 구독으로 전송될 때만 수신됩니다. 쿼리 창에서 이 방법을 시도해 보겠습니다. 콘솔에서 구독이 계속 실행되는 동안 다른 콘솔을 열고 쿼리 창에서 다음 요청을 실행합니다.

> **Note**
>
> 이 예시에서는 유효한 JSON 문자열을 사용합니다.

```
mutation PublishData {
    publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
        data
        name
    }
}
```

결과는 다음과 같습니다.

```
{
  "data": {
    "publish": {
      "data": "{\"msg\":\"hello world!\"}",
      "name": "channel"
    }
  }
}
```

AWS AppSync를 나가지 않고 간단히 메시지를 게시하고 수신해 봄으로써 로컬 해석기 사용을 보여드렸습니다.

튜토리얼: GraphQL 해석기 결합

GraphQL 스키마의 해석기와 필드는 매우 뛰어난 유연성으로 1:1 관계를 갖습니다. 데이터 소스는 스키마와 독립적으로 해석기에 구성되므로, 다양한 데이터 소스를 통해 GraphQL 유형을 확인하거나 조작할 수 있으므로 필요에 따라 스키마를 혼합하고 일치시킬 수 있습니다.

다음 시나리오는 스키마에서 데이터 소스를 혼합하고 일치시키는 방법을 보여줍니다. 시작하기 전에, AWS Lambda, Amazon DynamoDB 및 Amazon OpenSearch Service에 대한 데이터 소스 및 해석기 구성에 익숙해야 합니다.

스키마 예제

다음 스키마에는 각각 3개의 `Query` 및 `Mutation` 작업이 있는 `Post` 유형이 있습니다.

```
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
}
```

이 예제에서는 각각 데이터 소스가 필요한 총 6개의 해석기가 있습니다. 이 문제를 해결하는 한 가지 방법은 `Posts`라는 단일 Amazon DynamoDB 테이블에 연결하여 `AllPost` 필드는 스캔을 실행하고 `searchPosts` 필드는 쿼리를 실행하는 것입니다(DynamoDB용 JavaScript 해석기 함수 참조 참조).

그러나 Amazon DynamoDB에만 국한되지는 않습니다. 비즈니스 요구 사항을 충족하기 위해 Lambda 또는 OpenSearch Service와 같은 다양한 데이터 소스가 존재합니다.

해석기를 통해 데이터 변경

AWS AppSync 데이터 소스에서 직접 지원하지 않는 타사 데이터베이스에서 결과를 반환해야 할 수도 있습니다. 또한 데이터를 API 클라이언트에 반환하기 전에 복잡한 수정을 수행해야 할 수도 있습니다. 이는 클라이언트의 타임스탬프 차이 또는 이전 버전과의 호환성 문제 처리와 같은 데이터 형식의 부적절한 서식으로 인해 발생할 수 있습니다. 이 경우 AWS Lambda 함수를 데이터 소스로서 AWS AppSync API에 연결하는 것이 적절한 해결책입니다. 설명을 위해, 다음 예제에서 AWS Lambda 함수는 서드 파티 데이터 스토어에서 가져온 데이터를 조작합니다.

```
export const handler = (event, context, callback) => {
    // fetch data
    const result = fetcher()

    // apply complex business logic
    const data = transform(result)

    // return to AppSync
    return data
};
```

이 함수는 완벽하게 유효한 Lambda 함수로 GraphQL 스키마의 `AllPost` 필드에 연결하여 모든 결과에서 반환하는 쿼리가 좋아요/싫어요에 대한 난수를 가져오도록 합니다.

DynamoDB 및 OpenSearch Service

일부 애플리케이션의 경우 DynamoDB에 대한 단순 조회 쿼리와 변형을 수행할 수 있으며, 백그라운드 프로세스를 통해 문서를 OpenSearch Service로 전송할 수 있습니다. 단순히 `searchPosts` 해석기를 OpenSearch Service 데이터 소스에 연결하고 GraphQL 쿼리를 사용하여 (DynamoDB에서 가져온 데이터의) 검색 결과를 반환할 수 있습니다. 이 기능은 키워드, 퍼지 워드 일치 또는 지역 검색 조회 등 애플리케이션에 고급 검색 작업 추가 시 매우 유용할 수 있습니다. DynamoDB에서 데이터 전송은 ETL 프로세스를 통해 수행하거나 Lambda을 사용하여 DynamoDB에서 스트리밍할 수 있습니다.

이러한 특정 데이터 소스로 시작하려면 DynamoDB 및 Lambda 자습서를 참조하세요.

예를 들어, 이전 튜토리얼의 스키마를 사용하여 다음 뮤테이션을 수행하면 DynamoDB에 항목이 추가됩니다.

```
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
  }
}
```

이렇게 하면 DynamoDB에 데이터가 기록되고, Lambda를 통해 Amazon OpenSearch Service로 데이터를 스트리밍합니다. 그러면 이를 사용하여 다양한 필드로 게시물을 검색할 수 있습니다. 예를 들면 데이터가 Amazon OpenSearch Service에 있으므로 다음과 같이 공백으로만 이루어졌더라도 자유 형식의 텍스트를 사용하여 작성자나 콘텐츠 필드를 검색할 수 있습니다.

```
query searchName{
    searchAuthor(name:"   Nadia   "){
        id
        title
        content
    }
}

---------- or ----------

query searchContent{
    searchContent(text:"test"){
        id
        title
        content
    }
}
```

데이터가 DynamoDB에 직접 기록되므로 `allPost{...}` 및 `getPost{...}` 쿼리를 사용하여 테이블에 대해 효율적인 목록이나 항목 조회 작업을 수행할 수 있습니다. 이 스택은 DynamoDB 스트림에 대해 다음 예제 코드를 사용합니다.

> **Note**
>
> 이 Python 코드는 예시이며 프로덕션 코드에 사용하기 위한 것이 아닙니다.

```
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.'
```

DynamoDB 스트림을 사용하여 `id`를 기본 키로 하는 DynamoDB 테이블에 이 부분을 연결하면, 원본에 대한 모든 변경 내용이 OpenSearch Service 도메인으로 스트리밍됩니다. 이 구성에 대한 자세한 내용은 DynamoDB Streams 설명서를 참조하십시오.

자습서: Amazon OpenSearch Service 해석기

AWS AppSync에서는 고유의 AWS 계정에서 프로비저닝한 도메인이 VPC 내에 존재하지 않는 경우, 이 도메인의 Amazon OpenSearch Service 사용을 지원합니다. 도메인을 프로비저닝한 후에는 도메인을 데이터 원본에 연결할 수 있으며, 이때 쿼리, 변형 및 구독 등 GraphQL 작업을 수행하도록 스키마의 해석기를 구성할 수 있습니다. 이 자습서에서는 몇 가지 일반적인 예제를 살펴봅니다.

자세한 내용은 OpenSearch용 JavaScript 해석기 함수 참조를 참조하세요.

새 OpenSearch Service 도메인 생성

이 자습서를 시작하기 위해서는 기존의 OpenSearch Service 도메인이 필요합니다. 아직 없는 경우 다음 샘플을 사용할 수 있습니다. AWS AppSync 데이터 소스로 이동하려면 먼저 최대 15분에 걸쳐 OpenSearch Service 도메인을 생성해야 합니다.

```
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
```

AWS 계정의 미국 서부(오레곤) 리전에서 다음 AWS CloudFormation 스택을 시작할 수 있습니다.

OpenSearch Service 데이터 소스 구성

OpenSearch Service 도메인이 생성된 후에는 AWS AppSync GraphQL API로 이동하여 데이터 소스 탭을 선택합니다. 데이터 소스 생성을 선택하고 데이터 소스의 친숙한 이름(예: '*oss*')을 입력합니다. 그런 다음 데이터 소스 유형에 대해 Amazon OpenSearch 도메인을 선택하고 적절한 리전을 선택하면 OpenSearch Service 도메인이 나열됩니다. 도메인을 선택한 후에는 새 역할을 생성할 수 있으며, 그러면 AWS AppSync에서 역할에 맞는 권한을 할당하거나 다음 인라인 정책이 지정된 기존 역할을 선택할 수 있습니다.

```
{
    "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/*"
            ]
        }
    ]
}
```

또한 해당 역할에 대해 AWS AppSync와의 신뢰 관계를 설정해야 합니다.

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

또한 OpenSearch Service 도메인에는 Amazon OpenSearch Service 콘솔을 통해 수정할 수 있는 자체 액세스 정책이 있습니다. 사용자는 OpenSearch Service 도메인에 대한 적정 작업 및 리소스를 사용하여 아래와 비슷한 정책을 추가해야 합니다. 보안 주체는 AWS AppSync 데이터 소스 역할이 되며, 해당 콘솔에서 생성하도록 허용한 경우 IAM 콘솔에서 찾을 수 있습니다.

```
{
    "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/*"
        }
    ]
}
```

해석기 연결

이제 데이터 소스가 OpenSearch Service 도메인에 연결되었으며, 다음 예제에서처럼 해석기를 사용하여 GraphQL 스키마에 연결할 수 있습니다.

```
 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
```

```
}
```

사용자 정의 Post 유형과 id 필드가 있습니다. 다음 예제에서는 이 유형을 OpenSearch Service 도메인에 넣는 프로세스(자동화할 수 있음)가 있다고 가정하고, 이 프로세스는 /post/_doc의 경로 루트에 매핑되며, 여기서 post는 인덱스입니다. 이 루트 경로에서 개별 문서 검색, /id/post*를 사용한 와일드카드 검색 또는 /post/_search 경로를 사용한 다중 문서 검색을 수행할 수 있습니다. 예를 들어, User라는 다른 유형이 있는 경우 user라는 새 인덱스로 문서를 색인한 다음 /user/_search의 경로로 검색을 수행할 수 있습니다.

AWS AppSync 콘솔의 스키마 편집기에서 searchPosts 쿼리를 포함하도록 이전 Posts 스키마를 수정합니다.

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

스키마를 저장합니다. 해석기 창에서 searchPosts를 찾아 연결을 선택합니다. OpenSearch Service 데이터 소스를 선택하고 해석기를 저장합니다. 아래 코드 조각을 사용하여 해석기 코드를 업데이트합니다.

```
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)
}
```

여기서는 앞의 스키마에 post 필드 아래에 OpenSearch Service에서 색인된 문서가 있다고 가정합니다. 데이터를 다르게 구조화하는 경우 그에 맞게 업데이트해야 합니다.

검색 수정

이전 해석기 요청 핸들러는 모든 레코드에 대해 단순 쿼리를 수행합니다. 특정 작성자별로 검색하려 한다고 가정하겠습니다. 또한, 작성자를 GraphQL 쿼리에 정의된 인수로 사용하려 한다고 가정하겠습니다. AWS AppSync 콘솔의 스키마 편집기에서 allPostsByAuthor 쿼리를 추가합니다.

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

해석기 창에서 allPostsByAuthor를 찾아 연결을 선택합니다. OpenSearch Service 데이터 소스를 선택하고 다음 코드를 사용합니다.

```
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)
}
```

body가 author 필드에 대한 쿼리 용어로 채워져서 클라이언트에서 인수로 전달됩니다. 선택 사항으로 표준 텍스트와 같이 미리 채워진 정보를 사용할 수도 있습니다.

OpenSearch Service에 데이터 추가

GraphQL 뮤테이션의 결과로서 OpenSearch Service 도메인에 데이터를 추가해야 할 수 있습니다. 이는 검색 및 다른 목적으로 사용할 때 유용한 메커니즘입니다. GraphQL 구독을 사용하여 데이터를 실시간으로 만들 수 있으므로, 이는 클라이언트에 OpenSearch Service 도메인의 데이터에 대한 업데이트를 알리는 메커니즘으로 작용할 수 있습니다.

AWS AppSync 콘솔의 스키마 페이지로 돌아가서 addPost() 뮤테이션에 대한 연결을 선택합니다. OpenSearch Service 데이터 소스를 다시 선택하고 다음 코드를 사용합니다.

```
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
}
```

전과 마찬가지로, 이 코드도 데이터의 구조화 방식을 보여주는 예입니다. 다양한 필드 이름 또는 인덱스가 있으면 path 및 body를 업데이트해야 합니다. 이 예는 요청 핸들러에서 `ctx.args`로 작성할 수도 있는 `context.arguments`를 사용하는 방법도 보여줍니다.

단일 문서 가져오기

마지막으로, 스키마에서 `getPost(id:ID)` 쿼리를 사용하여 개별 문서를 반환하려는 경우 AWS AppSync 콘솔의 스키마 편집기에서 이 쿼리를 찾아 연결을 선택합니다. OpenSearch Service 데이터 소스를 다시 선택하고 다음 코드를 사용합니다.

```
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
}
```

쿼리 및 뮤테이션 수행

이제 OpenSearch Service 도메인에 대해 GraphQL 작업을 수행할 수 있습니다. AWS AppSync 콘솔의 쿼리 탭으로 이동하여 새 레코드를 추가합니다.

```
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
       )
}
```

오른쪽에 뮤테이션 결과가 표시됩니다. 마찬가지로, 이제 OpenSearch Service 도메인에 대해 `searchPosts` 쿼리를 실행할 수 있습니다.

```
query search {
    searchPosts {
        id
        title
        author
        content
    }
}
```

모범 사례

- OpenSearch Service는 기본 데이터베이스가 아니라 데이터 쿼리용입니다. GraphQL 해석기 결합에서 설명했듯이 OpenSearch Service를 Amazon DynamoDB와 함께 사용할 수도 있습니다.
- AWS AppSync 서비스 역할에서 클러스터에 액세스하도록 허용하여 도메인에 대한 액세스 권한을 부여하기만 하면 됩니다.
- 최저 비용 클러스터를 제공하는 간단한 개발부터 시작하여, 프로덕션으로 들어가면서 고가용성(HA)을 제공하는 대규모 클러스터로 이동할 수 있습니다.

자습서: DynamoDB 트랜잭션 해석기

AWS AppSync에서는 단일 리전에 있는 하나 이상의 테이블에서 AppSync DynamoDB 트랜잭션 작업 사용을 지원합니다. 지원되는 작업은 `TransactGetItems`, `TransactWriteItems`입니다. AWS AppSync의 이러한 기능을 사용하면 다음과 같은 작업을 수행할 수 있습니다.

- 단일 쿼리에서 키 목록을 전달하고 테이블의 결과 반환
- 단일 쿼리의 하나 이상의 테이블에서 레코드 읽기
- 트랜잭션의 레코드를 하나 이상의 테이블에 전부 또는 전무 방식으로 기록
- 일부 조건이 충족되면 트랜잭션 실행

권한

다른 해석기와 마찬가지로 AWS AppSync에서 데이터 소스를 생성하고 역할을 생성하거나 기존 역할을 사용해야 합니다. 트랜잭션 작업에는 DynamoDB 테이블에 대한 여러 권한이 필요하기 때문에 읽기 또는 쓰기 작업에 대해 구성된 역할 권한을 부여해야 합니다.

```
{
    "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**
>
> 역할은 AWS AppSync의 데이터 소스에 연결되어 있고 필드의 해석기는 데이터 소스에 대해 간접 호출됩니다. DynamoDB에 대해 가져오도록 구성된 데이터 소스에는 구성을 간단하기 유지하기 위해 지정된 테이블이 하나뿐입니다. 따라서 단일 해석기에서 여러 테이블에 대해 트랜잭션 작업을 수행하는 경우(고급 작업임) 데이터 원본 액세스에 대한 역할을 해석기가 상호 작용하는 모든 테이블에 부여해야 합니다. IAM 정책의 리소스 필드에서 부여할 수 있습니다. 테이블에 대한 트랜잭션 호출의 구성은 해석기 코드에서 수행되는데, 이 내용은 아래에서 설명합니다.

데이터 원본

간단하게 설명하기 위해 이 자습서에서 사용되는 모든 해석기에 대해 동일한 데이터 원본을 사용합니다.

`accountNumber`를 파티션 키로 사용하는 savingAccounts 및 checkingAccounts라는 두 개의 테이블과 `transactionId`를 파티션 키로 사용하는 transactionHistory 테이블이 있습니다. 아래 CLI 명령을 사용하여 테이블을 만들 수 있습니다. `region`을 해당 리전으로 바꿔야 합니다.

CLI 사용

```
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
```

AWS AppSync 콘솔의 데이터 소스에서 새 DynamoDB 데이터 소스를 만들고 이름을 TransactTutorial로 지정합니다. savingAccounts를 테이블로 선택합니다(단, 트랜잭션을 사용할 때 특정 테이블은 중요하지 않음). 새 역할과 데이터 소스를 만들도록 선택합니다. 데이터 소스 구성을 검토하여 생성된 역할의 이름을 확인할 수 있습니다. IAM 콘솔에서 데이터 소스가 모든 테이블과 상호 작용하도록 허용하는 인라인 정책을 추가할 수 있습니다.

`region`과 `accountID`를 리전 및 계정 ID로 바꿉니다:

```
{
    "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/*"
            ]
        }
    ]
}
```

트랜잭션

이 예에서 컨텍스트는 전형적인 은행 거래이며 여기서 다음을 위해 `TransactWriteItems`를 사용합니다.

- 저축 계좌에서 당좌 예금 계좌로 돈을 이체
- 각 트랜잭션에 대한 새 트랜잭션 레코드 생성

그런 다음 `TransactGetItems`를 사용하여 저축 계좌와 당좌 예금 계좌의 세부 정보를 검색합니다.

GraphQL 스키마는 다음과 같이 정의합니다.

```
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 - 계좌 정보 채우기

계좌 간에 돈을 이체하려면 테이블에 세부 정보를 채워 넣어야 합니다. 이를 위해 GraphQL 작업 `Mutation.populateAccounts`를 사용하겠습니다.

스키마 섹션에서 `Mutation.populateAccounts` 작업 옆에 있는 연결을 클릭합니다. `TransactTutorial` 데이터 소스를 선택한 다음 생성을 선택합니다.

다음 코드를 사용합니다.

```
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 }
}
```

해석기를 저장하고 AppSync AWS 콘솔의 쿼리 섹션으로 이동하여 계좌 정보를 채웁니다.

다음 변형을 실행합니다.

```
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
    }
  }
}
```

하나의 뮤테이션으로 3개의 저축 계좌와 3개의 당좌 예금 계좌 정보를 채웠습니다.

DynamoDB 콘솔을 사용하여 데이터가 savingAccounts 및 checkingAccounts 테이블 모두에 표시되는지 확인합니다.

TransactWriteItems - 송금

다음 코드를 사용하여 `transferMoney` 뮤테이션에 해석기를 연결합니다. 이체할 때마다 당좌 예금 계좌와 저축 계좌 모두에 성공 한정자가 필요하고 거래의 이체를 추적해야 합니다.

```
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 }
}
```

이제 AWS AppSync 콘솔의 쿼리 섹션으로 이동하여 다음과 같이 transferMoney 뮤테이션을 실행합니다.

```
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
    }
  }
}
```

하나의 변형으로 3건의 은행 거래를 보냈습니다. DynamoDB 콘솔을 사용하여 데이터가 savingAccounts, checkingAccounts 및 transactionHistory 테이블에 표시되는지 확인합니다.

TransactGetItems - 계좌 검색

한 번의 트랜잭션 요청으로 저축 계좌와 당좌 예금 계좌의 세부 정보를 검색하기 위해 스키마의 `Query.getAccounts` GraphQL 작업에 해석기를 연결하겠습니다. 연결을 선택하고 자습서 시작 부분에서 생성한 `TransactTutorial` 데이터 소스를 선택합니다. 다음 코드를 사용합니다.

```
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 }
}
```

해석기를 저장하고 AppSync AWS 콘솔의 쿼리 섹션으로 이동합니다. 저축 계좌와 당좌 예금 계좌를 검색하려면 다음 쿼리를 실행합니다.

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
      username
      balance
    }
  }
}
```

AWS AppSync를 사용하여 DynamoDB 트랜잭션을 사용하는 방법을 보여드렸습니다.

자습서: DynamoDB 배치 해석기

AWS AppSync에서는 단일 리전에 있는 하나 이상의 테이블에서 AppSync DynamoDB 배치 작업 사용을 지원합니다. 지원되는 작업은 `BatchGetItem`, `BatchPutItem` 및 `BatchDeleteItem`입니다. AWS AppSync의 이러한 기능을 사용하면 다음과 같은 작업을 수행할 수 있습니다.

- 단일 쿼리에서 키 목록을 전달하고 테이블의 결과 반환
- 단일 쿼리의 하나 이상의 테이블에서 레코드 읽기
- 하나 이상의 테이블에 대량으로 레코드 쓰기
- 관계가 있을 수 있는 여러 테이블에서 조건부로 레코드 쓰기 또는 삭제

AWS AppSync의 배치 작업에는 배치가 아닌 작업과 크게 차이나는 점이 2개 있습니다.

- 데이터 원본 역할에 해석기가 액세스할 모든 테이블에 대한 권한이 있어야 합니다.
- 해석기에 대한 테이블 사양이 요청 객체의 일부입니다.

단일 테이블 배치

시작하기 위해 새로운 GraphQL API를 만들어 보겠습니다. AWS AppSync 콘솔에서 API 생성, GraphQL API 및 처음부터 설계를 선택합니다. API 이름을 `BatchTutorial API`로 지정하고 다음을 선택한 후, GraphQL 리소스 지정 단계에서 나중에 GraphQL 리소스 생성을 선택하고 다음을 클릭합니다. 세부 정보를 검토하고 API를 생성합니다. 스키마 페이지로 이동하여 다음 스키마를 붙여넣고, 쿼리에서 ID 목록을 전달한다는 점에 유의합니다.

```
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]
}
```

스키마를 저장하고 페이지 상단에서 리소스 생성을 선택합니다. 기존 유형 사용을 선택하고 Post 유형을 선택합니다. 테이블 Posts의 이름을 지정합니다. 기본 키가 id로 설정되어 있는지 확인하고 GraphQL 자동 생성을 선택 취소한 다음(직접 코드를 제공해야 함) 생성을 선택합니다. 시작하기 위해 AWS AppSync는 새 DynamoDB 테이블과 적절한 역할로 테이블에 연결된 데이터 소스를 생성합니다. 하지만 여전히 역할에 추가해야 하는 몇 가지 권한이 있습니다. 데이터 소스 페이지로 이동하여 새 데이터 소스를 선택합니다. 기존 역할 선택에서 테이블에 대한 역할이 자동으로 생성된 것을 확인할 수 있습니다. 역할(appsync-ds-ddb-aaabbbcccddd-Posts와 비슷하게 표시되어야 함)을 메모한 다음 IAM 콘솔(https://console.aws.amazon.com/iam/)로 이동합니다. IAM 콘솔에서 역할을 선택한 다음 표에서 역할을 선택합니다. 역할의 권한 정책에서 정책 옆에 있는 '+'를 클릭합니다(역할 이름과 이름이 비슷해야 함). 정책이 나타나면 접을 수 있는 항목의 상단에서 편집을 선택합니다. 정책에 일괄 권한(특히 dynamodb:BatchGetItem 및 dynamodb:BatchWriteItem)을 추가해야 합니다. 결과는 다음과 같을 것입니다.

```
{
    "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:…"
            ]
        }
    ]
}
```

다음을 선택한 후 변경 사항 저장을 선택합니다. 이제 정책에서 일괄 처리를 허용해야 합니다.

AWS AppSync 콘솔로 돌아가서 스키마 페이지로 이동하여 `Mutation.batchAdd` 필드 옆의 첨부를 선택합니다. `Posts` 테이블을 데이터 소스로 사용하여 해석기를 만듭니다. 코드 편집기에서 핸들러를 아래 코드 조각으로 바꿉니다. 이 코드 조각은 GraphQL `input PostInput` 유형의 각 항목을 자동으로 가져와서 `BatchPutItem` 작업에 필요한 맵을 빌드합니다.

```
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;
}
```

AWS AppSync 콘솔의 쿼리 페이지로 이동하여 다음 `batchAdd` 뮤테이션을 실행합니다.

```
mutation add {
    batchAdd(posts:[{
            id: 1 title: "Running in the Park"},{
            id: 2 title: "Playing fetch"
        }]){
            id
            title
    }
}
```

화면에 결과가 인쇄되어 있어야 합니다. DynamoDB 콘솔을 검토하여 `Posts` 테이블에 기록된 값을 스캔하면 결과를 확인할 수 있습니다.

그런 다음 해석기를 연결하는 과정을 반복하되, Posts 테이블을 데이터 소스로 사용하는 Query.batchGet 필드에 대해 반복합니다. 핸들러를 아래 코드로 바꿉니다. 이렇게 하면 GraphQL ids:[] 형식의 각 항목을 자동으로 받아 BatchGetItem 작업에 필요한 맵을 빌드합니다.

```
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;
}
```

이제 AWS AppSync 콘솔의 쿼리 페이지로 돌아가 다음 batchGet 쿼리를 실행합니다.

```
query get {
    batchGet(ids:[1,2,3]){
        id
        title
    }
}
```

앞서 추가한 두 id 값에 대한 결과를 반환해야 합니다. null 값은 값이 id인 3에 대해 반환되었습니다. 이는 Posts 테이블에 해당 값을 가진 레코드가 아직 없기 때문입니다. 또한, AWS AppSync에서는 쿼리에 키가 전달된 순서대로 결과를 반환하는데, 이는 AWS AppSync에서 사용자를 대신하여 수행하는 추가 기능입니다. 따라서 batchGet(ids:[1,3,2])로 전환하면 순서가 바뀝니다. 또한 어떤 id가 null 값을 반환하는지 알게 됩니다.

마지막으로 Posts 테이블을 데이터 소스로 사용하여 Mutation.batchDelete 필드에 해석기를 하나 더 연결합니다. 핸들러를 아래 코드로 바꿉니다. 이렇게 하면 GraphQL ids:[] 형식의 각 항목을 자동으로 받아 BatchGetItem 작업에 필요한 맵을 빌드합니다.

```
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;
}
```

이제 AWS AppSync 콘솔의 쿼리 페이지로 돌아가 다음 batchDelete 뮤테이션을 실행합니다.

```
mutation delete {
    batchDelete(ids:[1,2]){ id }
}
```

이제 id 1 및 2인 레코드가 삭제되어야 합니다. 앞에서 batchGet() 쿼리를 다시 실행하면 null을 반환해야 합니다.

다중 테이블 배치

또한 AWS AppSync에서는 여러 테이블 간에 배치 작업을 수행할 수 있습니다. 더 복잡한 애플리케이션을 빌드해 보겠습니다. 센서가 반려동물의 위치와 체온을 알려주는 반려동물 건강 앱을 개발한다고 상상해 보세요. 센서는 배터리로 구동되며 몇 분마다 네트워크에 연결을 시도합니다. 센서가 연결을 설정하면 판독값을 AWS AppSync API로 전송합니다. 그런 다음 트리거가 데이터를 분석해 애완동물 주인에게 대시보드가 제공됩니다. 센서와 백엔드 데이터 스토어 간의 상호 작용에 대해 중점적으로 설명해 보겠습니다.

AWS AppSync 콘솔에서 API 생성, GraphQL API 및 처음부터 설계를 선택합니다. API 이름을 `MultiBatchTutorial API`로 지정하고 다음을 선택한 후, GraphQL 리소스 지정 단계에서 나중에 GraphQL 리소스 생성을 선택하고 다음을 클릭합니다. 세부 정보를 검토하고 API를 생성합니다. 스키마 페이지로 이동하여 다음 스키마를 붙여넣고 저장합니다.

```
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
}
```

DynamoDB 테이블 두 개를 생성해야 합니다.

- `locationReadings`는 센서 위치 판독값을 저장합니다.
- `temperatureReadings`는 센서 온도 측정값을 저장합니다.

두 테이블은 동일한 기본 키 구조를 공유합니다. 파티션 키는 `sensorId (String)`, 정렬 키는 `timestamp (String)`입니다.

페이지 상단에서 리소스 생성을 선택합니다. 기존 유형 사용을 선택하고 `locationReadings` 유형을 선택합니다. 테이블 `locationReadings`의 이름을 지정합니다. 기본 키는 `sensorId`로, 정렬 키는 `timestamp`로 설정되어 있는지 확인합니다. GraphQL 자동 생성을 선택 취소하고(직접 코드를 제공해야 함) 생성을 선택합니다. `temperatureReadings`를 유형 및 테이블 이름으로 사용하여 `temperatureReadings`에 이 프로세스를 반복합니다. 위와 같은 키를 사용합니다.

새 테이블에는 자동으로 생성된 역할이 포함됩니다. 여전히 역할에 추가해야 하는 몇 가지 권한이 있습니다. 데이터 소스 페이지로 이동하여 `locationReadings`를 선택합니다. 기존 역할 선택에서 역할을 볼 수 있습니다. 역할(`appsync-ds-ddb-aaabbbcccddd-locationReadings`와 비슷하게 표시되어야 함)을 메모한 다음 IAM 콘솔(https://console.aws.amazon.com/iam/)로 이동합니다. IAM 콘솔에서 역할을 선택한 다음 표에서 역할을 선택합니다. 역할의 권한 정책에서 정책 옆에 있는 '+'를 클릭합니다(역할 이름과 이름이 비슷해야 함). 정책이 나타나면 접을 수 있는 항목의 상단에서 편집을 선택합니다. 이 정책에 권한을 추가해야 합니다. 결과는 다음과 같을 것입니다.

```
{
    "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/*"
            ]
        }
    ]
}
```

다음을 선택한 후 변경 사항 저장을 선택합니다. 위와 동일한 정책 코드 조각을 사용하여 `temperatureReadings` 데이터 소스에 대해 이 프로세스를 반복합니다.

BatchPutItem - 센서 판독값 기록

센서는 인터넷에 연결되면 판독값을 전송할 수 있어야 합니다. GraphQL 필드 `Mutation.recordReadings`가 센서가 판독값을 전송하는 데 사용할 API입니다. 이 필드에 해석기를 추가해야 합니다.

AWS AppSync 콘솔의 스키마 페이지에서 `Mutation.recordReadings` 필드 옆의 첨부를 선택합니다. 다음 화면에서 `locationReadings` 테이블을 데이터 소스로 사용하여 해석기를 만듭니다.

해석기를 생성한 후 편집기에서 다음 코드를 사용하여 핸들러를 대체합니다. 이 `BatchPutItem` 작업을 통해 여러 테이블을 지정할 수 있습니다.

```
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
}
```

배치 작업을 사용하면 호출 시 오류 및 결과가 둘 다 반환될 수 있습니다. 이러한 경우 몇 가지 오류 처리를 작업을 자유롭게 수행할 수 있습니다.

> **Note**
>
> `utils.appendError()` 사용은 `util.error()`와 유사하지만 요청 또는 응답 핸들러 평가를 중단하지 않는다는 큰 차이점이 있습니다. 대신 필드에 오류가 있다는 신호를 보내지만 템플릿이 계속해서 평가되도록 하고 이어서 호출자에게 데이터를 반환합니다. 애플리케이션이 일부 결과를 반환해야 하는 경우에는 `utils.appendError()`를 사용하는 것이 좋습니다.

해석기를 저장하고 AppSync AWS 콘솔의 쿼리 페이지로 이동합니다. 이제 일부 센서 측정값을 전송할 수 있습니다.

다음 변형을 실행합니다.

```
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
    }
  }
}
```

하나의 뮤테이션으로 센서 판독값 10개를 전송했고, 판독값은 테이블 2개로 분할되어 있었습니다. DynamoDB 콘솔을 사용하여 데이터가 `locationReadings` 및 `temperatureReadings` 테이블 모두에 표시되는지 확인합니다.

BatchDeleteItem - 센서 판독값 삭제

마찬가지로 센서 판독값의 배치를 삭제할 수 있어야 합니다. 이를 위해 `Mutation.deleteReadings` GraphQL 필드를 사용해 보겠습니다. AWS AppSync 콘솔의 스키마 페이지에서 `Mutation.deleteReadings` 필드 옆의 첨부를 선택합니다. 다음 화면에서 `locationReadings` 테이블을 데이터 소스로 사용하여 해석기를 만듭니다.

해석기를 생성한 후 코드 편집기에서 핸들러를 아래의 코드 조각으로 대체합니다. 이 해석기에서는 제공된 입력에서 `sensorId`와 `timestamp`를 추출하는 도우미 함수 매퍼를 사용합니다.

```
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
}
```

해석기를 저장하고 AppSync AWS 콘솔의 쿼리 페이지로 이동합니다. 이제 센서 판독값 두 개를 삭제해 보겠습니다.

다음 변형을 실행합니다.

```
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**
>
> `DeleteItem` 작업과 달리 완전히 삭제된 항목은 응답에서 반환되지 않습니다. 전달된 키만 반환됩니다. 자세히 알아보려면 DynamoDB용 JavaScript 해석기 함수 참조의 BatchDeleteItem을 참조하세요.

DynamoDB 콘솔을 사용하여 판독값 2개가 `locationReadings` 및 `temperatureReadings` 테이블에서 삭제되었는지 확인합니다.

BatchGetItem - 판독값 검색

앱의 또 다른 일반 작업은 특정 시점에 센서의 판독값을 검색하는 것입니다. 스키마의 `Query.getReadings` GraphQL 필드에 해석기를 연결해 보겠습니다. AWS AppSync 콘솔의 스키마 페이지에서 `Query.getReadings` 필드 옆의 첨부를 선택합니다. 다음 화면에서 `locationReadings` 테이블을 데이터 소스로 사용하여 해석기를 만듭니다.

다음 코드를 사용하겠습니다.

```
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' })),
```

```
  ]
}
```

해석기를 저장하고 AppSync AWS 콘솔의 쿼리 페이지로 이동합니다. 이제, 센서 판독값을 검색해 보겠습니다.

다음 쿼리를 실행합니다.

```
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
    }
  }
}
```

AWS AppSync을 사용하여 DynamoDB 배치 작업을 사용하는 방법을 보여드렸습니다.

오류 처리

AWS AppSync에서 데이터 소스 작업은 때때로 일부 결과를 반환할 수 있습니다. 일부 결과는 작업의 출력이 일부 데이터와 오류로 구성된 경우를 나타내는 데 사용되는 용어입니다. 오류는 본질적으로 애플리케이션별로 처리되므로 AWS AppSync에서는 응답 핸들러에서 오류를 처리할 수 있는 기회를 제공합니다. 해석기 호출 오류(있는 경우)는 컨텍스트에서 `ctx.error`로 확인할 수 있습니다. 호출 오류에는 항상 `ctx.error.message` 및 `ctx.error.type` 속성으로 액세스할 수 있는 메시지와 유형이 포함되어 있습니다. 응답 핸들러에서는 다음 세 가지 방법으로 부분 결과를 처리할 수 있습니다.

1. 데이터를 반환하여 호출 오류를 가립니다.
2. 핸들러 평가를 중단해 오류를 나타냅니다(`util.error(...)` 사용). 이 경우에는 데이터가 반환되지 않습니다.
3. 오류를 추가하고(`util.appendError(...)` 사용) 데이터도 반환합니다.

DynamoDB 배치 작업을 사용해 위의 3가지 방법을 각각 시연해 보겠습니다.

DynamoDB 배치 작업

DynamoDB 배치 작업을 사용하면 배치를 부분적으로 완료할 수 있습니다. 즉, 요청된 항목 또는 키 중 일부가 처리되지 않은 상태로 남아 있을 수 있습니다. AWS AppSync가 배치를 완료할 수 없는 경우 처리되지 않은 항목 및 호출 오류가 컨텍스트에 따라 설정됩니다.

이 자습서 이전 단원에서 사용한 `Query.getReadings` 작업의 `BatchGetItem` 필드 구성을 사용하여 오류 처리를 구현하겠습니다. 이때, `Query.getReadings` 필드를 실행하는 동안 `temperatureReadings` DynamoDB 테이블에 프로비저닝된 처리량이 부족하다고 가정해 보겠습니다. DynamoDB는 배치의 나머지 요소를 처리하기 위해 AWS AppSync의 두 번째 시도 중에 `ProvisionedThroughputExceededException`을 발생시켰습니다.

다음 JSON은 DynamoDB 배치 호출 이후, 응답 핸들러가 호출되기 전 직렬화된 컨텍스트를 나타냅니다.

```
{
  "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": []
}
```

컨텍스트에 대해서 알아야 할 몇 가지가 있습니다.

- 호출 오류는 AWS AppSync에 의해 컨텍스트에서 `ctx.error`로 설정되었으며 오류 유형은 `DynamoDB:ProvisionedThroughputExceededException`으로 설정되었습니다.
- 오류가 있더라도 결과는 `ctx.result.data`에서 테이블당 매핑됩니다.
- 처리되지 않고 남아 있는 키는 `ctx.result.data.unprocessedKeys`에서 확인할 수 있습니다. 여기서 AWS AppSync는 키(sensorId:1, timestamp:2018-02-01T17:21:05.000+08:00)를 사용해 항목을 검색할 수 없습니다. 테이블 처리량이 부족하기 때문입니다.

> **Note**
>
> `BatchPutItem`의 경우 `ctx.result.data.unprocessedItems`입니다.
> `BatchDeleteItem`의 경우 `ctx.result.data.unprocessedKeys`입니다.

이 오류를 세 가지 다른 방법으로 처리해 보겠습니다.

1. 호출 오류 가리기

호출 오류를 처리하지 않고 데이터를 반환하면 오류를 효과적으로 가려 주어진 GraphQL 필드에 대한 결과가 항상 성공입니다.

여기서 작성한 코드는 알기 쉽고 결과 데이터에만 중점을 두고 있습니다.

응답 핸들러

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

GraphQL 응답

```
{
  "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
      }
    ]
  }
}
```

데이터에 대해서만 실행되었으므로 오류 응답에 오류가 추가되지 않습니다.

2. 응답 핸들러 실행을 중단하는 오류 발생

클라이언트의 관점에서 부분 실패를 전체 실패로 간주해야 하는 경우 데이터가 반환되지 않도록 응답 핸들러 실행을 중단할 수 있습니다. `util.error(...)` 유틸리티 메서드가 정확하게 이러한 동작을 수행합니다.

응답 핸들러 코드

```
export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type, null,
 ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}
```

GraphQL 응답

```
{
  "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. (...)"
    }
  ]
}
```

일부 결과가 DynamoDB 배치 작업에서 반환될 수 있긴 하지만 오류를 일으키기로 선택합니다. 그러면 `getReadings` GraphQL 필드는 null이고 GraphQL 응답 오류 블록에 오류가 추가됩니다.

3. 오류를 추가하고 데이터 및 오류를 둘 다 반환

특정한 경우 사용자 경험을 개선하기 위해 애플리케이션에서는 일부 결과를 반환하고 클라이언트에게 처리되지 않은 항목이 있음을 알릴 수 있습니다. 그러면 클라이언트는 재시도를 수행하거나 오류를 최종 사용자에게 다시 번역할지 결정할 수 있습니다. `util.appendError(...)`는 애플리케이션 디자이너가 응답 핸들러 평가를 방해하지 않고 컨텍스트에 오류를 추가하도록 하여 이러한 동작을 가능하게 하는 유틸리티 메서드입니다. 응답 핸들러 평가 후 AWS AppSync에서는 GraphQL 응답의 오류 블록에 오류를 추가해 모든 컨텍스트 오류를 처리합니다.

응답 핸들러 코드

```
export function response(ctx) {
  if (ctx.error) {
    util.appendError(ctx.error.message, ctx.error.type, null,
 ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}
```

GraphQL 응답의 오류 블록 내에서 호출 오류 및 unprocessedKeys 요소를 전달합니다. 아래 응답에 표시된 것처럼 getReadings 필드 역시 locationReadings 테이블의 일부 데이터를 반환합니다.

GraphQL 응답

```
{
  "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. (...)"
    }
  ]
}
```

자습서: HTTP 해석기

AWS AppSync에서는 지원되는 데이터 소스(즉 AWS Lambda, Amazon DynamoDB, Amazon OpenSearch Service 또는 Amazon Aurora)을 사용하여 다양한 작업을 수행하고 모든 임의 HTTP 엔드포인트를 통해 GraphQL 필드를 해석합니다. HTTP 엔드포인트가 사용 가능해진 후에는 데이터 원본에 연결할 수 있습니다. 쿼리, 변형 및 구독 등 GraphQL 작업을 수행하도록 스키마의 해석기를 구성할 수 있습니다. 이 자습서에서는 몇 가지 일반적인 예제를 살펴봅니다.

이 자습서에서는 AWS AppSync GraphQL 엔드포인트에 REST API(Amazon API Gateway 및 Lambda를 사용하여 생성)를 사용합니다.

REST API 만들기

다음 AWS CloudFormation 템플릿을 사용하여 이 자습서에 사용되는 REST 엔드포인트를 설정할 수 있습니다.

AWS CloudFormation 스택은 다음 단계를 수행합니다.

1. 마이크로서비스에 대한 비즈니스 로직이 포함된 Lambda 함수를 설정합니다.
2. 다음 엔드포인트/메서드/콘텐츠 유형 조합으로 API Gateway REST API를 설정합니다.

API 리소스 경로	HTTP 메서드	지원되는 콘텐츠 유형
/v1/사용자	POST	application/json
/v1/사용자	GET	application/json

API 리소스 경로	HTTP 메서드	지원되는 콘텐츠 유형
/v1/사용자/1	GET	application/json
/v1/사용자/1	PUT	application/json
/v1/사용자/1	DELETE	application/json

GraphQL API 생성

AWS AppSync에서 GraphQL API를 생성하려면:

1. AWS AppSync 콘솔을 열고 API 생성을 선택합니다.
2. GraphQL API를 선택한 다음 처음부터 새로 디자인을 선택합니다. 다음(Next)을 선택합니다.
3. API에 `UserData`를 입력합니다. 다음(Next)을 선택합니다.
4. `Create GraphQL resources later`를 선택합니다. 다음(Next)을 선택합니다.
5. 입력 내용을 검토하고 API 생성을 선택합니다.

AWS AppSync 콘솔에서 API 키 인증 모드를 사용하여 새 GraphQL API를 생성합니다. 콘솔을 사용하여 GraphQL API를 추가로 구성하고 요청을 실행할 수 있습니다.

GraphQL 스키마 생성

GraphQL API가 마련되었으면 GraphQL 스키마를 생성해 보십시오. AWS AppSync 콘솔의 스키마 편집기에서 아래 코드 조각을 사용합니다.

```
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
}
```

HTTP 데이터 소스 구성

HTTP 데이터 원본을 구성하려면 다음을 수행하십시오.

1. AWS AppSync GraphQL API의 데이터 소스 페이지에서 데이터 소스 생성을 선택합니다.
2. 데이터 소스의 이름(예: `HTTP_Example`)을 입력합니다.
3. 데이터 소스 유형에서 HTTP 엔드포인트를 선택합니다.
4. 자습서의 시작 부분에서 생성한 API 게이트웨이 엔드포인트로 엔드포인트를 설정합니다. Lambda 콘솔로 이동하여 애플리케이션에서 애플리케이션을 찾으면 스택에서 생성된 엔드포인트를 찾을 수 있습니다. 애플리케이션의 설정에서 AWS AppSync의 엔드포인트가 될 API 엔드포인트를 확인할 수 있습니다. 단계 이름을 엔드포인트의 일부로 포함하지 않았는지 확인합니다. 예를 들어 엔드포인트가 `https://aaabbbcccd.execute-api.us-east-1.amazonaws.com/v1`인 경우 `https://aaabbbcccd.execute-api.us-east-1.amazonaws.com`을 입력합니다.

> **Note**
>
> 현재 AWS AppSync는 퍼블릭 엔드포인트만 지원합니다.
> AWS AppSync 서비스에서 인식하는 인증 기관에 대한 자세한 내용은 HTTPS 엔드포인트에 대해 AWS AppSync에서 인식하는 인증 기관(CA)을 참조하세요.

해석기 구성

이 단계에서는 HTTP 데이터 소스를 `getUser` 및 `addUser` 쿼리에 연결합니다.

getUser 해석기를 설정하려면:

1. AWS AppSync GraphQL API에서 스키마 탭을 선택합니다.
2. 스키마 편집기의 오른쪽에 있는 해석기 창의 쿼리 유형 아래에서 getUser 필드를 찾아 연결을 선택합니다.
3. 해석기 유형을 Unit으로 유지하고 런타임을 APPSYNC_JS로 유지합니다.
4. 데이터 소스 이름에서 이전에 만든 HTTP 엔드포인트를 선택합니다.
5. 생성을 선택합니다.
6. 해석기 코드 편집기에서 다음 코드 조각을 요청 핸들러로 추가합니다.

```
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. 다음 코드 조각을 응답 핸들러로 추가합니다.

```
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. 쿼리 탭을 선택한 후, 다음 쿼리를 실행합니다.

```
query GetUser{
    getUser(id:1){
```

```
        id
        username
    }
}
```

그러면 다음과 같은 응답이 반환됩니다.

```
{
    "data": {
        "getUser": {
            "id": "1",
            "username": "nadia"
        }
    }
}
```

addUser 해석기를 설정하려면:

1. 스키마 탭을 선택합니다.
2. 스키마 편집기의 오른쪽에 있는 해석기 창의 쿼리 유형 아래에서 addUser 필드를 찾아 연결을 선택합니다.
3. 해석기 유형을 Unit으로 유지하고 런타임을 APPSYNC_JS로 유지합니다.
4. 데이터 소스 이름에서 이전에 만든 HTTP 엔드포인트를 선택합니다.
5. 생성을 선택합니다.
6. 해석기 코드 편집기에서 다음 코드 조각을 요청 핸들러로 추가합니다.

```
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. 다음 코드 조각을 응답 핸들러로 추가합니다.

```
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. 쿼리 탭을 선택한 후, 다음 쿼리를 실행합니다.

```
mutation addUser{
    addUser(userInput:{
        id:"2",
        username:"shaggy"
    }){
        id
        username
    }
}
```

`getUser` 쿼리를 다시 실행하면 다음과 같은 응답이 반환되어야 합니다.

```
{
    "data": {
        "getUser": {
        "id": "2",
        "username": "shaggy"
        }
    }
}
```

AWS 서비스 간접 호출

HTTP 해석기를 사용하여 AWS 서비스에 대한 GraphQL API 인터페이스를 설정할 수 있습니다. AWS에 대한 HTTP 요청은 서명 버전 4 프로세스로 서명해야 AWS가 요청을 보낸 사람을 식별할 수 있습니

다. AWS AppSync는 IAM 역할을 HTTP 데이터 소스와 연결할 때 사용자를 대신하여 서명을 계산합니다.

HTTP 해석기를 사용하여 AWS 서비스를 호출하기 위해 다음 두 개의 추가 구성 요소를 제공합니다.

- AWS 서비스 API를 호출할 권한이 있는 AWS IAM 역할
- 데이터 원본의 서명 구성

예를 들어 HTTP 해석기를 사용하여 ListGraphqlApis 작업을 호출하려면 먼저 다음 정책을 연결하여 AWS AppSync가 가정하는 IAM 역할을 생성합니다.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "appsync:ListGraphqlApis"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

다음에는 AWS AppSync에 대한 HTTP 데이터 소스를 생성합니다. 이 예에서는 미국 서부(오레곤) 리전에서 AWS AppSync를 호출합니다. 서명 리전과 서비스 이름이 포함된 `http.json`이라는 파일에서 다음 HTTP 구성을 설정합니다.

```
{
    "endpoint": "https://appsync.us-west-2.amazonaws.com/",
    "authorizationConfig": {
        "authorizationType": "AWS_IAM",
        "awsIamConfig": {
            "signingRegion": "us-west-2",
            "signingServiceName": "appsync"
        }
    }
}
```

그런 다음, AWS CLI를 사용하여 다음과 같이 연결된 역할이 있는 데이터 소스를 생성합니다.

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

해석기를 스키마의 필드에 연결하는 경우 다음 요청 매핑 템플릿을 사용하여 AWS AppSync를 호출합니다.

```
{
    "version": "2018-05-29",
    "method": "GET",
    "resourcePath": "/v1/apis"
}
```

이 데이터 소스에 대해 GraphQL 쿼리를 실행하면 AWS AppSync는 제공한 역할을 사용하여 요청에 서명하고 서명을 요청에 포함시킵니다. 쿼리는 해당 AWS 리전에서 계정에 있는 AWS AppSync GraphQL API의 목록을 반환합니다.

자습서: 데이터 API를 사용하는 Aurora PostgreSQL

AWS AppSync는 데이터 API로 활성화된 Amazon Aurora 클러스터에 대해 SQL 문을 실행하기 위한 데이터 소스를 제공합니다. AWS AppSync 해석기를 사용하면 GraphQL 쿼리, 변형 및 구독과 관련된 데이터 API에 대해 SQL 문을 실행할 수 있습니다.

> **Note**
>
> 이 자습서에서는 `US-EAST-1` 리전을 사용합니다.

클러스터 생성

AWS AppSync에 Amazon RDS 데이터 소스를 추가하기 전에 먼저 Aurora Serverless 클러스터에서 데이터 API를 활성화합니다. AWS Secrets Manager을 사용하여 암호도 구성해야 합니다. Aurora Serverless 클러스터를 생성하기 위해 AWS CLI를 사용할 수 있습니다.

```
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
```

그러면 클러스터에 대한 ARN이 반환됩니다. 명령으로 클러스터의 상태를 확인할 수 있습니다.

```
aws rds describe-db-clusters \
    --db-cluster-identifier appsync-tutorial \
    --query "DBClusters[0].Status"
```

다음과 같이 이전 단계의 USERNAME 및 COMPLEX_PASSWORD를 사용하는 것처럼 입력 파일로 AWS Secrets Manager 콘솔 또는 AWS CLI를 통해 암호를 생성합니다.

```
{
    "username": "USERNAME",
    "password": "COMPLEX_PASSWORD"
}
```

생성한 암호는 CLI에 파라미터로 전달합니다.

```
aws secretsmanager create-secret \
    --name appsync-tutorial-rds-secret \
    --secret-string file://creds.json
```

그러면 암호에 대한 ARN이 반환됩니다. AWS AppSync 콘솔에 데이터 소스를 생성할 때는 나중에 Aurora Serverless 클러스터의 ARN 및 암호를 적어 둡니다.

데이터 API 활성화

클러스터 상태가 available로 변경되면 Amazon RDS 설명서에 따라 데이터 API를 활성화하십시오. AWS AppSync 데이터 소스로 추가하기 전에 데이터 API를 활성화해야 합니다. AWS CLI를 사용하여 데이터 API를 활성화할 수도 있습니다.

```
aws rds modify-db-cluster \
    --db-cluster-identifier appsync-tutorial \
    --enable-http-endpoint \
    --apply-immediately
```

데이터베이스 및 테이블 생성

데이터 API를 활성화한 후 AWS CLI의 `aws rds-data execute-statement` 명령을 사용하여 작동하는지 확인합니다. 그러면 AWS AppSync API에 추가하기 전에 Aurora Serverless 클러스터가 올바르게 구성되도록 할 수 있습니다. 먼저 다음과 같이 `--sql` 파라미터를 사용하여 TESTDB라는 데이터베이스를 생성합니다.

```
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\""
```

오류 없이 실행되면 `create table` 명령을 사용하여 두 개의 테이블을 추가합니다.

```
 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);'
```

모든 것이 문제 없이 실행되면 API에서 해당 클러스터를 데이터 소스로 추가할 수 있습니다.

GraphQL 스키마 생성

이제 Aurora Serverless 데이터 API가 구성된 테이블을 사용하여 실행되었으므로 GraphQL 스키마를 생성해 보겠습니다. 이 작업은 수동으로 수행할 수 있지만 AWS AppSync를 사용하면 API 생성 마법사를 통해 기존 데이터베이스에서 테이블 구성을 가져와서 빠르게 시작할 수 있습니다.

시작하려면 다음과 같이 하십시오.

1. AWS AppSync 콘솔에서 API 생성을 선택한 다음 Amazon Aurora 클러스터로 시작을 선택합니다.
2. API 이름과 같은 API 세부 정보를 지정한 다음 API를 생성할 데이터베이스를 선택합니다.
3. 데이터베이스 유형을 선택합니다. 필요한 경우 리전을 업데이트한 다음 Aurora 클러스터 및 TESTDB 데이터베이스를 선택합니다.
4. 암호를 선택한 다음 가져오기를 선택합니다.
5. 테이블이 검색되면 유형 이름을 업데이트하십시오. Todos를 Todo로, Tasks를 Task로 변경하십시오.
6. 스키마 미리 보기를 선택하여 생성된 스키마를 미리 볼 수 있습니다. 스키마는 다음과 비슷한 모습이 됩니다.

```
type Todo {
  id: Int!
  description: String!
  due: AWSDate!
  createdAt: String
}

type Task {
  id: Int!
  todoId: Int!
  description: String
}
```

7. 역할의 경우 AWS AppSync에서 새 역할을 생성하거나 아래와 유사한 정책을 사용해 역할을 하나 생성하도록 할 수 있습니다.

```
{
    "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:*"
            ]
        }
    ]
}
```

이 정책에는 역할 액세스 권한을 부여하는 문이 두 개 있습니다. 첫 번째 리소스는 Aurora 클러스터이고 두 번째는 AWS Secrets Manager ARN입니다.

다음을 선택하고 구성 세부 정보를 검토한 다음 API 생성을 선택합니다. API가 이제 완전히 작동합니다. 스키마 페이지에서 API의 전체 세부 정보를 검토할 수 있습니다.

RDS용 해석기

API 생성 흐름에 따라 유형과 상호 작용하는 해석기가 자동으로 생성되었습니다. 스키마 페이지를 보면 다음에 필요한 해석기를 찾을 수 있습니다.

- `Mutation.createTodo` 필드를 통해 `todo`를 생성하십시오.
- `Mutation.updateTodo` 필드를 통해 `todo`를 업데이트하십시오.
- `Mutation.deleteTodo` 필드를 통해 `todo`를 삭제하십시오.
- `Query.getTodo` 필드를 통해 단일 `todo`를 가져오십시오.
- `todos` 필드를 통해 `Query.listTodos`를 모두 나열하십시오.

`Task` 유형에는 유사한 필드와 해석기가 첨부되어 있습니다. 몇 가지 해석기를 좀 더 자세히 살펴보겠습니다.

Mutation.createTodo

AWS AppSync 콘솔의 스키마 편집기 오른쪽에서 `createTodo(...): Todo` 옆에 있는 `testdb`를 선택합니다. 해석기 코드는 `rds` 모듈의 `insert` 함수를 사용하여 `todos` 테이블에 데이터를 추가하는

삽입 문을 동적으로 만듭니다. Postgres를 사용하고 있기 때문에 `returning` 문을 활용하여 삽입된 데이터를 다시 가져올 수 있습니다.

due 필드의 DATE 유형을 올바르게 지정하도록 해석기를 업데이트해 보겠습니다.

```
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]
}
```

해석기를 저장합니다. 유형 힌트는 입력 객체의 due를 올바르게 DATE 유형으로 표시합니다. 이렇게 하면 Postgres 엔진이 값을 올바르게 해석할 수 있습니다. 다음으로 스키마를 업데이트하여 `CreateTodo` 입력에서 `id`를 제거합니다. Postgres 데이터베이스는 생성된 ID를 반환할 수 있으므로 단일 요청으로 이 ID를 사용하여 결과를 생성하고 반환할 수 있습니다.

```
input CreateTodoInput {
    due: AWSDate!
```

```
    createdAt: String
    description: String!
}
```

스키마를 변경하고 업데이트하십시오. 쿼리 편집기로 이동하여 데이터베이스에 항목을 추가합니다.

```
mutation CreateTodo {
  createTodo(input: {description: "Hello World!", due: "2023-12-31"}) {
    id
    due
    description
    createdAt
  }
}
```

결과는 다음과 같습니다.

```
{
  "data": {
    "createTodo": {
      "id": 1,
      "due": "2023-12-31",
      "description": "Hello World!",
      "createdAt": "2023-11-14 20:47:11.875428"
    }
  }
}
```

Query.listTodos

콘솔의 스키마 편집기 오른쪽에서 `listTodos(id: ID!): Todo` 옆에 있는 `testdb`를 선택합니다. 요청 핸들러는 선택 유틸리티 함수를 사용하여 런타임에 요청을 동적으로 작성합니다.

```
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)
}
```

due 날짜를 기준으로 todos를 필터링하려고 합니다. due 값을 DATE로 캐스팅하도록 해석기를 업데이트해 보겠습니다. 가져오기 목록과 요청 핸들러를 업데이트하십시오.

```
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 };
}
```

쿼리를 실행해 보겠습니다. 쿼리 편집기:

```
query LIST {
  listTodos(limit: 10, filter: {due: {between: ["2021-01-01", "2025-01-02"]}}) {
    items {
      id
      due
      description
    }
  }
}
```

Mutation.updateTodo

Todo를 update할 수도 있습니다. 쿼리 편집기에서 id 1의 첫 번째 Todo 항목을 업데이트해 보겠습니다.

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits"}) {
    description
    due
    id
  }
}
```

업데이트하려는 항목의 id를 지정해야 합니다. 특정 조건을 충족하는 항목만 업데이트하도록 조건을 지정할 수도 있습니다. 예를 들어 설명이 edits로 시작하는 경우에만 항목을 편집하고 싶을 수 있습니다.

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits: make a change"}, condition:
 {description: {beginsWith: "edits"}}) {
    description
```

```
    due
    id
  }
}
```

create 및 list 작업을 처리한 방식과 마찬가지로 해석기를 업데이트하여 DATE에 due 필드를 캐스팅할 수 있습니다. 이 변경 사항을 updateTodo에 저장하십시오.

```
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];
}
```

이제 다음과 같은 조건을 적용하여 업데이트를 시도해 보십시오.

```
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

deleteTodo 변형으로 Todo를 delete할 수 있습니다. 이는 updateTodo 변형과 비슷하게 작동하며 삭제하려는 항목의 id를 지정해야 합니다.

```
mutation DELETE {
  deleteTodo(input: {id: 1}) {
    description
    due
    id
  }
}
```

사용자 지정 쿼리 작성

rds 모듈 유틸리티를 사용하여 SQL 문을 생성했습니다. 데이터베이스와 상호 작용하는 사용자 지정 정적 문을 직접 작성할 수도 있습니다. 먼저 스키마를 업데이트하여 CreateTask 입력에서 id 필드를 제거합니다.

```
input CreateTaskInput {
    todoId: Int!
```

```
    description: String
}
```

다음으로, 몇 가지 작업을 만듭니다. 작업에는 Todo와의 외래 키 관계가 있습니다.

```
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 }
}
```

getTodoAndTasks라는 Query 유형에 새 필드를 생성합니다.

```
getTodoAndTasks(id: Int!): Todo
```

Todo 유형에 tasks 필드를 추가합니다.

```
type Todo {
    due: AWSDate!
    id: Int!
    createdAt: String
    description: String!
    tasks:TaskConnection
}
```

스키마를 저장합니다. 콘솔의 스키마 편집기 오른쪽에서 getTodosAndTasks(id: Int!): Todo에 대해 해석기 첨부를 선택합니다. Amazon RDS 데이터 소스를 선택합니다. 다음 코드로 해석기를 업데이트합니다.

```
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;
}
```

이 코드에서는 sql 태그 템플릿을 사용하여 런타임 중에 동적 값을 안전하게 전달할 수 있는 SQL 문을 작성합니다. createPgStatement는 한 번에 최대 두 개의 SQL 요청을 처리할 수 있습니다. 이를 사용하여 하나의 쿼리를 todo에 보내고 다른 쿼리는 tasks에 보냅니다. JOIN 문이나 다른 방법을 사용해도 이 작업을 수행할 수 있었을 것입니다. 요점은 직접 SQL 문을 작성하여 비즈니스 로직을 구현할 수 있게 하는 것입니다. 쿼리 편집기에서 쿼리를 사용하려면 다음과 같이 시도해 볼 수 있습니다.

```
query TodoAndTasks {
  getTodosAndTasks(id: 2) {
    id
    due
    description
    tasks {
      items {
        id
        description
      }
    }
  }
}
```

모든 클러스터 삭제

> **Important**
>
> 클러스터 삭제는 영구적입니다. 이 작업을 수행하기 전에 프로젝트를 철저하게 검토하십시오.

클러스터를 삭제하려면 다음과 같이 하십시오.

```
$ aws rds delete-db-cluster \
    --db-cluster-identifier appsync-tutorial \
    --skip-final-snapshot
```

해석기 자습서(VTL)

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

데이터 소스 및 해석기는 AWS AppSync가 GraphQL 요청을 변환하고 AWS 리소스에서 정보를 가져오는 방법입니다. AWS AppSync는 자동 프로비저닝과 특정 데이터 소스 유형과의 연결을 지원합니다. AWS AppSync는 AWS Lambda, Amazon DynamoDB, 관계형 데이터베이스(Amazon Aurora Serverless), Amazon OpenSearch Service, HTTP 엔드포인트와 같은 데이터 소스를 지원합니다. 기존 AWS 리소스와 함께 GraphQL API를 사용하거나 데이터 소스 및 해석기를 빌드할 수 있습니다. 이 섹션은 작업 세부 사항과 튜닝 옵션을 더 잘 이해할 수 있도록 일련의 자습서에서 이 프로세스를 안내합니다.

AWS AppSync는 해석기를 위해 Apache Velocity 템플릿 언어(VTL)로 작성된 매핑 템플릿을 사용합니다. 매핑 템플릿 사용에 대한 자세한 내용은 해석기 매핑 템플릿 참조를 참조하세요. VTL 사용에 대한 자세한 내용은 해석기 매핑 템플릿 프로그래밍 안내서에서 확인할 수 있습니다.

AWS AppSync는 스키마에서 프로비저닝(선택 사항) 및 샘플 스키마 시작에 설명된 대로 GraphQL 스키마에서 DynamoDB 테이블의 자동 프로비저닝을 지원합니다. 또한 스키마를 생성하고 해석기를 연결할 기존 DynamoDB 테이블에서 가져올 수도 있습니다. 이 부분에 대한 자세한 내용은 Amazon DynamoDB에서 가져오기(선택 사항) 단원을 참조하세요.

주제

- 자습서: DynamoDB 해석기
- 자습서: Lambda 해석기
- 자습서: Amazon OpenSearch Service 해석기
- 자습서: 로컬 해석기
- 튜토리얼: GraphQL 해석기 결합
- 자습서: DynamoDB 배치 해석기
- 자습서: DynamoDB 트랜잭션 해석기
- 자습서: HTTP 해석기

- 자습서: Aurora Serverless
- 자습서: 파이프라인 해석기
- 자습서: 델타 동기화

자습서: DynamoDB 해석기

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

이 자습서에서는 자체 Amazon DynamoDB 테이블을 AWS AppSync로 가져오는 방법과 이러한 테이블을 GraphQL API에 연결하는 방법을 보여줍니다.

AWS AppSync에서 자동으로 DynamoDB 리소스를 프로비저닝하도록 할 수 있습니다. 또는 선호하는 경우 데이터 원본 및 해석기를 생성하여 기존 테이블을 GraphQL 스키마에 연결할 수 있습니다. 두 경우 모두 GraphQL 문을 통해 DynamoDB 데이터베이스를 읽고 여기에 쓸 수 있고 실시간 데이터를 구독할 수 있습니다.

GraphQL 문을 DynamoDB 작업으로 변환하고 응답을 다시 GraphQL로 변환하기 위해서는 순서대로 완료해야 하는 특정한 구성 단계가 있습니다. 이 자습서에서는 여러 가지 실제 시나리오와 데이터 액세스 패턴을 통해 구성 프로세스를 설명합니다.

DynamoDB 테이블 설정

이 자습서를 시작하려면 먼저 아래 단계에 따라 AWS 리소스를 프로비저닝해야 합니다.

1. CLI에서 다음 AWS CloudFormation 템플릿을 사용하여 AWS 리소스를 프로비저닝합니다.

   ```
   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
   ```

   또는 AWS 계정의 미국 서부 2(오레곤) 리전에서 다음 AWS CloudFormation 스택을 시작할 수도 있습니다.

다음을 생성합니다.

- `Post` 데이터를 가지고 있는 `AppSyncTutorial-Post`라는 DynamoDB 테이블
- AWS AppSync가 `Post` 테이블과 상호 작용할 수 있도록 허용하는 IAM 역할 및 연결된 IAM 관리 정책입니다.

2. 스택 및 생성된 리소스에 대해 자세히 알아보려면 다음 CLI 명령을 실행합니다.

```
aws cloudformation describe-stacks --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

3. 나중에 리소스를 삭제하려면 다음을 실행할 수 있습니다.

```
aws cloudformation delete-stack --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

GraphQL API 생성

AWS AppSync에서 GraphQL API를 생성하려면:

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
   - API 대시보드에서 API 생성을 선택합니다.
2. API 사용자 지정 또는 Amazon DynamoDB에서 가져오기 창에서 처음부터 빌드를 선택합니다.
   - 같은 창 오른쪽에 있는 시작을 선택합니다.
3. API 이름 필드에서 API 이름을 `AWSAppSyncTutorial`로 설정합니다.
4. 생성을 선택합니다.

AWS AppSync 콘솔에서 API 키 인증 모드를 사용하여 새 GraphQL API를 생성합니다. 콘솔에서 GraphQL API의 나머지 부분을 설정하고 이 자습서의 나머지 부분에서 이 API에 대한 쿼리를 실행할 수 있습니다.

기본 Post API 정의

AWS AppSync GraphQL API를 생성했으므로 게시 데이터의 기본적인 생성, 검색 및 삭제를 허용하는 기본 스키마를 설정할 수 있습니다.

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
   - API 대시보드에서 방금 만든 API를 선택합니다.
2. 사이드바에서 스키마를 선택합니다.
   - 스키마 창에서 파일 콘텐츠를 다음 코드로 바꿉니다.

```
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. Save를 선택합니다.

이 스키마는 `Post` 객체를 추가하고 가져올 작업과 `Post` 형식을 정의합니다.

DynamoDB 테이블에 대한 데이터 소스 구성

다음으로 스키마에 정의된 쿼리 및 뮤테이션을 `AppSyncTutorial-Post` DynamoDB 테이블에 연결합니다.

먼저, AWS AppSync에서 테이블을 인식해야 합니다. 다음과 같이 AWS AppSync에서 데이터 소스를 설정하여 인식하도록 할 수 있습니다.

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
   a. API 대시보드에서 GraphQL API를 선택합니다.
   b. 사이드바에서 데이터 소스를 선택합니다.
2. [Create data source]를 선택합니다.
   a. 데이터 소스 이름에 `PostDynamoDBTable`을 입력합니다.
   b. 데이터 소스 유형에서 Amazon DynamoDB 테이블을 선택합니다.
   c. 리전에서 US-WEST-2를 선택합니다.
   d. 테이블 이름에서 AppSyncTutorial-Post DynamoDB 테이블을 선택합니다.
   e. 새 IAM 역할을 만들거나(권장) `lambda:invokeFunction` IAM 권한이 있는 기존 역할을 선택합니다. 기존 역할에는 데이터 소스 연결 섹션에 설명된 대로 신뢰 정책이 필요합니다.

      다음은 리소스에서 작업을 수행하는 데 필요한 권한을 가진 IAM 정책의 예입니다.

```
{
     "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. 생성을 선택합니다.

addPost 해석기 설정(DynamoDB PutItem)

AWS AppSync에서 DynamoDB 테이블을 인식하면 해석기를 정의해 이 테이블을 개별 쿼리 및 뮤테이션에 연결할 수 있습니다. 생성한 첫 번째 해석기는 `addPost` 해석기로, 이 해석기를 통해 `AppSyncTutorial-Post` DynamoDB 테이블에서 게시물을 생성할 수 있습니다.

해석기의 구성 요소는 다음과 같습니다.

- 해석기를 연결할 GraphQL 스키마의 위치. 이 경우 `addPost` 형식의 `Mutation` 필드에서 해석기를 설정합니다. 이 해석기는 호출자가 `mutation { addPost(...){...} }`를 호출하면 호출됩니다.
- 이 해석기에 사용할 데이터 원본. 이 경우 앞서 정의한 `PostDynamoDBTable` 데이터 원본을 사용하려고 합니다. 따라서 `AppSyncTutorial-Post` DynamoDB 테이블에 항목을 추가할 수 있습니다.
- 요청 매핑 템플릿. 요청 매핑 템플릿의 용도는 호출자로부터 수신되는 요청을 받아 해당 요청을 AWS AppSync에서 DynamoDB에 대해 수행할 명령으로 변환하는 것입니다.
- 응답 매핑 템플릿. 응답 매핑 템플릿의 작업은 DynamoDB의 응답을 받아 GraphQL에 필요한 것으로 변환하는 것입니다. 이러한 템플릿은 DynamoDB의 데이터 모양과 GraphQL의 `Post` 형식이 다른 경우 유용하지만 이 경우 모양이 서로 동일하기 때문에 데이터를 그냥 전달하면 됩니다.

해석기를 설정하려면:

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
   a. API 대시보드에서 GraphQL API를 선택합니다.
   b. 사이드바에서 데이터 소스를 선택합니다.
2. [Create data source]를 선택합니다.
   a. 데이터 소스 이름에 `PostDynamoDBTable`을 입력합니다.
   b. 데이터 소스 유형에서 Amazon DynamoDB 테이블을 선택합니다.
   c. 리전에서 US-WEST-2를 선택합니다.
   d. 테이블 이름에서 AppSyncTutorial-Post DynamoDB 테이블을 선택합니다.
   e. 새 IAM 역할을 만들거나(권장) `lambda:invokeFunction` IAM 권한이 있는 기존 역할을 선택합니다. 기존 역할에는 데이터 소스 연결 섹션에 설명된 대로 신뢰 정책이 필요합니다.

      다음은 리소스에서 작업을 수행하는 데 필요한 권한을 가진 IAM 정책의 예입니다.

      ```
      {
      ```

```
    "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. 생성을 선택합니다.
4. 스키마 탭을 선택합니다.
5. 오른쪽의 데이터 유형 창의 뮤테이션 유형에서 addPost 필드를 찾은 다음 연결을 선택합니다.
6. 작업 메뉴에서 런타임 업데이트를 선택한 다음 단위 해석기(VTL만 해당)를 선택합니다.
7. Data source name(데이터 원본 이름)에서 PostDynamoDBTable을 선택합니다.
8. 다음을 Configure the request mapping template(요청 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
    "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 }
    }
}
```

참고: 형식은 모든 키와 속성 값에 대해 지정됩니다. 예를 들어, `author` 필드를 `{ "S" : "${context.arguments.author}" }`로 설정합니다. `S` 부분은 값이 문자열 값이 될 것임을 AWS AppSync 및 DynamoDB에 나타냅니다. 실제 값은 `author` 인수에서 채워집니다. 마찬가지

로 `version` 필드는 형식에 `N`을 사용하기 때문에 숫자 필드입니다. 마지막으로, `ups`, `downs` 및 `version` 필드를 초기화합니다.

이 자습서에서는 DynamoDB에 삽입되는 새 항목을 인덱싱하는 GraphQL `ID!` 유형이 클라이언트 인수의 일부로 제공되도록 지정했습니다. AWS AppSync는 `$utils.autoId()`라는 자동 ID 생성 유틸리티와 함께 제공되며, 이 유틸리티를 `"id" : { "S" : "${$utils.autoId()}" }` 형태로도 사용할 수 있습니다. 그런 다음 `id: ID!`의 스키마 정의 외부에 `addPost()`를 둘 수 있는데 자동으로 삽입됩니다. 이 자습서에서는 이러한 기술을 사용하지 않지만 DynamoDB 테이블에 쓰는 경우에는 이 기술의 사용을 고려해야 합니다.

매핑 템플릿에 대한 자세한 내용은 해석기 매핑 템플릿 개요 참조 문서를 참조하십시오. GetItem 요청 매핑에 대한 자세한 내용은 GetItem 참조 문서를 참조하십시오. 행식에 대한 자세한 내용은 형식 시스템(요청 매핑) 참조 문서를 참조하십시오.

9. 다음을 Configure the response mapping template(응답 매핑 템플릿 구성)에 붙여 넣습니다.

```
$utils.toJson($context.result)
```

참고: `AppSyncTutorial-Post` 테이블의 데이터 모양이 GraphQL의 `Post` 형식 모양과 정확하게 일치하기 때문에 응답 매핑 템플릿에서는 결과를 바로 전달하기만 합니다. 또한 파일을 하나만 생성할 수 있도록 이 자습서의 모든 예에서는 동일한 응답 매핑 템플릿을 사용합니다.

10. Save를 선택합니다.

게시물 추가를 위한 API 호출

이제 해석기가 설정되었으므로 AWS AppSync는 들어오는 `addPost` 뮤테이션을 DynamoDB PutItem 작업으로 변환하는 방법을 알고 있습니다. 이제 변형을 실행해 테이블에 데이터를 입력할 수 있습니다.

- Queries 탭을 선택합니다.
- 쿼리 창에 다음 변형을 붙여 넣습니다.

```
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
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- 새로 생성한 게시물의 결과가 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

다음과 같은 작업이 수행되었습니다.

- AWS AppSync에서 addPost 뮤테이션 요청을 수신했습니다.
- AWS AppSync에서 요청 및 요청 매핑 템플릿을 받아 요청 매핑 설명서를 생성했습니다. 이 문서는 다음과 같은 형태입니다.

```
{
    "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 AppSync에서 요청 매핑 설명서를 사용해 DynamoDB `PutItem` 요청을 생성 및 실행했습니다.
- AWS AppSync에서 `PutItem` 요청의 결과를 받아 다시 GraphQL 유형으로 변환했습니다.

```
{
    "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
}
```

- 응답 매핑 문서를 통해 전달했습니다(변경 없이 전달됨).
- GraphQL 응답에서 새로 생성된 객체가 반환되었습니다.

getPost 해석기 설정(DynamoDB GetItem)

이제 `AppSyncTutorial-Post` DynamoDB 테이블에 데이터를 추가할 수 있으므로 `AppSyncTutorial-Post` 테이블에서 데이터를 검색할 수 있도록 `getPost` 쿼리를 설정해야 합니다. 이렇게 하기 위해 다른 해석기를 설정합니다.

- 스키마 탭을 선택합니다.
- 오른쪽의 데이터 유형 창의 쿼리 유형에서 getPost 필드를 찾은 다음 연결을 선택합니다.
- 작업 메뉴에서 런타임 업데이트를 선택한 다음 단위 해석기(VTL만 해당)를 선택합니다.
- Data source name(데이터 원본 이름)에서 PostDynamoDBTable을 선택합니다.
- 다음을 Configure the request mapping template(요청 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

- 다음을 Configure the response mapping template(응답 매핑 템플릿 구성)에 붙여 넣습니다.

```
$utils.toJson($context.result)
```

- Save를 선택합니다.

게시물 가져오기를 위한 API 호출

이제 해석기가 설정되었으므로 AWS AppSync는 들어오는 `getPost` 쿼리를 DynamoDB `GetItem` 작업으로 변환하는 방법을 알고 있습니다. 이제 앞에서 생성한 게시물을 검색하는 쿼리를 실행할 수 있습니다.

- Queries 탭을 선택합니다.
- 쿼리 창에 다음을 붙여 넣습니다.

```
query getPost {
  getPost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- DynamoDB에서 가져온 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

다음과 같은 작업이 수행되었습니다.

- AWS AppSync에서 `getPost` 쿼리 요청을 수신했습니다.
- AWS AppSync에서 요청 및 요청 매핑 템플릿을 받아 요청 매핑 설명서를 생성했습니다. 이 문서는 다음과 같은 형태입니다.

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "123" }
    }
}
```

- AWS AppSync에서 요청 매핑 설명서를 사용해 DynamoDB GetItem 요청을 생성 및 실행했습니다.
- AWS AppSync에서 `GetItem` 요청의 결과를 받아 다시 GraphQL 유형으로 변환했습니다.

```
{
    "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
}
```

- 응답 매핑 문서를 통해 전달했습니다(변경 없이 전달됨).
- 응답에서 가져온 객체가 반환되었습니다.

또는 다음 예를 사용합니다.

```
query getPost {
  getPost(id:123) {
    id
    author
    title
  }
}
```

getPost 쿼리에 id, author, title만 필요한 경우, 요청 매핑 템플릿을 변경하여 프로젝션 표현식을 사용하여 DynamoDB 테이블에서 원하는 속성만 지정하면 DynamoDB에서 AWS AppSync로 불필요한 데이터 전송을 방지할 수 있습니다. 예를 들어 요청 매핑 템플릿은 아래 코드 조각과 비슷할 수 있습니다.

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "projection" : {
     "expression" : "#author, id, title",
     "expressionNames" : { "#author" : "author"}
    }
}
```

updatePost 뮤테이션 생성(DynamoDB UpdateItem)

이제 DynamoDB에서 Post 객체를 생성하고 검색할 수 있습니다. 다음에는, 객체 업데이트할 수 있도록 새 변형을 설정합니다. 이렇게 하기 위해 UpdateItem DynamoDB 작업을 사용합니다.

- 스키마 탭을 선택합니다.

- 스키마 창에서 다음과 같이 `Mutation` 형식을 수정하여 새로운 `updatePost` 변형을 추가할 수 있습니다.

```
type Mutation {
    updatePost(
        id: ID!,
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post
    addPost(
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}
```

- Save를 선택합니다.
- 오른쪽의 데이터 유형 창의 뮤테이션 유형에서 새로 생성한 updatePost 필드를 찾은 다음 연결을 선택합니다.
- 작업 메뉴에서 런타임 업데이트를 선택한 다음 단위 해석기(VTL만 해당)를 선택합니다.
- Data source name(데이터 원본 이름)에서 PostDynamoDBTable을 선택합니다.
- 다음을 Configure the request mapping template(요청 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
    "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 }
        }
    }
}
```

참고: 이 해석기는 PutItem 작업과 크게 다른 DynamoDB UpdateItem 작업을 사용합니다. 전체 항목을 쓰는 대신 특정 속성을 업데이트하도록 DynamoDB에 요청합니다. DynamoDB 업데이트 표현식을 사용하여 이 작업을 완료합니다. 표현식 자체는 `expression` 섹션의 `update` 필드에서 지정합니다. 이 표현식은 `author`, `title`, `content` 및 url 속성을 설정한 다음 `version` 필드를 증분하도록 합니다. 사용되는 값은 표현식 자체에 나타나지 않습니다. 대신 표현식에는 이름이 콜론으로 시작되는 자리 표시자가 있습니다. 표현식은 `expressionValues` 필드에서 정의됩니다. 마지막으로, DynamoDB에는 `expression`에 표시할 수 없는 예약어가 있습니다. 예를 들어, `url`은 예약어기 때문에 `url` 필드를 업데이트하려면 이름 자리 표시자를 사용해 `expressionNames` 필드에 해당 자리 표시자를 정의합니다.

`UpdateItem` 요청 매핑에 대한 자세한 내용은 UpdateItem 참조 문서를 참조하십시오. 업데이트 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB UpdateExpressions 문서를 참조하십시오.

- 다음을 Configure the response mapping template(응답 매핑 템플릿 구성)에 붙여 넣습니다.

```
$utils.toJson($context.result)
```

게시물 업데이트를 위한 API 호출

이제 해석기가 설정되었으므로 AWS AppSync는 들어오는 `update` 뮤테이션을 DynamoDB `Update` 작업으로 변환하는 방법을 알고 있습니다. 이제 앞서 작성한 항목을 업데이트하는 변형을 실행할 수 있습니다.

- Queries 탭을 선택합니다.
- 쿼리 창에 다음 변형을 붙여 넣습니다. 또한 앞서 적어둔 값을 갖도록 `id` 인수를 업데이트해야 합니다.

```
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
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- DynamoDB에서 업데이트된 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

이 예제에서는 요청 매핑 템플릿이 AWS AppSync 및 DynamoDB에 이러한 필드에 대해 어떠한 작업을 수행하도록 요청하지 않았기 때문에 ups 및 downs 필드가 수정되지 않았습니다. 또한 AWS AppSync 및 DynamoDB에 version 필드에 1을 추가하도록 요청했기 때문에 version 필드가 1씩 증가했습니다.

updatePost 해석기 수정(DynamoDB UpdateItem)

이 작업은 updatePost 변형을 시작하기 좋은 지점이지만 다음과 같이 두 가지 주요한 문제가 있습니다.

- 필드를 하나만 업데이트하고자 하는 경우에도 필드를 모두 업데이트해야 합니다.
- 두 사람이 객체를 수정하는 경우 정보가 손실될 수 있습니다.

이러한 문제를 해결하기 위해 요청에 지정된 인수만 수정한 다음 UpdateItem 작업에 조건을 추가하도록 updatePost 변형을 수정하려고 합니다.

1. 스키마 탭을 선택합니다.
2. 스키마 패널에서 Mutation 유형의 updatePost 필드를 수정하여 author, title, content 및 url 인수에서 느낌표를 제거하고 id 필드는 그대로 둡니다. 이렇게 하면 해당 인수가 선택적 인수가 됩니다. 또한 새로운 필수 인수인 expectedVersion 버전을 추가합니다.

```
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. Save를 선택합니다.
4. 오른쪽의 데이터 유형 창의 뮤테이션 유형에서 updatePost 필드를 찾습니다.
5. PostDynamoDBTable을 선택하여 기존 해석기를 엽니다.
6. Configure the request mapping template(요청 매핑 템플릿 구성)에서 다음과 같이 요청 매핑 템플릿을 수정합니다.

```
{
    "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. Save를 선택합니다.

이 템플릿은 보다 복잡한 예 중 하나입니다. 이 템플릿은 보다 복잡한 예 중 하나로, 매핑 템플릿의 이점과 유연성을 보여줍니다. 이 템플릿은 모든 인수를 루핑하면서 `id`와 `expectedVersion`을 건너뜁니다. 인수가 어떤 값으로 설정되어 있는 경우 이 템플릿은 AWS AppSync 및 DynamoDB에 DynamoDB의 객체에 대한 속성을 업데이트하도록 요청합니다. 속성이 null로 설정되어 있는 경우 이 템플릿은 AWS AppSync 및 DynamoDB에 게시물 객체에서 속성을 제거하도록 요청합니다. 인수가 지정되어 있지 않은 경우에는 속성을 그냥 둡니다. 이 템플릿 역시 `version` 필드를 증가시킵니다.

또한 새로운 `condition` 섹션이 있습니다. 조건 표현식을 사용하면 작업을 수행하기 전에 DynamoDB에 이미 있는 객체의 상태를 기반으로 AWS AppSync 및 DynamoDB에 요청에 성공해야 할지 여부를 지시할 수 있습니다. 이 경우에는 DynamoDB에 현재 있는 항목의 `version` 필드가 `expectedVersion` 인수와 정확하게 일치하는 경우에만 `UpdateItem` 요청에 성공하면 됩니다.

조건 표현식에 대한 자세한 내용은 조건 표현식 참조 문서를 참조하십시오.

게시물 업데이트를 위한 API 호출

새 해석기를 사용해 `Post` 객체를 업데이트해 보십시오.

- Queries 탭을 선택합니다.
- 쿼리 창에 다음 변형을 붙여 넣습니다. 또한 앞서 적어둔 값을 갖도록 `id` 인수를 업데이트해야 합니다.

```
mutation updatePost {
  updatePost(
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 2
  ) {
    id
```

```
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- DynamoDB에서 업데이트된 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

이 요청에서는 AWS AppSync 및 DynamoDB에 `title` 및 `content` 필드만 업데이트하도록 요청했습니다. `version` 필드를 증분하는 것을 제외하고 다른 모든 필드는 그대로 둡니다. `title` 속성을 새 값으로 설정하고 게시물에서 `content` 속성을 제거했습니다. `author`, `url`, `ups` 및 `downs` 필드는 그대로 남아 있습니다.

변형 요청을 다시 실행해도 요청은 정확하게 그대로 유지됩니다. 다음과 유사한 응답이 나타납니다.

```
{
  "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)"
    }
  ]
}
```

조건 표현식이 false로 평가되었으므로 요청에 실패합니다.

- 요청을 처음 실행했을 때 DynamoDB에 있는 게시물의 `version` 필드 값은 2였고, 이는 `expectedVersion` 인수와 일치했습니다. 요청에 성공하자 DynamoDB의 `version` 필드가 3으로 증가했습니다.
- 요청을 두 번째로 실행했을 때 DynamoDB에 있는 게시물의 `version` 필드 값은 3이었고, 이는 `expectedVersion` 인수와 일치하지 않았습니다.

이러한 패턴을 일반적으로 낙관적 잠금이라고 합니다.

AWS AppSync DynamoDB 해석기의 기능은 DynamoDB에 있는 게시물 객체의 현재 값을 반환하는 것입니다. 현재 값은 GraphQL 응답의 `data` 섹션에 있는 `errors` 필드에서 찾을 수 있습니다. 애플리케이션에서는 이러한 정보를 사용하여 처리 방식을 결정합니다. 이 경우, DynamoDB에 있는 객체의

version 필드가 3으로 설정되어 있은데, 방금 expectedVersion 인수를 3으로 업데이트했으므로 결과는 다시 성공입니다.

조건 검사 실패 처리에 대한 자세한 내용은 조건 표현식 매핑 템플릿 참조 문서를 참조하십시오.

upvotePost 및 downvotePost 뮤테이션 생성(DynamoDB UpdateItem)

Post 형식에는 ups 및 downs 필드가 있어 추천 수 및 비추천 수를 기록할 수 있습니다. 하지만 지금까지 API를 사용하면 이러한 작업을 수행할 수 없었습니다. 게시물을 추천 및 비추천하도록 하는 몇 가지 변형을 추가해 보겠습니다.

- 스키마 탭을 선택합니다.
- 스키마 창에서 다음과 같이 Mutation 유형을 수정하여 새로운 upvotePost 및 downvotePost 뮤테이션을 추가할 수 있습니다.

```
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!
}
```

- Save를 선택합니다.
- 오른쪽의 데이터 유형 창의 뮤테이션 유형에서 새로 생성한 upvotePost 필드를 찾은 다음 연결을 선택합니다.
- 작업 메뉴에서 런타임 업데이트를 선택한 다음 단위 해석기(VTL만 해당)를 선택합니다.
- Data source name(데이터 원본 이름)에서 PostDynamoDBTable을 선택합니다.
- 다음을 Configure the request mapping template(요청 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
    "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 }
        }
    }
}
```

- 다음을 Configure the response mapping template(응답 매핑 템플릿 구성)에 붙여 넣습니다.

```
$utils.toJson($context.result)
```

- Save를 선택합니다.
- 오른쪽의 데이터 유형 창의 뮤테이션 유형에서 새로 생성한 downvotePost 필드를 찾은 다음 연결을 선택합니다.
- Data source name(데이터 원본 이름)에서 PostDynamoDBTable을 선택합니다.
- 다음을 Configure the request mapping template(요청 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
    "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 }
        }
    }
}
```

- 다음을 Configure the response mapping template(응답 매핑 템플릿 구성)에 붙여 넣습니다.

```
$utils.toJson($context.result)
```

- Save를 선택합니다.

게시물을 추천 및 비추천하는 API 호출

이제 새 해석기가 설정되었으므로 AWS AppSync는 들어오는 upvotePost 또는 downvote 뮤테이션을 DynamoDB UpdateItem 작업으로 변환하는 방법을 알고 있습니다. 이제 앞서 생성한 게시물을 추천 또는 비추천하는 변형을 실행할 수 있습니다.

- Queries 탭을 선택합니다.
- 쿼리 창에 다음 변형을 붙여 넣습니다. 또한 앞서 적어둔 값을 갖도록 id 인수를 업데이트해야 합니다.

```
mutation votePost {
  upvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- 게시물이 DynamoDB에서 업데이트되고 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

- Execute query(쿼리 실행)를 몇 번 더 선택합니다. 쿼리를 실행할 때마다 ups 및 version 필드가 1씩 증가하는 것이 보여야 합니다.
- 다음과 같이 downvotePost 변형을 호출하도록 쿼리를 변경합니다.

```
mutation votePost {
  downvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다. 쿼리를 실행할 때마다 downs 및 version 필드가 1씩 증가하는 것이 보여야 합니다.

```
{
  "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
    }
  }
}
```

deletePost 해석기 설정(DynamoDB DeleteItem)

설정하려는 다음 변형은 게시물을 삭제하는 변형입니다. 이 작업은 `DeleteItem` DynamoDB 작업을 사용하여 수행합니다.

- 스키마 탭을 선택합니다.
- 스키마 창에서 다음과 같이 `Mutation` 형식을 수정하여 새로운 `deletePost` 변형을 추가할 수 있습니다.

```
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!
}
```

  이번에는`expectedVersion` 필드를 선택 사항으로 설정했는데, 이에 관한 사항은 나중에 요청 매핑 템플릿을 추가할 때 설명하겠습니다.
- Save를 선택합니다.
- 오른쪽의 데이터 유형 창의 뮤테이션 유형에서 새로 생성한 delete 필드를 찾은 다음 연결을 선택합니다.
- 작업 메뉴에서 런타임 업데이트를 선택한 다음 단위 해석기(VTL만 해당)를 선택합니다.
- Data source name(데이터 원본 이름)에서 PostDynamoDBTable을 선택합니다.
- 다음을 Configure the request mapping template(요청 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
```

```
    "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
}
```

참고: `expectedVersion` 인수는 선택적 인수입니다. 호출자가 요청에서 `expectedVersion` 인수를 설정하면, 템플릿에서는 항목이 이미 삭제된 경우 또는 DynamoDB에 있는 `version` 속성이 `expectedVersion`과 정확하게 일치하는 경우, `DeleteItem` 요청이 완료되게 하는 조건을 추가합니다. 그냥 두면 `DeleteItem` 요청에 아무런 조건 표현식도 지정되지 않습니다. 이는 `version`의 값이나 해당 항목이 DynamoDB에 존재하는지 여부에 관계없이 완료됩니다.

- 다음을 Configure the response mapping template(응답 매핑 템플릿 구성)에 붙여 넣습니다.

```
$utils.toJson($context.result)
```

참고: 항목을 삭제하고 있더라도 항목이 아직 삭제되지 않은 경우 삭제된 항목을 반환할 수 있습니다.

- Save를 선택합니다.

`DeleteItem` 요청 매핑에 대한 자세한 내용은 DeleteItem 참조 문서를 참조하십시오.

게시물 삭제를 위한 API 호출

이제 해석기가 설정되었으므로 AWS AppSync는 들어오는 `delete` 뮤테이션을 DynamoDB `DeleteItem` 작업으로 변환하는 방법을 알고 있습니다. 이제 변형을 실행해 테이블에 데이터를 삭제할 수 있습니다.

- Queries 탭을 선택합니다.

- 쿼리 창에 다음 변형을 붙여 넣습니다. 또한 앞서 적어둔 값을 갖도록 id 인수를 업데이트해야 합니다.

```
mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- DynamoDB에서 게시물이 삭제되었습니다. AWS AppSync에서는 DynamoDB에서 삭제된 항목의 값을 반환합니다. 해당 값은 쿼리 창 오른쪽에 있는 결과 창에 표시되어야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

deletePost에 대한 호출이 DynamoDB에서 실제로 삭제된 항목인 경우에만 값이 반환됩니다.

- Execute query(쿼리 실행)을 다시 선택합니다.
- 호출에는 계속 성공하지만 반환되는 값은 없습니다.

```
{
  "data": {
    "deletePost": null
  }
}
```

이제 게시물을 삭제해 보는데, 이번에는 `expectedValue`를 지정해 보겠습니다. 지금까지 작업한 게시물 하나를 방금 삭제했기 때문에 먼저 새 게시물을 생성해야 합니다.

- 쿼리 창에 다음 변형을 붙여 넣습니다.

```
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
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- 새로 생성한 게시물의 결과가 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 곧 필요할 것이기 때문에 새로 생성한 객체의 `id`를 적어 둡니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

이제 게시물을 삭제해 보는데 expectedVersion에 대해 잘못된 값을 입력할 것입니다.

- 쿼리 창에 다음 변형을 붙여 넣습니다. 또한 앞서 적어둔 값을 갖도록 id 인수를 업데이트해야 합니다.

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.

```
{
  "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)"
    }
  ]
}
```

DynamoDB에 있는 게시물의 `version` 값이 인수에 지정된 `expectedValue`와 일치하지 않아 조건 표현식이 false로 평가되었기 때문에 요청에 실패했습니다. 객체의 현재 값은 GraphQL 응답에서 `data` 섹션의 `errors` 필드에 반환됩니다.

- 이 요청을 다시 실행하되 `expectedVersion`을 수정합니다.

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
```

```
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- 이번에는 요청에 성공하고 DynamoDB에서 삭제된 값이 반환됩니다.

```
{
  "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
    }
  }
}
```

- Execute query(쿼리 실행)을 다시 선택합니다.
- 호출에 계속해서 성공하지만 이번에는 게시물이 이미 DynamoDB에서 삭제되었으므로 반환되는 값이 없습니다.

```
{
  "data": {
    "deletePost": null
  }
}
```

allPost 해석기 설정(DynamoDB Scan)

지금까지 API는 살펴보고자 하는 각 게시물의 `id`를 아는 경우에만 유용했습니다. 테이블의 게시물을 모두 반환하는 새 해석기를 추가해 보겠습니다.

- 스키마 탭을 선택합니다.
- 스키마 창에서 다음과 같이 `Query` 형식을 수정하여 새로운 `allPost` 쿼리를 추가할 수 있습니다.

```
type Query {
```

```
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

- 새로운 PaginationPosts 형식을 다음과 같이 추가합니다.

```
type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}
```

- Save를 선택합니다.
- 오른쪽의 데이터 유형 창의 쿼리 유형에서 새로 생성한 allPost 필드를 찾은 다음 연결을 선택합니다.
- 작업 메뉴에서 런타임 업데이트를 선택한 다음 단위 해석기(VTL만 해당)를 선택합니다.
- Data source name(데이터 원본 이름)에서 PostDynamoDBTable을 선택합니다.
- 다음을 Configure the request mapping template(요청 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
    "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
}
```

이 해석기에는 두 가지 선택적 인수가 있는데, count는 단일 호출 시 반환되는 최대 항목 수를 지정하고, nextToken은 다음 결과 집합을 가져오는 데 사용할 수 있습니다(nextToken의 값을 가져오는 위치는 나중에 설명할 예정임).

- 다음을 Configure the response mapping template(응답 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

참고: 이 응답 매핑 템플릿은 지금까지 사용한 다른 모든 템플릿과 다릅니다. `allPost` 쿼리의 결과는 게시물 목록과 페이지 매김 토큰을 포함하고 있는 `PaginatedPosts`입니다. 이 객체의 모양은 AWS AppSync DynamoDB 해석기에서 반환된 값과 다릅니다. 게시물 목록은 AWS AppSync DynamoDB 해석기 결과에서는 `items`라고 하지만 `PaginatedPosts`에서는 `posts`라고 합니다.

- Save를 선택합니다.

`Scan` 요청 매핑에 대한 자세한 내용은 스캔 참조 문서를 참조하십시오.

모든 게시물을 스캔하는 API 호출

이제 해석기가 설정되었으므로 AWS AppSync는 들어오는 `allPost` 쿼리를 DynamoDB `Scan` 작업으로 변환하는 방법을 알고 있습니다. 이제 테이블을 스캔해 게시물을 모두 검색할 수 있습니다.

지금까지 작업했던 데이터를 모두 삭제했기 때문에 이 요청을 수행하기 전에 몇 가지 데이터로 테이블을 채워야 합니다.

- Queries 탭을 선택합니다.
- 쿼리 창에 다음 변형을 붙여 넣습니다.

```
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 }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.

이제 테이블을 스캔해 보겠습니다. 한 번에 5개 결과를 반환합니다.

- 쿼리 창에 다음 쿼리를 붙여 넣습니다.

```
query allPost {
  allPost(count: 5) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- 처음 5개 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

5개 결과가 반환되고 다음 결과 집합을 가져오는데 사용할 수 있는 nextToken도 있습니다.

- 이전 결과 집합에서 allPost을 포함하도록 nextToken 쿼리를 업데이트합니다.

```
query allPost {
  allPost(
    count: 5
    nextToken:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
  ) {
    posts {
      id
      author
    }
    nextToken
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- 나머지 4개 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 게시물이 모두 9개이며 남은 게시물이 없기 때문에 이 결과 집합에는 nextToken이 없습니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

allPostsByAuthor 해석기 설정(DynamoDB Query)

DynamoDB에서 모든 게시물을 스캔하는 것 이외에 DynamoDB를 쿼리해 특정 작성자가 생성한 게시물을 검색할 수 있습니다. 앞서 이미 생성한 DynamoDB 테이블에는 특정 작성자가 생성한 게시물을 모두 검색하는 DynamoDB `Query` 작업에 사용할 수 있는 `author-index`라는 `GlobalSecondaryIndex`가 있습니다.

- 스키마 탭을 선택합니다.
- 스키마 창에서 다음과 같이 `Query` 형식을 수정하여 새로운 `allPostsByAuthor` 쿼리를 추가할 수 있습니다.

```
type Query {
    allPostsByAuthor(author: String!, count: Int, nextToken: String): PaginatedPosts!
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

  참고: 여기서는 `allPost` 쿼리에 사용한 것과 동일한 `PaginatedPosts` 형식을 사용합니다.
- Save를 선택합니다.
- 오른쪽의 데이터 유형 창의 쿼리 유형에서 새로 생성한 allPostsByAuthor 필드를 찾은 다음 연결을 선택합니다.
- 작업 메뉴에서 런타임 업데이트를 선택한 다음 단위 해석기(VTL만 해당)를 선택합니다.

- Data source name(데이터 원본 이름)에서 PostDynamoDBTable을 선택합니다.
- 다음을 Configure the request mapping template(요청 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
    "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
}
```

`allPost` 해석기처럼 이 해석기에는 두 가지 선택적 인수가 있는데, `count`는 단일 호출 시 반환되는 최대 항목 수를 지정하고, `nextToken`은 다음 결과 집합을 가져오는 데 사용할 수 있습니다(`nextToken`의 값은 이전 호출에서 얻을 수 있음).

- 다음을 Configure the response mapping template(응답 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

참고: 이는 `allPost` 해석기에서 사용한 것과 동일한 응답 매핑 템플릿입니다.

- Save를 선택합니다.

`Query` 요청 매핑에 대한 자세한 내용은 쿼리 참조 문서를 참조하십시오.

작성자별 모든 게시물을 쿼리하는 API 호출

이제 해석기가 설정되었으므로 AWS AppSync는 들어오는 `allPostsByAuthor` 뮤테이션을 `author-index` 인덱스에 대한 DynamoDB `Query` 작업으로 변환하는 방법을 알고 있습니다. 이제 테이블을 쿼리해 특정 작성자별로 게시물을 모두 검색할 수 있습니다.

지금까지 사용한 모든 게시물은 작성자가 동일하기 때문에 검색하기 전에 추가 게시물로 테이블을 채우겠습니다.

- Queries 탭을 선택합니다.
- 쿼리 창에 다음 변형을 붙여 넣습니다.

```
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 }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.

이제 `Nadia`가 작성한 게시물을 모두 반환하도록 테이블을 쿼리하겠습니다.

- 쿼리 창에 다음 쿼리를 붙여 넣습니다.

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- `Nadia`가 작성한 모든 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

Query에 대한 페이지 매김은 Scan을 수행할 때와 동일하게 작동합니다. 예를 들어, AUTHORNAME의 모든 게시물을 살펴보겠습니다. 이 경우 한 번에 5개 결과를 반환합니다.

- 쿼리 창에 다음 쿼리를 붙여 넣습니다.

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- AUTHORNAME가 작성한 모든 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

- 다음과 같이 이전 쿼리에서 반환된 값으로 nextToken 인수를 업데이트합니다.

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
    nextToken:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
  ) {
    posts {
      id
```

```
      title
    }
    nextToken
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- AUTHORNAME가 작성한 잔여 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "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
    }
  }
}
```

집합 사용

지금까지 Post 형식은 플랫 키/값 객체였습니다. 또한 AWS AppSync DynamoDB 해석기를 사용하여 집합, 목록 및 맵 등과 같은 복잡한 객체를 모델링할 수 있습니다.

태그를 포함하도록 Post 형식을 업데이트해 보겠습니다. 게시물에는 태그가 0개 이상 있을 수 있는데, 태그는 DynamoDB에 문자열 집합으로 저장됩니다. 또한 태그를 추가 및 제거하는 몇 가지 변형과 특정 태그가 지정된 게시물을 스캔하는 새 쿼리를 설정해 볼 것입니다.

- 스키마 탭을 선택합니다.
- 스키마 창에서 다음과 같이 Post 형식을 수정하여 새로운 tags 필드를 추가할 수 있습니다.

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
}
```

- 스키마 창에서 다음과 같이 Query 형식을 수정하여 새로운 allPostsByTag 쿼리를 추가할 수 있습니다.

```
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
}
```

- 스키마 창에서 다음과 같이 Mutation 유형을 수정하여 새로운 addTag 및 removeTag 뮤테이션을 추가할 수 있습니다.

```
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!
}
```

- Save를 선택합니다.
- 오른쪽의 데이터 유형 창의 쿼리 유형에서 새로 생성한 allPostsByTag 필드를 찾은 다음 연결을 선택합니다.
- Data source name(데이터 원본 이름)에서 PostDynamoDBTable을 선택합니다.
- 다음을 Configure the request mapping template(요청 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
    "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
}
```

- 다음을 Configure the response mapping template(응답 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
```

```
}
```

- Save를 선택합니다.
- 오른쪽의 데이터 유형 창의 뮤테이션 유형에서 새로 생성한 addTag 필드를 찾은 다음 연결을 선택합니다.
- Data source name(데이터 원본 이름)에서 PostDynamoDBTable을 선택합니다.
- 다음을 Configure the request mapping template(요청 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
    "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 }
        }
    }
}
```

- 다음을 Configure the response mapping template(응답 매핑 템플릿 구성)에 붙여 넣습니다.

```
$utils.toJson($context.result)
```

- Save를 선택합니다.
- 오른쪽의 데이터 유형 창의 뮤테이션 유형에서 새로 생성한 removeTag 필드를 찾은 다음 연결을 선택합니다.
- Data source name(데이터 원본 이름)에서 PostDynamoDBTable을 선택합니다.
- 다음을 Configure the request mapping template(요청 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
    "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 }
        }
    }
}
```

- 다음을 Configure the response mapping template(응답 매핑 템플릿 구성)에 붙여 넣습니다.

```
$utils.toJson($context.result)
```

- Save를 선택합니다.

태그를 사용하는 API 호출

이제 해석기를 설정했으므로, AWS AppSync에서는 수신되는 `addTag`, `removeTag` 및 `allPostsByTag` 요청을 DynamoDB `UpdateItem` 및 `Scan` 작업으로 변환하는 방법을 알고 있습니다.

실행해 보기 위해 이전에 생성한 게시물 중 선택해 보겠습니다. 예를 들어, `Nadia`에서 작성한 게시물을 사용해 보겠습니다.

- Queries 탭을 선택합니다.
- 쿼리 창에 다음 쿼리를 붙여 넣습니다.

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- Nadia의 모든 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

- 제목이 "The cutest dog in the world"인 게시물을 사용하겠습니다. 나중에 사용할 것이기 때문에 이 게시물의 id를 적어 둡니다.

dog 태그를 추가합니다.

- 쿼리 창에 다음 변형을 붙여 넣습니다. 또한 앞서 적어둔 값을 갖도록 id 인수를 업데이트해야 합니다.

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- 새 태그로 게시물이 업데이트됩니다.

```
{
  "data": {
    "addTag": {
      "id": "10",
```

```
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

태그를 다음과 같이 더 추가할 수 있습니다.

- tag 인수를 puppy로 변경하도록 변형을 업데이트합니다.

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- 새 태그로 게시물이 업데이트됩니다.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

태그를 삭제할 수도 있습니다.

- 쿼리 창에 다음 변형을 붙여 넣습니다. 또한 앞서 적어둔 값을 갖도록 id 인수를 업데이트해야 합니다.

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- 게시물이 업데이트되고 puppy 태그가 삭제됩니다.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

또한 태그가 지정된 게시물을 모두 검색할 수도 있습니다.

- 쿼리 창에 다음 쿼리를 붙여 넣습니다.

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- dog 태그가 지정된 게시물이 다음과 같이 모두 반환됩니다.

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
            "dog",
            "puppy"
          ]
        }
      ],
      "nextToken": null
    }
  }
}
```

목록 및 맵 사용

DynamoDB 집합을 사용하는 것 이외에 DynamoDB 목록 및 맵을 사용하여 단일 객체 내에서 복잡한 데이터를 모델링할 수 있습니다.

게시물에 주석을 추가하는 기능을 추가해 보겠습니다. 이 기능은 DynamoDB의 `Post` 객체에 대한 맵 객체 목록으로 모델링됩니다.

참고: 실제 애플리케이션에서는 설명을 고유의 테이블에 모델링하게 될 것입니다. 이 자습서에서는 설명을 `Post` 테이블에 추가하겠습니다.

- 스키마 탭을 선택합니다.
- 스키마 창에서 다음과 같이 새로운 `Comment` 형식을 추가합니다.

```
type Comment {
    author: String!
    comment: String!
}
```

- 스키마 창에서 다음과 같이 `Post` 형식을 수정하여 새로운 `comments` 필드를 추가할 수 있습니다.

```
type Post {
```

```
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
  comments: [Comment!]
}
```

- 스키마 창에서 다음과 같이 Mutation 형식을 수정하여 새로운 addComment 변형을 추가할 수 있습니다.

```
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!
}
```

- Save를 선택합니다.
- 오른쪽의 데이터 유형 창의 뮤테이션 유형에서 새로 생성한 addComment 필드를 찾은 다음 연결을 선택합니다.
- Data source name(데이터 원본 이름)에서 PostDynamoDBTable을 선택합니다.

- 다음을 Configure the request mapping template(요청 매핑 템플릿 구성)에 붙여 넣습니다.

```
{
  "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)
    }
  }
}
```

  이 업데이트 표현식은 새 주석이 포함된 목록을 기존 `comments` 목록에 추가합니다. 목록이 아직 없으면 생성됩니다.

- 다음을 Configure the response mapping template(응답 매핑 템플릿 구성)에 붙여 넣습니다.

```
$utils.toJson($context.result)
```

- Save를 선택합니다.

설명 추가를 위한 API 호출

이제 해석기를 설정했으므로, AWS AppSync에서는 수신되는 `addComment` 요청을 DynamoDB `UpdateItem` 작업으로 변환하는 방법을 알고 있습니다.

태그를 추가한 것과 동일한 게시물에 주석을 추가해 보겠습니다.

- Queries 탭을 선택합니다.

- 쿼리 창에 다음 쿼리를 붙여 넣습니다.

```
mutation addComment {
  addComment(
    id:10
    author: "Steve"
    comment: "Such a cute dog."
  ) {
    id
    comments {
      author
      comment
    }
  }
}
```

- Execute query(쿼리 실행)(주황색 재생 버튼)를 누릅니다.
- Nadia의 모든 게시물이 쿼리 창 오른쪽에 있는 결과 창에 나타나야 합니다. 예를 들면 다음과 같아야 합니다.

```
{
  "data": {
    "addComment": {
      "id": "10",
      "comments": [
        {
          "author": "Steve",
          "comment": "Such a cute dog."
        }
      ]
    }
  }
}
```

이 요청을 여러 번 실행하면 목록에 주석이 여러 개 추가됩니다.

결론

이 자습서에서는 AWS AppSync 및 GraphQL을 사용하여 DynamoDB의 Post 객체를 조작할 수 있는 API를 구축했습니다. 자세한 내용은 해석기 매핑 템플릿 참조 단원을 참조하십시오.

정리를 위해 콘솔에서 AppSync GraphQL API를 삭제할 수 있습니다.

이 자습서를 위해 생성한 DynamoDB 테이블 및 IAM 역할을 삭제하려면 다음을 실행하여 `AWSAppSyncTutorialForAmazonDynamoDB` 스택을 삭제하거나 AWS CloudFormation 콘솔을 방문하여 스택을 삭제할 수도 있습니다.

```
aws cloudformation delete-stack \
    --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

자습서: Lambda 해석기

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync와 함께 AWS Lambda를 사용하여 모든 GraphQL 필드를 확인할 수 있습니다. 예를 들어, GraphQL 쿼리는 Amazon Relational Database Service(Amazon RDS) 인스턴스로 호출을 보내고, GraphQL 뮤테이션은 Amazon Kinesis 스트림에 쓸 수 있습니다. 이 단원에서는 GraphQL 필드 작업의 호출에 따라 비즈니스 로직을 수행하는 Lambda 함수의 작성 방법을 설명합니다.

Lambda 함수 생성

다음 예는 블로그 게시물 애플리케이션의 일부로 블로그 게시물에 대해 다양한 연산을 수행하는 `Node.js`로 작성된 Lambda 함수를 보여줍니다.

```
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;
    }
};
```

Lambda 함수는 ID별로 게시물을 검색하고, 게시물을 추가하고, 게시물 목록을 가져오고, 지정된 게시물에 대해 관련 게시물을 가져오는 작업을 처리합니다.

참고: `event.field`에 대해 `switch` 명령문을 사용하면 Lambda 함수가 현재 해석 중인 필드를 확인할 수 있습니다.

AWS 관리 콘솔 또는 AWS CloudFormation 스택을 사용하여 이 Lambda 함수를 생성합니다. CloudFormation 스택에서 함수를 만들려면 다음 AWS Command Line Interface(AWS CLI) 명령을 사용할 수 있습니다.

```
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
```

여기에서 AWS 계정의 미국 서부(오레곤) AWS 리전에서 AWS CloudFormation 스택을 시작할 수도 있습니다.

Lambda용 데이터 소스 구성

Lambda 함수를 만든 후 AWS AppSync 콘솔에서 GraphQL API로 이동한 다음 데이터 소스 탭을 선택합니다.

데이터 소스 생성을 선택하고 친숙한 데이터 소스 이름(예: **Lambda**)을 입력한 다음 데이터 소스 유형에서 AWS Lambda 함수를 선택합니다. 리전에서 함수와 동일한 리전을 선택합니다. (제공된 CloudFormation 스택에서 함수를 생성했다면 해당 함수는 US-WEST-2에 있을 것입니다.) 함수 ARN의 경우, Lambda 함수의 Amazon Resource Name(ARN)을 선택합니다.

Lambda 함수를 선택한 후 새 AWS Identity and Access Management(IAM) 역할을 만들거나(AWS AppSync에서 적절한 권한을 할당) 다음 인라인 정책이 있는 기존 역할을 선택할 수 있습니다.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/LAMBDA_FUNCTION"
        }
    ]
}
```

또한 다음과 같이 IAM 역할에 대해 AWS AppSync와 신뢰 관계를 설정해야 합니다:

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

GraphQL 스키마 생성

이제 데이터 소스가 Lambda 함수에 연결되었으며 GraphQL 스키마를 생성해 보겠습니다.

AWS AppSync 콘솔의 스키마 편집기에서 스키마가 다음 스키마와 일치하는지 확인합니다.

```
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]
}
```

해석기 구성

이제 Lambda 데이터 소스와 유효한 GraphQL 스키마를 등록했으며, 해석기를 사용하여 GraphQL 필드를 Lambda 데이터 소스에 연결할 수 있습니다.

해석기를 생성하려면 매핑 템플릿이 필요합니다. 매핑 템플릿에 대한 자세히 알아보려면 Resolver Mapping Template Overview 단원을 참조하세요.

Lambda 매핑 템플릿에 대한 자세한 내용은 Resolver mapping template reference for Lambda 단원을 참조하세요.

이 단계에서는 `getPost(id:ID!): Post`, `allPosts: [Post]`, `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`, `Post.relatedPosts: [Post]` 필드에 대한 해석기를 Lambda 함수에 연결합니다.

AWS AppSync 콘솔의 스키마 편집기에서 오른쪽에 있는 `getPost(id:ID!): Post`용 해석기 연결을 선택합니다.

그런 다음 작업 메뉴에서 런타임 업데이트를 선택한 다음 단위 해석기(VTL만 해당)를 선택합니다.

그런 다음 Lambda 데이터 소스를 선택합니다. request mapping template(요청 매핑 템플릿) 섹션에서 Invoke And Forward Arguments(인수 호출 및 전달)을 선택합니다.

`payload` 객체를 수정하여 필드 이름을 추가합니다. 템플릿은 다음과 같아야 합니다.

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments":  $utils.toJson($context.arguments)
    }
}
```

response mapping template(요청 매핑 템플릿) 섹션에서 Return Lambda Result(Lambda 결과 반환)를 선택합니다.

이 경우에는 기본 템플릿을 있는 그대로 사용하겠습니다. 이 템플릿은 다음과 같습니다.

```
$utils.toJson($context.result)
```

Save를 선택합니다. 이제 첫 번째 해석기를 성공적으로 연결했습니다. 다음과 같이 나머지 필드에 대해 이 작업을 반복합니다.

`addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` 요청 매핑 템플릿

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "addPost",
        "arguments":  $utils.toJson($context.arguments)
    }
}
```

`addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` 응답 매핑 템플릿

```
$utils.toJson($context.result)
```

`allPosts: [Post]` 요청 매핑 템플릿

```
{
    "version": "2017-02-28",
```

```
    "operation": "Invoke",
    "payload": {
        "field": "allPosts"
    }
}
```

allPosts: [Post] 응답 매핑 템플릿

```
$utils.toJson($context.result)
```

Post.relatedPosts: [Post] 요청 매핑 템플릿

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "relatedPosts",
        "source":  $utils.toJson($context.source)
    }
}
```

Post.relatedPosts: [Post] 응답 매핑 템플릿

```
$utils.toJson($context.result)
```

GraphQL API 테스트

Lambda 함수가 GraphQL 해석기에 연결되었으므로 콘솔이나 클라이언트 애플리케이션을 사용하여 뮤테이션 및 쿼리를 실행할 수 있습니다.

AWS AppSync 콘솔에서 왼쪽에 있는 쿼리를 선택한 후, 다음 코드를 붙여 넣습니다.

addPost 변형

```
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 쿼리

```
query getPost {
    getPost(id: "2") {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

allPosts 쿼리

```
query allPosts {
    allPosts {
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {
            id
            title
        }
    }
}
```

오류 반환

필드 해상도를 지정하면 오류가 발생할 수 있습니다. AWS AppSync를 사용하면 다음 소스에서 오류를 발생시킬 수 있습니다.

- 요청 또는 응답 매핑 템플릿
- Lambda 함수

매핑 템플릿에서

의도적인 오류를 발생시키려면 Velocity 템플릿 언어(VTL) 템플릿의 `$utils.error` 도우미 메서드를 사용하면 됩니다. `errorMessage`, `errorType` 및 선택적 `data` 값을 인수로 사용합니다. `data`는 오류가 발생한 경우 외부 데이터를 클라이언트로 반환할 때 유용합니다. GraphQL 최종 응답에서 `errors`에 `data` 객체가 추가됩니다.

다음 예시는 `Post.relatedPosts: [Post]` 응답 매핑 템플릿에서 이 데이터 객체를 사용하는 방법을 보여줍니다.

```
$utils.error("Failed to fetch relatedPosts", "LambdaFailure", $context.result)
```

다음과 유사한 GraphQL 응답이 산출됩니다.

```
{
    "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"
                }
            ]
        }
    ]
}
```

여기서 `allPosts[0].relatedPosts`는 오류로 인해 null이며 `errorMessage`, `errorType` 및 `data`가 `data.errors[0]` 객체 안에 있습니다.

Lambda 함수에서

또한 AWS AppSync는 Lambda 함수에서 발생하는 오류를 이해합니다. Lambda 프로그래밍 모델은 handled 오류를 발생시킵니다. Lambda 함수에서 오류가 발생하는 경우 AWS AppSync에서 현재 필드가 해석되지 않습니다. Lambda에서 반환되는 오류 메시지만 응답에 설정됩니다. 현재는 Lambda 함수에서 오류를 발생시켜 관련 없는 데이터를 클라이언트에 다시 전달할 수 없습니다.

참고: Lambda 함수가 unhandled 오류를 발생시키는 경우 AWS AppSync는 Lambda에서 설정한 오류 메시지를 사용합니다.

다음 Lambda 함수는 오류를 발생시킵니다.

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    callback("I fail. Always.");
};
```

다음과 유사한 GraphQL 응답이 반환됩니다.

```
{
    "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."
        }
    ]
}
```

고급 사용 사례: 일괄 처리

이 예제의 Lambda 함수에는 해당 게시물에 대한 관련 게시물 목록을 반환하는 `relatedPosts` 필드가 있습니다. 예제 쿼리에서 Lambda 함수의 `allPosts` 필드 호출은 5개 게시물을 반환합니다. 각각의 반환된 게시물에 대해 `relatedPosts`를 해석하도록 지정했으므로 `relatedPosts` 필드 작업이 5회 호출됩니다.

```
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
        }
    }
}
```

이 특정 예제에서는 큰 문제가 되지 않을 수 있지만, 이러한 복합적이고 과도한 가져오기로 인해 애플리케이션이 빠르게 손상될 수 있습니다.

동일한 쿼리에서 반환된 관련 Posts에 대해 relatedPosts를 다시 가져오려 할 경우 호출 횟수가 크게 증가할 것입니다.

```
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
            }
        }
    }
}
```

이 매우 단순한 쿼리에서는 AWS AppSync가 Lambda 함수를 1 + 5 + 25 = 31번 호출합니다.

이는 상당히 일반적인 문제로 대개 N+1 문제(이 경우에 N = 5)라고도 하며 이로 인해 애플리케이션의 지연 시간 및 비용이 증가될 수 있습니다.

이 문제를 해결하는 한 가지 방법은 유사한 필드 해석기 요청을 하나로 묶는 것입니다. 이 예제에서는 지정된 단일 게시물에 대한 관련 게시물 목록을 해석하는 Lambda 함수 하나 대신에, 해당 게시물 배치에 대한 관련 게시물 목록을 해석합니다.

예시를 배치위해 `Post.relatedPosts: [Post]` 해석기를 배치 활성화된 해석기로 전환하겠습니다.

AWS AppSync 콘솔의 오른쪽에서 기존 `Post.relatedPosts: [Post]` 해석기를 선택합니다. 요청 매핑 템플릿을 다음으로 변경합니다.

```
{
    "version": "2017-02-28",
    "operation": "BatchInvoke",
    "payload": {
        "field": "relatedPosts",
        "source":  $utils.toJson($context.source)
    }
}
```

`operation` 필드만 `Invoke`에서 `BatchInvoke`로 변경되었습니다. 이제 페이로드 필드가 템플릿에 지정된 배열이 됩니다. 이 예제에서는 Lambda 함수가 다음을 입력으로 수신합니다.

```
[
    {
        "field": "relatedPosts",
        "source": {
            "id": 1
        }
    },
    {
        "field": "relatedPosts",
        "source": {
            "id": 2
        }
    },
    ...
]
```

`BatchInvoke`가 요청 매핑 템플릿에 지정되었으면 Lambda 함수가 요청 목록을 수신하고 결과 목록을 반환합니다.

특히, 결과 목록이 요청 페이로드 항목의 순서와 크기 면에서 일치해야 하므로 AWS AppSync가 그에 맞게 결과를 일치시킬 수 있습니다.

이 일괄 처리 예제에서 Lambda 함수는 다음과 같이 결과의 배치를 반환합니다.

```
[
    [{"id":"2","title":"Second book"}, {"id":"3","title":"Third book"}],   //
 relatedPosts for id=1
    [{"id":"3","title":"Third book"}]
           // relatedPosts for id=2
]
```

Node.js의 다음 Lambda 함수는 다음과 같이 `Post.relatedPosts` 필드에 대한 이 일괄 처리 기능을 보여줍니다.

```
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;
    }
};
```

개별 오류 반환

앞의 예에서는 Lambda 함수에서 단일 오류가 반환되거나 매핑 템플릿에서 오류를 일으킬 수 있음을 보았습니다. 일괄 처리된 호출의 경우, Lambda 함수에서 오류가 발생하면 전체 배치가 실패로 플래그 지정됩니다. 이는 데이터 스토어와의 연결 중단 같은 취소 불가능한 오류가 발생하는 특정 시나리오를 잘 설명할 수 있습니다. 하지만 배치의 일부 항목이 성공하고 다른 항목이 실패하는 경우, 오류 및 유효한 데이터를 모두 반환할 수 있습니다. AWS AppSync에서는 배치의 원래 크기와 일치하는 요소를 나열하기 위해 배치 응답이 필요하므로 유효한 데이터와 오류를 구분할 수 있는 데이터 구조를 사용자가 정의해야 합니다.

예를 들어 Lambda 함수가 관련 게시물의 일괄 처리를 반환해야 하는 경우 각 객체에 선택적 data, errorMessage 및 errorType 필드가 있는 `Response` 객체 목록을 반환하도록 선택할 수 있습니다. errorMessage 필드가 있는 경우 오류가 발생했음을 의미합니다.

다음 코드는 Lambda 함수를 업데이트하는 방법을 보여줍니다.

```
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;
```

```
    }
};
```

이 예의 경우,다음의 응답 매핑 템플릿을 통해 Lambda 함수의 각 항목과 발생하는 모든 오류를 구문 분석할 수 있습니다.

```
#if( $context.result && $context.result.errorMessage )
    $utils.error($context.result.errorMessage, $context.result.errorType,
 $context.result.data)
#else
    $utils.toJson($context.result.data)
#end
```

이 예에서는 다음과 유사한 GraphQL 응답이 반환됩니다.

```
{
  "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"
    }
  ]
}
```

최대 배치 크기 구성

기본적으로 BatchInvoke를 사용하면 AWS AppSync는 Lambda 함수에 요청을 최대 5개 항목의 일괄 처리로 전송합니다. Lambda 해석기의 최대 배치 크기를 구성할 수 있습니다.

해석기에서 최대 일괄 처리 크기를 구성하려면 AWS Command Line Interface(AWS CLI)에서 다음 명령을 사용합니다.

```
$ 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**
>
> 요청 매핑 템플릿을 제공할 때는 BatchInvoke 작업을 사용하여 일괄 처리를 사용해야 합니다.

다음 명령을 사용하여 직접 Lambda 해석기에서 일괄 처리를 활성화하고 구성할 수도 있습니다.

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
 relatedPosts \
 --data-source-name "<lambda-datasource>" \
 --max-batch-size X
```

VTL 템플릿을 사용한 최대 배치 크기 구성

VTL 요청 내 템플릿이 있는 Lambda 해석기의 경우, 최대 배치 크기는 VTL에서 `BatchInvoke` 작업으로 직접 지정하지 않는 한 효과가 없습니다. 마찬가지로 최상위 뮤테이션을 수행하는 경우 GraphQL 사양에서 병렬 뮤테이션을 순차적으로 실행하도록 요구하기 때문에 뮤테이션에 대한 일괄 처리가 수행되지 않습니다.

예를 들어, 다음과 같은 뮤테이션이 있다고 가정합니다.

```
type Mutation {
    putItem(input: Item): Item
    putItems(inputs: [Item]): [Item]
}
```

첫 번째 뮤테이션을 사용하여 아래 코드 조각과 같이 10개 `Items`를 생성할 수 있습니다.

```
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
    }
}
```

이 예제에서는 Lambda 해석기에서 최대 일괄 처리 크기가 10으로 설정되어 있어도 `Items`는 10개의 그룹으로 일괄 처리되지 않습니다. 대신 GraphQL 사양에 따라 순차적으로 실행됩니다.

실제 배치 뮤테이션을 수행하려면 다음 예제를 따라 두 번째 뮤테이션을 사용할 수 있습니다.

```
mutation MyMutation {
    putItems([$someItem1, $someItem2, $someItem3,$someItem4, $someItem5, $someItem6,
    $someItem7, $someItem8, $someItem9, $someItem10]) {
    id,
    name
    }
}
```

직접 Lambda 해석기를 사용한 일괄 처리 사용에 대한 자세한 내용은 Direct Lambda 해석기 단원을 참조하세요.

자습서: Amazon OpenSearch Service 해석기

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync에서는 고유의 AWS 계정에서 프로비저닝한 도메인이 VPC 내에 존재하지 않는 경우, 이 도메인의 Amazon OpenSearch Service 사용을 지원합니다. 도메인을 프로비저닝한 후에는 도메인을 데이터 원본에 연결할 수 있으며, 이때 쿼리, 변형 및 구독 등 GraphQL 작업을 수행하도록 스키마의 해석기를 구성할 수 있습니다. 이 자습서에서는 몇 가지 일반적인 예제를 살펴봅니다.

자세한 내용은 OpenSearch용 해석기 매핑 템플릿 참조 단원을 참조하세요.

원클릭 설치

Amazon OpenSearch Service가 구성된 상태에서 AWS AppSync에서 GraphQL 엔드포인트를 자동으로 설정하려면 이 AWS CloudFormation 템플릿을 사용할 수 있습니다:

AWS CloudFormation 배포가 완료된 후에는 곧바로 GraphQL 쿼리 및 변형 실행으로 이동할 수 있습니다.

새 OpenSearch Service 도메인 생성

이 자습서를 시작하기 위해서는 기존의 OpenSearch Service 도메인이 필요합니다. 아직 없는 경우 다음 샘플을 사용할 수 있습니다. AWS AppSync 데이터 소스로 이동하려면 먼저 최대 15분에 걸쳐 OpenSearch Service 도메인을 생성해야 합니다.

```
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
```

AWS 계정의 미국 서부 2(오레곤) 리전에서 다음 AWS CloudFormation 스택을 시작할 수도 있습니다.

OpenSearch Service를 위한 데이터 소스 구성

OpenSearch Service 도메인이 생성된 후에는 AWS AppSync GraphQL API로 이동하여 데이터 소스 탭을 선택합니다. 새로 만들기를 선택하고 'oss'와 같은 데이터 소스에 사용할 친숙한 이름을 입력합니

다. 그런 다음 데이터 소스 유형에 대해 Amazon OpenSearch 도메인을 선택하고 적절한 리전을 선택하면 OpenSearch Service 도메인이 나열됩니다. 도메인을 선택한 후에는 새 역할을 생성할 수 있으며, 그러면 AWS AppSync에서 역할에 맞는 권한을 할당하거나 다음 인라인 정책이 지정된 기존 역할을 선택할 수 있습니다.

```
{
    "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/*"
            ]
        }
    ]
}
```

또한 해당 역할에 대해 AWS AppSync와의 신뢰 관계를 설정해야 합니다.

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

또한 OpenSearch Service 도메인에는 Amazon OpenSearch Service 콘솔을 통해 수정할 수 있는 자체 액세스 정책이 있습니다. 사용자는 OpenSearch Service 도메인에 대한 적정 작업 및 리소스를 사용하

여 다음과 비슷한 정책을 추가해야 합니다. 보안 주체는 AppSync 데이터 소스 역할로, 콘솔에서 이 역할을 생성하도록 할 경우 IAM 콘솔에서 확인할 수 있습니다.

```
{
    "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/*"
        }
    ]
}
```

해석기 연결

이제 데이터 소스가 OpenSearch Service 도메인에 연결되었으며, 다음 예제에서처럼 해석기를 사용하여 GraphQL 스키마에 연결할 수 있습니다.

```
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
}
...
```

사용자 정의 Post 유형과 id 필드가 있습니다. 다음 예제에서는 이 유형을 OpenSearch Service 도메인에 넣는 프로세스(자동화할 수 있음)가 있다고 가정하고, 이 프로세스는 /post/_doc의 경로 루트에 매핑되며, 여기서 post는 인덱스입니다. 이 루트 경로에서 개별 문서 검색, /id/post*를 사용한 와일드카드 검색 또는 /post/_search 경로를 사용한 다중 문서 검색을 수행할 수 있습니다. 예를 들어, User라는 다른 유형이 있는 경우 user라는 새 인덱스로 문서를 색인한 다음 /user/_search의 경로로 검색을 수행할 수 있습니다.

AWS AppSync 콘솔의 스키마 편집기에서 searchPosts 쿼리를 포함하도록 이전 Posts 스키마를 수정합니다.

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

스키마를 저장합니다. 오른쪽에 있는 searchPosts에서 Attach resolver(해석기 연결)를 선택합니다. 작업 메뉴에서 런타임 업데이트를 선택한 다음 단위 해석기(VTL만 해당)를 선택합니다. 그런 다음 OpenSearch Service 데이터 소스를 선택합니다. request mapping template(요청 매핑 템플릿) 섹션에서 Query posts(게시물 쿼리)의 드롭다운을 선택하여 기본 템플릿을 가져옵니다. path를 /post/_search로 수정합니다. 이 템플릿은 다음과 같습니다.

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/post/_search",
    "params":{
        "headers":{},
```

```
        "queryString":{},
        "body":{
            "from":0,
            "size":50
        }
    }
}
```

여기서는 앞의 스키마에 post 필드 아래에 OpenSearch Service에서 색인된 문서가 있다고 가정합니다. 데이터를 다르게 구조화하는 경우 그에 맞게 업데이트해야 합니다.

응답 매핑 템플릿 섹션에서 OpenSearch Service 쿼리로부터 데이터 결과를 다시 가져와서 GraphQL로 변환하려면 적절한 _source 필터를 지정해야 합니다. 다음 템플릿을 사용합니다.

```
[
    #foreach($entry in $context.result.hits.hits)
    #if( $velocityCount > 1 ) , #end
    $utils.toJson($entry.get("_source"))
    #end
]
```

검색 수정

이전 요청 매핑 템플릿은 모든 레코드에 대해 단순 쿼리를 수행합니다. 특정 작성자별로 검색하려 한다고 가정하겠습니다. 또한, 작성자를 GraphQL 쿼리에 정의된 인수로 사용하려 한다고 가정하겠습니다. AWS AppSync 콘솔의 스키마 편집기에서 allPostsByAuthor 쿼리를 추가합니다.

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

이제 해석기 연결을 선택하고 OpenSearch Service 데이터 소스를 선택하되, 응답 매핑 템플릿에서 다음 예제를 사용합니다.

```
{
    "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)
                }
            }
        }
    }
}
```

body가 author 필드에 대한 쿼리 용어로 채워져서 클라이언트에서 인수로 전달됩니다. 또한 원할 경우 표준 텍스트 같은 미리 채워진 정보를 사용하거나 다른 유틸리티를 사용할 수도 있습니다.

이 해석기를 사용하려는 경우 이전 예와 동일한 정보를 Response Mapping Template(응답 매핑 템플릿)에 입력합니다.

OpenSearch Service에 데이터 추가

GraphQL 뮤테이션의 결과로서 OpenSearch Service 도메인에 데이터를 추가해야 할 수 있습니다. 이는 검색 및 다른 목적으로 사용할 때 유용한 메커니즘입니다. GraphQL 구독을 사용하여 데이터를 실시간으로 만들 수 있으므로, 이는 클라이언트에 OpenSearch Service 도메인의 데이터에 대한 업데이트를 알리는 메커니즘으로 작용합니다.

AWS AppSync 콘솔의 스키마 페이지로 돌아가서 addPost() 뮤테이션에 대한 해석기 연결을 선택합니다. OpenSearch Service 데이터 소스를 다시 선택하고 다음 Posts 스키마에 대한 응답 매핑 템플릿을 사용합니다.

```
{
    "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)
        }
    }
}
```

전과 마찬가지로, 이 코드도 데이터의 구조화 방식을 보여주는 예입니다. 다양한 필드 이름 또는 인덱스가 있으면 해당되는 경우 path 및 body를 업데이트해야 합니다. 이 예제는 $context.arguments를 사용하여 GraphQL 변형 인수에서 템플릿을 채우는 방법도 보여줍니다.

계속 진행하기 전에 다음 응답 매핑 템플릿을 사용하여 뮤테이션 작업 결과 또는 오류 정보를 출력으로 반환합니다.

```
#if($context.error)
    $util.toJson($ctx.error)
#else
    $util.toJson($context.result)
#end
```

단일 문서 가져오기

마지막으로, 스키마에서 getPost(id:ID) 쿼리를 사용하여 개별 문서를 반환하려는 경우 AWS AppSync 콘솔의 스키마 편집기에서 이 쿼리를 찾아 해석기 연결을 선택합니다. OpenSearch Service 데이터 소스를 다시 선택하고 다음 매핑 템플릿을 사용합니다.

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path": $util.toJson("post/_doc/$context.arguments.id"),
    "params":{
        "headers":{},
        "queryString":{},
        "body":{}
    }
}
```

위의 path에서 빈 본문과 함께 id 인수를 사용하므로 이 문은 단일 문서를 반환합니다. 하지만 지금은 목록이 아니라 단일 항목을 반환하려고 하므로 다음과 같은 응답 매핑 템플릿을 사용해야 합니다.

```
$utils.toJson($context.result.get("_source"))
```

쿼리 및 변형 수행

이제 OpenSearch Service 도메인에 대해 GraphQL 작업을 수행할 수 있습니다. AWS AppSync 콘솔의 쿼리 탭으로 이동하여 새 레코드를 추가합니다.

```
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
       )
}
```

오른쪽에 뮤테이션 결과가 표시됩니다. 마찬가지로, 이제 OpenSearch Service 도메인에 대해 `searchPosts` 쿼리를 실행할 수 있습니다.

```
query searchPosts {
    searchPosts {
        id
        title
        author
        content
    }
}
```

모범 사례

- OpenSearch Service는 기본 데이터베이스가 아니라 데이터 쿼리용입니다. GraphQL 해석기 결합에서 설명했듯이 OpenSearch Service를 Amazon DynamoDB와 함께 사용할 수도 있습니다.
- AWS AppSync 서비스 역할에서 클러스터에 액세스하도록 허용하여 도메인에 대한 액세스 권한을 부여하기만 하면 됩니다.
- 최저 비용 클러스터를 제공하는 간단한 개발부터 시작하여, 프로덕션으로 들어가면서 고가용성(HA)을 제공하는 대규모 클러스터로 이동할 수 있습니다.

자습서: 로컬 해석기

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync를 사용하면 지원되는 데이터 소스(AWS Lambda, Amazon DynamoDB 또는 Amazon OpenSearch Service)를 사용하여 다양한 작업을 수행할 수 있습니다. 하지만 특정 시나리오에서는 지원되는 데이터 원본을 호출할 필요가 없을 수도 있습니다.

이러한 경우에는 로컬 해석기가 유용합니다. 로컬 해석기는 원격 데이터 원본을 호출하는 것이 아니라 단순히 요청 매핑 템플릿의 결과를 응답 매핑 템플릿으로 전달할 뿐입니다. 필드 해석은 AWS AppSync에서 이루어집니다.

로컬 해석기는 여러 사용 사례에서 유용합니다. 가장 널리 알려진 사용 사례는 데이터 원본 호출을 트리거하지 않고 알림을 게시하는 경우입니다. 이 사용 사례를 설명하기 위해 사용자가 서로 간에 페이징할 수 있는 페이징 애플리케이션을 빌드해 보겠습니다. 이 예제에서는 Subscriptions를 활용하므로, Subscriptions에 대해 잘 모르는 경우 실시간 데이터 자습서에 따라 실행할 수도 있습니다.

페이징 애플리케이션 생성

페이징 애플리케이션에서 클라이언트는 받은 편지함을 구독하고 페이지를 다른 클라이언트로 보낼 수 있습니다. 각 페이지에는 메시지가 포함됩니다. 스키마는 다음과 같습니다.

```
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
}
```

Mutation.page 필드에 해석기를 연결해 보십시오. Schema(스키마) 창의 오른쪽 패널에서 필드 정의 옆에 있는 Attach Resolver(해석기 연결)를 클릭합니다. None(없음) 유형의 새 데이터 원본을 생성하고 이름을 PageDataSource로 지정합니다.

요청 매핑 템플릿 창에서 다음을 입력합니다.

```
{
  "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()"
  }
}
```

응답 매핑 템플릿에서는 기본값인 Forward the result(결과 전달)를 선택합니다. 해석기를 저장합니다. 이제 애플리케이션이 준비되었으며 페이징하십시오!

페이지 전송 및 구독

클라이언트에서 페이지를 수신하도록 하려면 먼저 클라이언트가 받은 편지함을 구독해야 합니다.

Queries(쿼리) 창에서 inbox 구독을 실행하십시오.

```
subscription Inbox {
    inbox(to: "Nadia") {
        body
        to
        from
```

```
        sentAt
    }
}
```

Nadia는 `Mutation.page` 뮤테이션이 호출될 때마다 페이지를 수신합니다. 변형을 실행하여 변형을 호출해 보겠습니다.

```
mutation Page {
    page(to: "Nadia", body: "Hello, World!") {
        body
        to
        from
        sentAt
    }
}
```

AWS AppSync를 나가지 않고 간단히 페이지를 전송하고 수신해 봄으로써 로컬 해석기 사용을 보여드렸습니다.

튜토리얼: GraphQL 해석기 결합

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

GraphQL 스키마의 해석기와 필드는 매우 뛰어난 유연성으로 1:1 관계를 갖습니다. 데이터 원본이 스키마와 독립적으로 해석기에 구성되어 있으므로, 스키마를 요구에 가장 잘 맞게 짜맞춰서 다양한 데이터 원본을 통해 GraphQL 유형을 해석하거나 조작할 수 있습니다.

다음 예제 시나리오는 스키마에서 데이터 소스를 혼합하고 일치시키는 방법을 보여줍니다. 시작하기 전에 이전 튜토리얼에서 설명한 대로 AWS Lambda, Amazon DynamoDB 및 Amazon OpenSearch Service에 대한 데이터 소스 및 해석기 설정에 익숙해지는 것이 좋습니다.

스키마 예제

다음 스키마에는 `Post` 작업 3개와 `Query` 작업 3개가 정의된 `Mutation` 유형이 있습니다.

```
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
}
```

이 예제에서는 총 6개 해석기를 연결하게 됩니다. 한 가지 가능한 방법은 DynamoDB 해석기 매핑 템플릿 참조에 설명된 대로 이 모든 것을 `Posts`라고 하는 Amazon DynamoDB 테이블에서 가져와서 `AllPosts`가 스캔을 실행하고 `searchPosts`가 쿼리를 실행하도록 하는 것입니다. 하지만 Lambda 또는 OpenSearch Service를 통해 이러한 GraphQL 쿼리를 해석하는 등 사용자의 비즈니스 요구에 맞는 다른 방법도 있습니다.

해석기를 통해 데이터 변경

DynamoDB(또는 Amazon Aurora) 같은 데이터베이스의 결과를 클라이언트로 반환할 때 몇 가지 속성을 변경한 상태로 반환해야 할 수도 있습니다. 클라이언트의 타임스탬프 차이 같은 데이터 형식의 포맷팅이나 이전 버전과의 호환성 문제를 처리하기 위한 것이 이유일 수 있습니다. 설명을 돕기 위해, 다음 예제에서 AWS Lambda 함수는 GraphQL 해석기가 호출될 때마다 난수를 할당하여 블로그 게시물에 대한 좋아요/싫어요 평가를 다룹니다.

```
'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);
    });
};
```

이 함수는 완벽하게 유효한 Lambda 함수로 GraphQL 스키마의 `AllPosts` 필드에 연결하여 모든 결과에서 반환하는 쿼리가 좋아요/싫어요에 대한 난수를 가져오도록 합니다.

DynamoDB 및 OpenSearch Service

일부 애플리케이션의 경우 DynamoDB에 대한 단순 조회 쿼리와 변형을 수행할 수 있으며, 백그라운드 프로세스를 통해 문서를 OpenSearch Service로 전송할 수 있습니다. 그런 다음 단순히 `searchPosts` 해석기를 OpenSearch Service 데이터 소스에 연결하고 GraphQL 쿼리를 사용하여 (DynamoDB에서 가져온 데이터의) 검색 결과를 반환할 수 있습니다. 이 기능은 키워드, 퍼지 워드 일치 또는 지역 검색 조회 등 애플리케이션에 고급 검색 작업 추가 시 매우 유용할 수 있습니다. DynamoDB에서 데이터 전송은 ETL 프로세스를 통해 수행하거나 Lambda을 사용하여 DynamoDB에서 스트리밍할 수 있습니다.

AWS 계정에서 미국 서부 2(오레곤) 리전의 다음 AWS CloudFormation 스택을 사용하여 이 작업의 전체 예제를 시작할 수 있습니다.

이 예제의 스키마를 통해 다음과 같이 DynamoDB 해석기를 사용하여 게시물을 추가할 수 있습니다.

```
mutation add {
    putPost(author:"Nadia"
        title:"My first post"
        content:"This is some test content"
        url:"https://aws.amazon.com/appsync/"
    ){
        id
        title
    }
}
```

이렇게 하면 DynamoDB에 데이터가 기록되고, Lambda를 통해 Amazon OpenSearch Service로 데이터를 스트리밍합니다. 그러면 다양한 필드로 모든 게시물을 검색할 수 있습니다. 예를 들면 데이터가 Amazon OpenSearch Service에 있으므로 다음과 같이 공백으로만 이루어졌더라도 자유 형식의 텍스트를 사용하여 작성자나 콘텐츠 필드를 검색할 수 있습니다.

```
query searchName{
    searchAuthor(name:"   Nadia   "){
        id
        title
        content
    }
}

query searchContent{
    searchContent(text:"test"){
        id
        title
        content
    }
}
```

데이터가 DynamoDB에 직접 기록되므로 `allPosts{...}` 및 `singlePost{...}` 쿼리를 사용하여 테이블에 대해 효율적인 목록이나 항목 조회 작업을 수행할 수 있습니다. 이 스택은 DynamoDB 스트림에 대해 다음 예제 코드를 사용합니다.

참고:: 이 코드는 예제일 뿐입니다.

```
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);
};
```

DynamoDB 스트림을 사용하여 `id`를 기본 키로 하는 DynamoDB 테이블에 이 부분을 연결하면, 원본에 대한 모든 변경 내용이 OpenSearch Service 도메인으로 스트리밍됩니다. 이 구성에 대한 자세한 내용은 DynamoDB Streams 설명서를 참조하십시오.

자습서: DynamoDB 배치 해석기

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync에서는 단일 리전에 있는 하나 이상의 테이블에서 AppSync DynamoDB 배치 작업 사용을 지원합니다. 지원되는 작업은 `BatchGetItem`, `BatchPutItem` 및 `BatchDeleteItem`입니다. AWS AppSync의 이러한 기능을 사용하면 다음과 같은 작업을 수행할 수 있습니다.

- 단일 쿼리에서 키 목록을 전달하고 테이블의 결과 반환
- 단일 쿼리의 하나 이상의 테이블에서 레코드 읽기
- 하나 이상의 테이블에 대량으로 레코드 쓰기

- 관계가 있을 수 있는 여러 테이블에서 조건부로 레코드 쓰기 또는 삭제

AWS AppSync에서 DynamoDB와 함께 배치 작업을 사용하는 것은 백엔드 작업 및 테이블 구조에 대한 사고와 지식이 조금 더 필요한 고급 기술입니다. 또한 AWS AppSync의 배치 작업에는 배치가 아닌 작업과 크게 차이나는 점이 2개 있습니다.

- 데이터 원본 역할에 해석기가 액세스할 모든 테이블에 대한 권한이 있어야 합니다.
- 해석기에 대한 테이블 사양이 매핑 템플릿의 일부입니다.

권한

다른 해석기와 마찬가지로 AWS AppSync에서 데이터 소스를 생성하고 역할을 생성하거나 기존 역할을 사용해야 합니다. 배치 작업에는 DynamoDB 테이블에 대한 여러 권한이 필요하기 때문에 읽기 또는 쓰기 작업에 대해 구성된 역할 권한을 부여해야 합니다.

```
{
    "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/*"
            ]
        }
    ]
}
```

참고: 역할은 AWS AppSync의 데이터 소스에 연결되어 있고 필드의 해석기는 데이터 소스에 대해 호출됩니다. DynamoDB에 대해 가져오도록 구성된 데이터 소스에는 구성을 간단하기 유지하기 위해 지정된 테이블이 하나뿐입니다. 따라서 단일 해석기에서 여러 테이블에 대해 배치 작업을 수행하는 경우(고급 작업임) 데이터 원본 액세스에 대한 역할을 해석기가 상호 작용하는 모든 테이블에 부여해야 합니다. IAM 정책의 리소스 필드에서 부여할 수 있습니다. 배치 호출을 수행하려는 테이블의 구성은 해석기 템플릿에서 수행되는데, 이 내용은 아래에서 설명합니다.

데이터 원본

간단하게 설명하기 위해 이 자습서에서 사용되는 모든 해석기에 대해 동일한 데이터 원본을 사용합니다. 데이터 소스 탭에서 새 DynamoDB 데이터 소스를 생성하고 BatchTutorial이라고 이름을 지정합니다. 테이블 이름은 배치 작업을 위한 요청 매핑 템플릿의 일부로 지정되므로 어떤 것이든 지정할 수 있습니다. 테이블의 이름을 empty라고 지정하겠습니다.

이 자습서의 경우 다음 인라인 정책이 적용된 역할이 모두 작동합니다.

```
{
    "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/*"
            ]
        }
    ]
}
```

단일 테이블 배치

이 예에서는 배치 작업을 통해 항목을 추가 및 제거하려는 Posts라는 단일 테이블이 있다고 가정합니다. 다음 스키마를 사용하는데, 쿼리를 위해 ID 목록을 전달합니다.

```
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
}
```

다음 요청 매핑 템플릿을 사용하여 batchAdd( ) 필드에 해석기를 첨부합니다. 이렇게 하면 GraphQL input PostInput 형식의 각 항목을 자동으로 받아 BatchPutItem 작업에 필요한 맵을 빌드합니다.

```
#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)
    }
}
```

이 경우 응답 매핑 템플릿은 간단한 패스스루인데, 테이블 이름이 다음과 같이 컨텍스트 객체에 ..data.Posts로 추가됩니다.

```
$util.toJson($ctx.result.data.Posts)
```

이제 AWS AppSync 콘솔의 쿼리 페이지로 이동하여 다음 batchAdd 뮤테이션을 실행합니다.

```
mutation add {
```

```
    batchAdd(posts:[{
            id: 1 title: "Running in the Park"},{
            id: 2 title: "Playing fetch"
        }]){
            id
            title
    }
}
```

결과가 화면에 출력되어야 하며 DynamoDB 콘솔을 통해 두 값이 Posts 테이블에 기록되었는지 독립적으로 확인할 수 있습니다.

다음으로, 다음 요청 매핑 템플릿을 사용하여 `batchGet()` 필드에 해석기를 첨부합니다. 이렇게 하면 GraphQL `ids:[]` 형식의 각 항목을 자동으로 받아 `BatchGetItem` 작업에 필요한 맵을 빌드합니다.

```
#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"}
                }
        }
    }
}
```

이 경우 응답 매핑 템플릿은 다시 간단한 패스스루인데, 테이블 이름이 다시 컨텍스트 객체에 `..data.Posts`로 추가됩니다.

```
$util.toJson($ctx.result.data.Posts)
```

이제 AWS AppSync 콘솔의 쿼리 페이지로 돌아가 다음 batchGet 쿼리를 실행합니다.

```
query get {
    batchGet(ids:[1,2,3]){
        id
        title
    }
}
```

앞서 추가한 두 id 값에 대한 결과를 반환해야 합니다. null 값은 값이 id인 3에 대해 반환되었습니다. 이는 Posts 테이블에 해당 값을 가진 레코드가 아직 없기 때문입니다. 또한, AWS AppSync에서는 쿼리에 키가 전달된 순서대로 결과를 반환하는데, 이는 AWS AppSync에서 사용자를 대신하여 수행하는 추가 기능입니다. batchGet(ids:[1,3,2)로 전환하면 순서가 바뀝니다. 또한 어떤 id가 null 값을 반환하는지 알게 됩니다.

마지막으로, 다음 요청 매핑 템플릿을 사용하여 batchDelete() 필드에 해석기를 첨부합니다. 이렇게 하면 GraphQL ids:[] 형식의 각 항목을 자동으로 받아 BatchGetItem 작업에 필요한 맵을 빌드합니다.

```
#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)
    }
}
```

이 경우 응답 매핑 템플릿은 다시 간단한 패스스루인데, 테이블 이름이 다시 컨텍스트 객체에 ..data.Posts로 추가됩니다.

```
$util.toJson($ctx.result.data.Posts)
```

이제 AWS AppSync 콘솔의 쿼리 페이지로 돌아가 다음 batchDelete 뮤테이션을 실행합니다.

```
mutation delete {
    batchDelete(ids:[1,2]){ id }
}
```

이제 id 1 및 2인 레코드가 삭제되어야 합니다. 앞에서 batchGet() 쿼리를 다시 실행하면 null을 반환해야 합니다.

다중 테이블 배치

또한 AWS AppSync에서는 여러 테이블 간에 배치 작업을 수행할 수 있습니다. 더 복잡한 애플리케이션을 빌드해 보겠습니다. 센서가 애완동물의 위치와 신체 온도를 보고하는 애완동물 건강 앱을 빌드한다고 가정해 보겠습니다. 센서는 배터리로 구동되며 몇 분마다 네트워크에 연결을 시도합니다. 센서가 연결을 설정하면 판독값을 AWS AppSync API로 전송합니다. 그런 다음 트리거가 데이터를 분석해 애완동물 주인에게 대시보드가 제공됩니다. 센서와 백엔드 데이터 스토어 간의 상호 작용에 대해 중점적으로 설명해 보겠습니다.

전제 조건으로 먼저 두 개의 DynamoDB 테이블을 만들어 보겠습니다. locationReadings는 센서 위치 판독값을 저장하고 temperatureReadings는 센서 온도 판독값을 저장할 것입니다. 두 테이블은 동일한 기본 키 구조를 공유하는데, sensorId (String)는 파티션 키이고, timestamp (String)는 정렬 키입니다.

다음 GraphQL 스키마를 사용해 보겠습니다.

```
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 - 센서 판독값 기록

센서는 인터넷에 연결되면 판독값을 전송할 수 있어야 합니다. GraphQL 필드 `Mutation.recordReadings`가 센서가 판독값을 전송하는 데 사용할 API입니다. API를 실행하기 위해 해석기를 연결해 보겠습니다.

`Mutation.recordReadings` 필드 옆의 첨부를 선택합니다. 다음 화면에서 자습서 시작 부분에서 생성한 것과 같은 `BatchTutorial` 데이터 원본을 선택합니다.

다음 요청 매핑 템플릿을 추가해 보겠습니다.

요청 매핑 템플릿

```
## 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)
    }
}
```

보시다시피 `BatchPutItem` 작업을 통해 여러 테이블을 지정할 수 있습니다.

다음 응답 매핑 템플릿을 사용해 보겠습니다.

응답 매핑 템플릿

```
## 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)
```

배치 작업을 사용하면 호출 시 오류 및 결과가 둘 다 반환될 수 있습니다. 이러한 경우 몇 가지 오류 처리를 작업을 자유롭게 수행할 수 있습니다.

참고: $utils.appendError() 사용은 $util.error()와 유사하지만 매핑 템플릿 평가를 중단하지 않는다는 큰 차이점이 있습니다. 대신 필드에 오류가 있다는 신호를 보내지만 템플릿이 계속해서 평가되도록 하고 이어서 호출자에게 데이터를 반환합니다. 애플리케이션이 일부 결과를 반환해야 하는 경우에는 $utils.appendError()를 사용하는 것이 좋습니다.

해석기를 저장하고 AppSync AWS 콘솔의 쿼리 페이지로 이동합니다. 센서 판독값을 몇 개 전송해 보겠습니다!

다음 변형을 실행합니다.

```
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
    }
  }
```

```
}
```

하나의 변형으로 센서 판독값 10개를 전송했고, 판독값은 테이블 2개로 분할되어 있었습니다. DynamoDB 콘솔을 사용하여 데이터가 locationReadings 및 temperatureReadings 테이블 모두에 표시되는지 확인합니다.

BatchDeleteItem - 센서 판독값 삭제

마찬가지로 센서 판독값의 배치를 삭제해야 합니다. 이를 위해 `Mutation.deleteReadings` GraphQL 필드를 사용해 보겠습니다. `Mutation.recordReadings` 필드 옆의 첨부를 선택합니다. 다음 화면에서 자습서 시작 부분에서 생성한 것과 같은 `BatchTutorial` 데이터 원본을 선택합니다.

다음 요청 매핑 템플릿을 사용해 보겠습니다.

요청 매핑 템플릿

```
## 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)
    }
}
```

이 응답 매핑 템플릿은 Mutation.recordReadings에 사용한 것과 동일합니다.

응답 매핑 템플릿

```
## 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)
```

해석기를 저장하고 AppSync AWS 콘솔의 쿼리 페이지로 이동합니다. 이제 센서 판독값 두 개를 삭제해 보겠습니다!

다음 변형을 실행합니다.

```
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
    }
  }
}
```

DynamoDB 콘솔을 사용하여 판독값 2개가 locationReadings 및 temperatureReadings 테이블에서 삭제되었는지 확인합니다.

BatchGetItem - 판독값 검색

애완동물 건강 앱의 또 다른 일반 작업은 특정 시점에 센서의 판독값을 검색하는 것입니다. 스키마의 `Query.getReadings` GraphQL 필드에 해석기를 연결해 보겠습니다. 연결을 선택하고 다음 화면에서 자습서 시작 부분에서 생성한 것과 같은 `BatchTutorial` 데이터 원본을 선택합니다.

다음 요청 매핑 템플릿을 추가해 보겠습니다.

요청 매핑 템플릿

```
## 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
        }
    }
}
```

이제 BatchGetItem 작업을 사용합니다.

`SensorReading` 목록을 반환하도록 선택했으므로 응답 매핑 템플릿이 약간 다릅니다. 호출 결과를 원하는 모양으로 매핑해 보겠습니다.

응답 매핑 템플릿

```
## 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)
```

해석기를 저장하고 AppSync AWS 콘솔의 쿼리 페이지로 이동합니다. 이제, 센서 판독값을 검색해 보겠습니다!

다음 쿼리를 실행합니다.

```
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
    }
  }
}
```

AWS AppSync을 사용하여 DynamoDB 배치 작업을 사용하는 방법을 보여드렸습니다.

오류 처리

AWS AppSync에서 데이터 소스 작업은 때때로 일부 결과를 반환할 수 있습니다. 일부 결과는 작업의 출력이 일부 데이터와 오류로 구성된 경우를 나타내는 데 사용되는 용어입니다. 오류는 본질적으로 애플리케이션별로 처리되므로 AWS AppSync에서는 응답 매핑 템플릿에서 오류를 처리할 수 있는 기회를 제공합니다. 해석기 호출 오류(있는 경우)는 컨텍스트에서 `$ctx.error`로 확인할 수 있습니다. 호출 오류에는 항상 `$ctx.error.message` 및 `$ctx.error.type` 속성으로 액세스할 수 있는 메시지

와 유형이 포함되어 있습니다. 응답 매핑 템플릿 호출 중 다음과 같은 3가지 방법으로 일부 결과를 처리할 수 있습니다.

1. 데이터를 반환하여 호출 오류를 가립니다.
2. 응답 매핑 템플릿 평가를 중단해 오류를 나타냅니다(`$util.error(...)` 사용). 이 경우에는 데이터가 반환되지 않습니다.
3. 오류를 추가하고(`$util.appendError(...)` 사용) 데이터도 반환합니다.

DynamoDB 배치 작업을 사용해 위의 3가지 방법을 각각 시연해 보겠습니다!

DynamoDB 배치 작업

DynamoDB 배치 작업을 사용하면 배치를 부분적으로 완료할 수 있습니다. 즉, 요청된 항목 또는 키 중 일부가 처리되지 않은 상태로 남아 있을 수 있습니다. AWS AppSync가 배치를 완료할 수 없는 경우 처리되지 않은 항목 및 호출 오류가 컨텍스트에 따라 설정됩니다.

이 자습서 이전 단원에서 사용한 `Query.getReadings` 작업의 `BatchGetItem` 필드 구성을 사용하여 오류 처리를 구현하겠습니다. 이때, `Query.getReadings` 필드를 실행하는 동안 `temperatureReadings` DynamoDB 테이블에 프로비저닝된 처리량이 부족하다고 가정해 보겠습니다. DynamoDB는 배치의 나머지 요소를 처리하기 위해 AWS AppSync의 두 번째 시도 중에 ProvisionedThroughputExceededException을 발생시켰습니다.

다음 JSON은 DynamoDB 배치 호출 이후, 응답 매핑 템플릿이 평가되기 전 직렬화된 컨텍스트를 나타냅니다.

```
{
  "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": []
}
```

컨텍스트에 대해서 알아야 할 몇 가지가 있습니다.

- 호출 오류는 AWS AppSync에 의해 컨텍스트에서 `$ctx.error`로 설정되었으며 오류 유형은 DynamoDB:ProvisionedThroughputExceededException으로 설정되었습니다.
- 오류가 있더라도 결과는 `$ctx.result.data`에서 테이블당 매핑됩니다.
- 처리되지 않고 남아 있는 키는 `$ctx.result.data.unprocessedKeys`에서 확인할 수 있습니다. 여기서 AWS AppSync는 키(sensorId:1, timestamp:2018-02-01T17:21:05.000+08:00)를 사용해 항목을 검색할 수 없습니다. 테이블 처리량이 부족하기 때문입니다.

참고: `BatchPutItem`의 경우 `$ctx.result.data.unprocessedItems`입니다. `BatchDeleteItem`의 경우 `$ctx.result.data.unprocessedKeys`입니다.

이 오류를 세 가지 다른 방법으로 처리해 보겠습니다.

1. 호출 오류 가리기

호출 오류를 처리하지 않고 데이터를 반환하면 오류를 효과적으로 가려 주어진 GraphQL 필드에 대한 결과가 항상 성공입니다.

여기서 작성한 응답 매핑 템플릿은 알기 쉽고 결과 데이터에만 중점을 두고 있습니다.

응답 매핑 템플릿:

```
$util.toJson($ctx.result.data)
```

GraphQL 응답:

```
{
  "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
      }
    ]
  }
}
```

데이터에 대해서만 실행되었으므로 오류 응답에 오류가 추가되지 않습니다.

2. 템플릿 실행 중단을 위한 오류 발생

클라이언트의 관점에서 부분 실패를 전체 실패로 간주해야 하는 경우 데이터가 반환되지 않도록 템플릿 실행을 중단할 수 있습니다. `$util.error(...)` 유틸리티 메서드가 정확하게 이러한 동작을 수행합니다.

응답 매핑 템플릿:

```
## 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)
```

GraphQL 응답:

```
{
  "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. (...)"
    }
  ]
}
```

일부 결과가 DynamoDB 배치 작업에서 반환될 수 있긴 하지만 오류를 일으키기로 선택합니다. 그러면 `getReadings` GraphQL 필드는 null이고 GraphQL 응답 오류 블록에 오류가 추가됩니다.

3. 오류를 추가하고 데이터 및 오류를 둘 다 반환

특정한 경우 사용자 경험을 개선하기 위해 애플리케이션에서는 일부 결과를 반환하고 클라이언트에게 처리되지 않은 항목이 있음을 알릴 수 있습니다. 그러면 클라이언트는 재시도를 수행하거나 오류를 최

종 사용자에게 다시 번역할지 결정할 수 있습니다. `$util.appendError(...)`는 애플리케이션 디자이너가 템플릿 평가를 방해하지 않고 컨텍스트에 오류를 추가하도록 하여 이러한 동작을 가능하게 하는 유틸리티 메서드입니다. 템플릿 평가 후 AWS AppSync에서는 GraphQL 응답의 오류 블록에 오류를 추가해 모든 컨텍스트 오류를 처리합니다.

응답 매핑 템플릿:

```
#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)
```

GraphQL 응답의 오류 블록 내에서 호출 오류 및 처리되지 않은 키 요소를 전달합니다. 아래 응답에 표시된 것처럼 `getReadings` 필드 역시 locationReadings 테이블의 일부 데이터를 반환합니다.

GraphQL 응답:

```
{
  "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. (...)"
    }
  ]
}
```

자습서: DynamoDB 트랜잭션 해석기

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync에서는 단일 리전에 있는 하나 이상의 테이블에서 AppSync DynamoDB 트랜잭션 작업 사용을 지원합니다. 지원되는 작업은 `TransactGetItems`, `TransactWriteItems`입니다. AWS AppSync의 이러한 기능을 사용하면 다음과 같은 작업을 수행할 수 있습니다.

- 단일 쿼리에서 키 목록을 전달하고 테이블의 결과 반환
- 단일 쿼리의 하나 이상의 테이블에서 레코드 읽기
- 트랜잭션의 레코드를 하나 이상의 테이블에 전부 또는 전무 방식으로 기록
- 일부 조건이 충족되면 트랜잭션 실행

권한

다른 해석기와 마찬가지로 AWS AppSync에서 데이터 소스를 생성하고 역할을 생성하거나 기존 역할을 사용해야 합니다. 트랜잭션 작업에는 DynamoDB 테이블에 대한 여러 권한이 필요하기 때문에 읽기 또는 쓰기 작업에 대해 구성된 역할 권한을 부여해야 합니다.

```
{
    "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/*"
            ]
        }
    ]
}
```

참고: 역할은 AWS AppSync의 데이터 소스에 연결되어 있고 필드의 해석기는 데이터 소스에 대해 호출됩니다. DynamoDB에 대해 가져오도록 구성된 데이터 소스에는 구성을 간단하기 유지하기 위해 지정된 테이블이 하나뿐입니다. 따라서 단일 해석기에서 여러 테이블에 대해 트랜잭션 작업을 수행하는 경우(고급 작업임) 데이터 원본 액세스에 대한 역할을 해석기가 상호 작용하는 모든 테이블에 부여해야 합니다. IAM 정책의 리소스 필드에서 부여할 수 있습니다. 테이블에 대한 트랜잭션 호출의 구성은 해석기 템플릿에서 수행되는데, 이 내용은 아래에서 설명합니다.

데이터 원본

간단하게 설명하기 위해 이 자습서에서 사용되는 모든 해석기에 대해 동일한 데이터 원본을 사용합니다. 데이터 소스 탭에서 새 DynamoDB 데이터 소스를 생성하고 TransactTutorial이라고 이름을 지정합니다. 테이블 이름은 트랜잭션 작업을 위한 요청 매핑 템플릿의 일부로 지정되므로 어떤 것이든 지정할 수 있습니다. 테이블의 이름을 `empty`라고 지정하겠습니다.

`accountNumber`를 파티션 키로 사용하는 savingAccounts 및 checkingAccounts라는 두 개의 테이블과 `transactionId`를 파티션 키로 사용하는 transactionHistory 테이블이 있습니다.

이 자습서의 경우 다음 인라인 정책이 적용된 역할이 모두 작동합니다. `region` 및 `accountId`를 리전 및 계정 ID로 바꿉니다.

```
{
    "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/*"
            ]
        }
    ]
}
```

트랜잭션

이 예에서 컨텍스트는 전형적인 은행 거래이며 여기서 다음을 위해 `TransactWriteItems`를 사용합니다.

- 저축 계좌에서 당좌 예금 계좌로 돈을 이체
- 각 트랜잭션에 대한 새 트랜잭션 레코드 생성

그런 다음 `TransactGetItems`를 사용하여 저축 계좌와 당좌 예금 계좌의 세부 정보를 검색합니다.

GraphQL 스키마는 다음과 같이 정의합니다.

```
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 - 계좌 정보 채우기

계좌 간에 돈을 이체하려면 테이블에 세부 정보를 채워 넣어야 합니다. 이를 위해 GraphQL 작업 `Mutation.populateAccounts`를 사용하겠습니다.

스키마 섹션에서 `Mutation.populateAccounts` 작업 옆에 있는 연결을 클릭합니다. VTL 유닛 해석기로 이동한 다음 동일한 `TransactTutorial` 데이터 소스를 선택합니다.

이제 다음 요청 매핑 템플릿을 사용합니다.

요청 매핑 템플릿

```
#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)
}
```

그리고 다음 응답 매핑 템플릿을 사용합니다.

응답 매핑 템플릿

```
#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)
```

해석기를 저장하고 AppSync AWS 콘솔의 쿼리 섹션으로 이동하여 계좌 정보를 채웁니다.

다음 변형을 실행합니다.

```
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
    }
  }
}
```

하나의 변형으로 3개의 저축 계좌와 3개의 당좌 예금 계좌 정보를 채웠습니다.

DynamoDB 콘솔을 사용하여 데이터가 savingAccounts 및 checkingAccounts 테이블 모두에 표시되는지 확인합니다.

TransactWriteItems - 송금

다음 요청 매핑 템플릿을 사용하여 `transferMoney` 뮤테이션에 해석기를 첨부합니다. `amounts`, `savingAccountNumbers` 및 `checkingAccountNumbers`의 값은 동일합니다.

```
#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)
}
```

한 번의 TransactWriteItems 작업으로 3건의 은행 거래를 하겠습니다. 다음 응답 매핑 템플릿을 사용합니다.

```
#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)
```

이제 AWS AppSync 콘솔의 쿼리 섹션으로 이동하여 다음과 같이 transferMoney 뮤테이션을 실행합니다.

```
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
    }
  }
}
```

하나의 변형으로 2건의 은행 거래를 보냈습니다. DynamoDB 콘솔을 사용하여 데이터가 savingAccounts, checkingAccounts 및 transactionHistory 테이블에 표시되는지 확인합니다.

TransactGetItems - 계좌 검색

한 번의 트랜잭션 요청으로 저축 계좌와 당좌 예금 계좌의 세부 정보를 검색하기 위해 스키마의 `Query.getAccounts` GraphQL 작업에 해석기를 연결하겠습니다. 연결을 선택하고 VTL 유닛 해석기로 이동한 다음 화면에서 자습서 시작 부분에서 생성한 것과 같은 `TransactTutorial` 데이터 소스를 선택합니다. 다음과 같이 템플릿을 구성합니다.

요청 매핑 템플릿

```
#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)
}
```

응답 매핑 템플릿

```
#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)
```

해석기를 저장하고 AppSync AWS 콘솔의 쿼리 섹션으로 이동합니다. 저축 계좌와 당좌 예금 계좌를 검색하려면 다음 쿼리를 실행합니다.

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
      username
      balance
    }
  }
}
```

AWS AppSync를 사용하여 DynamoDB 트랜잭션을 사용하는 방법을 보여드렸습니다.

자습서: HTTP 해석기

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync에서는 지원되는 데이터 소스(즉 AWS Lambda, Amazon DynamoDB, Amazon OpenSearch Service 또는 Amazon Aurora)을 사용하여 다양한 작업을 수행하고 모든 임의 HTTP 엔드포인트를 통해 GraphQL 필드를 해석합니다. HTTP 엔드포인트가 사용 가능해진 후에는 데이터 원본에 연결할 수 있습니다. 쿼리, 변형 및 구독 등 GraphQL 작업을 수행하도록 스키마의 해석기를 구성할 수 있습니다. 이 자습서에서는 몇 가지 일반적인 예제를 살펴봅니다.

이 자습서에서는 AWS AppSync GraphQL 엔드포인트에 REST API(Amazon API Gateway 및 Lambda를 사용하여 생성)를 사용합니다.

원클릭 설치

HTTP 엔드포인트가 구성된(Amazon API Gateway 및 Lambda 사용) AWS AppSync에서 GraphQL 엔드포인트를 자동으로 설치하려는 경우 다음 AWS CloudFormation 템플릿을 사용할 수 있습니다.

REST API 만들기

다음 AWS CloudFormation 템플릿을 사용하여 이 자습서에 사용되는 REST 엔드포인트를 설정할 수 있습니다.

AWS CloudFormation 스택은 다음 단계를 수행합니다.

1. 마이크로서비스에 대한 비즈니스 로직이 포함된 Lambda 함수를 설정합니다.
2. 다음 엔드포인트/메서드/콘텐츠 유형 조합으로 API Gateway REST API를 설정합니다.

API 리소스 경로	HTTP 메서드	지원되는 콘텐츠 유형
/v1/사용자	POST	application/json
/v1/사용자	GET	application/json
/v1/사용자/1	GET	application/json
/v1/사용자/1	PUT	application/json
/v1/사용자/1	DELETE	application/json

GraphQL API 생성

AWS AppSync에서 GraphQL API를 생성하려면:

- AWS AppSync 콘솔을 열고 API 생성을 선택합니다.
- API에 `UserData`를 입력합니다.
- 사용자 지정 스키마를 선택합니다.
- 생성을 선택합니다.

AWS AppSync 콘솔에서 API 키 인증 모드를 사용하여 새 GraphQL API를 생성합니다. 콘솔에서 GraphQL API의 나머지 부분을 설정하고 이 자습서의 나머지 부분에서 이 API에 대한 쿼리를 실행할 수 있습니다.

GraphQL 스키마 생성

GraphQL API가 마련되었으면 GraphQL 스키마를 생성해 보십시오. AWS AppSync 콘솔의 스키마 편집기에서 스키마가 다음 스키마와 일치하는지 확인합니다.

```
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
}
```

HTTP 데이터 원본 구성

HTTP 데이터 원본을 구성하려면 다음을 수행하십시오.

- DataSources(데이터 원본) 탭에서 New(새로 만들기)를 선택한 후, 데이터 원본에 대해 친숙한 이름(예: HTTP)을 입력합니다.
- Data source type(데이터 원본 형식)에서 HTTP를 선택합니다.
- 엔드포인트를 생성된 API Gateway 엔드포인트로 설정합니다. 단계 이름을 엔드포인트의 일부로 포함하지 않았는지 확인합니다.

참고: 현재 AWS AppSync는 퍼블릭 엔드포인트만 지원합니다.

참고: AWS AppSync 서비스에서 인식하는 인증 기관에 대한 자세한 내용은 HTTPS 엔드포인트에 대해 AWS AppSync에서 인식하는 인증 기관(CA)을 참조하세요.

해석기 구성

이 단계에서는 http 데이터 원본을 getUser 쿼리에 연결합니다.

해석기를 설정하려면:

- 스키마 탭을 선택합니다.
- 오른쪽에 있는 Data types(데이터 형식) 창의 쿼리 형식에서 getUser 필드를 찾은 다음 연결을 선택합니다.
- Data source name(데이터 원본 이름)에서 HTTP를 선택합니다.
- Configure the request mapping template(요청 매핑 템플릿 구성)에 다음 코드를 붙여 넣습니다.

```
{
    "version": "2018-05-29",
    "method": "GET",
    "params": {
        "headers": {
            "Content-Type": "application/json"
        }
    },
    "resourcePath": $util.toJson("/v1/users/${ctx.args.id}")
}
```

- Configure the response mapping template(응답 매핑 템플릿 구성)에 다음 코드를 붙여 넣습니다.

```
## 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
```

- 쿼리 탭을 선택한 후, 다음 쿼리를 실행합니다.

```
query GetUser{
    getUser(id:1){
```

```
        id
        username
    }
}
```

그러면 다음과 같은 응답이 반환됩니다.

```
{
    "data": {
        "getUser": {
            "id": "1",
            "username": "nadia"
        }
    }
}
```

- 스키마 탭을 선택합니다.
- 오른쪽에 있는 Data types(데이터 형식) 창의 변형에서 addUser 필드를 찾은 다음 연결을 선택합니다.
- Data source name(데이터 원본 이름)에서 HTTP를 선택합니다.
- Configure the request mapping template(요청 매핑 템플릿 구성)에 다음 코드를 붙여 넣습니다.

```
{
    "version": "2018-05-29",
    "method": "POST",
    "resourcePath": "/v1/users",
    "params":{
      "headers":{
        "Content-Type": "application/json",
      },
      "body": $util.toJson($ctx.args.userInput)
    }
}
```

- Configure the response mapping template(응답 매핑 템플릿 구성)에 다음 코드를 붙여 넣습니다.

```
## 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
```

- 쿼리 탭을 선택한 후, 다음 쿼리를 실행합니다.

```
mutation addUser{
    addUser(userInput:{
        id:"2",
        username:"shaggy"
    }){
        id
        username
    }
}
```

그러면 다음과 같은 응답이 반환됩니다.

```
{
    "data": {
        "getUser": {
        "id": "2",
        "username": "shaggy"
        }
    }
}
```

AWS 서비스 간접 호출

HTTP 해석기를 사용하여 AWS 서비스에 대한 GraphQL API 인터페이스를 설정할 수 있습니다. AWS에 대한 HTTP 요청은 서명 버전 4 프로세스로 서명해야 AWS가 요청을 보낸 사람을 식별할 수 있습니

다. AWS AppSync는 IAM 역할을 HTTP 데이터 소스와 연결할 때 사용자를 대신하여 서명을 계산합니다.

HTTP 해석기를 사용하여 AWS 서비스를 호출하기 위해 다음 두 개의 추가 구성 요소를 제공합니다.

- AWS 서비스 API를 호출할 권한이 있는 AWS IAM 역할
- 데이터 원본의 서명 구성

예를 들어 HTTP 해석기를 사용하여 ListGraphqlApis 작업을 호출하려면 먼저 다음 정책을 연결하여 AWS AppSync가 가정하는 IAM 역할을 생성합니다.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "appsync:ListGraphqlApis"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

다음에는 AWS AppSync에 대한 HTTP 데이터 소스를 생성합니다. 이 예에서는 미국 서부(오레곤) 리전에서 AWS AppSync를 호출합니다. 서명 리전과 서비스 이름이 포함된 `http.json`이라는 파일에서 다음 HTTP 구성을 설정합니다.

```
{
    "endpoint": "https://appsync.us-west-2.amazonaws.com/",
    "authorizationConfig": {
        "authorizationType": "AWS_IAM",
        "awsIamConfig": {
            "signingRegion": "us-west-2",
            "signingServiceName": "appsync"
        }
    }
}
```

그런 다음, AWS CLI를 사용하여 다음과 같이 연결된 역할이 있는 데이터 소스를 생성합니다.

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

해석기를 스키마의 필드에 연결하는 경우 다음 요청 매핑 템플릿을 사용하여 AWS AppSync를 호출합니다.

```
{
    "version": "2018-05-29",
    "method": "GET",
    "resourcePath": "/v1/apis"
}
```

이 데이터 소스에 대해 GraphQL 쿼리를 실행하면 AWS AppSync는 제공한 역할을 사용하여 요청에 서명하고 서명을 요청에 포함시킵니다. 쿼리는 해당 AWS 리전에서 계정에 있는 AWS AppSync GraphQL API의 목록을 반환합니다.

자습서: Aurora Serverless

AWS AppSync는 데이터 API로 활성화된 Amazon Aurora Serverless 클러스터에 대해 SQL 명령을 실행하기 위한 데이터 소스를 제공합니다. AppSync 해석기를 사용하면 GraphQL 쿼리, 변형 및 구독과 관련된 데이터 API에 대해 SQL 문을 실행할 수 있습니다.

클러스터 생성

AppSync에 RDS 데이터 소스를 추가하기 전에 먼저 Aurora Serverless 클러스터에서 데이터 API를 사용하도록 설정하고 AWS Secrets Manager를 사용하여 암호를 구성해야 합니다. AWS CLI를 사용하여 Aurora Serverless 클러스터를 처음 생성할 수 있습니다.

```
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
```

그러면 클러스터에 대한 ARN이 반환됩니다.

다음과 같이 이전 단계의 USERNAME 및 COMPLEX_PASSWORD를 사용하는 것처럼 입력 파일로 AWS Secrets Manager 콘솔 또는 CLI를 통해 암호를 생성합니다.

```
{
    "username": "USERNAME",
    "password": "COMPLEX_PASSWORD"
}
```

생성한 암호는 AWS CLI에 파라미터로 전달합니다.

```
aws secretsmanager create-secret --name HttpRDSSecret --secret-string file://creds.json
 --region us-east-1
```

그러면 암호에 대한 ARN이 반환됩니다.

데이터 원본을 생성할 때 AppSync 콘솔에서 나중에 사용할 수 있도록 암호화 Aurora Serverless 클러스터의 ARN을 적어 둡니다.

데이터 API 활성화

RDS 문서의 지침에 따라 클러스터에 대한 데이터 API를 활성화할 수 있습니다. AppSync 데이터 원본으로 추가하기 전에 데이터 API를 활성화해야 합니다.

데이터베이스 및 테이블 생성

데이터 API를 활성화하면 AWS CLI에서 데이터 API가 `aws rds-data execute-statement` 명령과 작동하는지 확인할 수 있습니다. 그러면 AppSync API에 추가하기 전에 Aurora Serverless 클러스터가 올바르게 구성되도록 할 수 있습니다. 먼저 다음과 같이 `--sql` 파라미터를 사용하여 TESTDB라는 데이터베이스를 생성합니다.

```
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"
```

오류 없이 실행되면 create table 명령을 사용하여 테이블을 추가합니다.

```
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"
```

모든 것이 문제 없이 실행되면 AppSync API에서 해당 클러스터를 데이터 원본으로 추가하는 단계로 진행할 수 있습니다.

GraphQL 스키마

Aurora Serverless 데이터 API가 테이블과 함께 실행 중이므로 변형 및 구독을 수행하기 위해 GraphQL 스키마를 생성하고 해석기를 연결해 보겠습니다. AWS AppSync 콘솔에서 새 API를 생성하고 스키마 페이지로 이동해 다음과 같이 입력합니다.

```
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
}
```

스키마를 저장하고 데이터 원본 페이지로 이동해 새 데이터 원본을 생성합니다. 데이터 원본 유형으로 관계형 데이터베이스를 선택하고 기억하기 쉬운 이름을 입력합니다. 마지막 단계에서 생성한 데이터베이스 이름과 해당 데이터베이스를 생성한 클러스터 ARN을 사용합니다. 역할의 경우 AppSync에서 새 역할을 생성하거나 아래와 유사한 정책을 사용해 역할을 하나 생성하도록 할 수 있습니다.

```
{
    "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:*"
            ]
        }
    ]
}
```

이 정책에는 역할 액세스 권한을 부여하는 명령문이 2개 있습니다. 첫 번째 리소스는 Aurora Serverless 클러스터이고 두 번째는 AWS Secrets Manager ARN입니다. 생성을 클릭하기 전에 AppSync 데이터 소스 구성에서 두 ARN을 모두 제공해야 합니다.

해석기 구성

이제 유효한 GraphQL 스키마와 RDS 데이터 원본이 있으므로 해석기를 스키마에 대한 GraphQL 필드에 연결할 수 있습니다. API는 다음과 같은 기능을 제공합니다.

1. Mutation.createPet 필드를 통해 애완 동물 생성
2. Mutation.updatePet 필드를 통해 애완 동물 업데이트
3. Mutation.deletePet 필드를 통해 애완 동물 삭제
4. Query.getPet 필드를 통해 단일 애완 동물 가져오기
5. Query.listPets 필드를 통해 모든 애완 동물 나열
6. Query.listPetsByPriceRange 필드를 통해 가격 범위 내 애완 동물 나열

Mutation.createPet

AWS AppSync 콘솔의 스키마 편집기에서 오른쪽에 있는 `createPet(input: CreatePetInput!): Pet`용 해석기 연결을 선택합니다. RDS 데이터 원본을 선택합니다. 요청 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
#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)
    }
}
```

설명문 배열의 순서에 따라 SQL 문이 순차적으로 실행됩니다. 결과 역시 동일한 순서로 반환됩니다. 이것은 뮤테이션이므로 삽입 후에 select 문을 실행하여 커밋된 값을 검색하여 GraphQL 응답 매핑 템플릿을 채웁니다.

응답 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

명령문에 SQL 쿼리가 2개 있기 때문에 `$utils.rds.toJsonString($ctx.result))[1][0])`을 사용하여 데이터베이스에서 반환되는 매트릭스에서 두 번째 결과를 지정해야 합니다.

Mutation.updatePet

AWS AppSync 콘솔의 스키마 편집기에서 오른쪽에 있는 `updatePet(input: UpdatePetInput!): Pet`용 해석기 연결을 선택합니다. RDS 데이터 원본을 선택합니다. 요청 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
{
"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)
    }
}
```

응답 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

Mutation.deletePet

AWS AppSync 콘솔의 스키마 편집기에서 오른쪽에 있는 `deletePet(input: DeletePetInput!): Pet`용 해석기 연결을 선택합니다. RDS 데이터 원본을 선택합니다. 요청 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
{
"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"
    }
}
```

응답 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.getPet

이제 스키마에 대한 변형이 생성되었으므로 개별 항목을 가져와 나열하고 SQL 필터링을 적용하는 방법을 보여주기 위해 쿼리 3개를 연결해 볼 것입니다. AWS AppSync 콘솔의 스키마 편집기에서 오른쪽에 있는 `getPet(id: ID!): Pet`용 해석기 연결을 선택합니다. RDS 데이터 원본을 선택합니다. 요청 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
{
"version": "2018-05-29",
        "statements": [
            $util.toJson("select * from Pets WHERE id=:ID")
    ],
    "variableMap": {
        ":ID": "$ctx.args.id"
    }
}
```

응답 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.listPets

AWS AppSync 콘솔의 스키마 편집기에서 오른쪽에 있는 `getPet(id: ID!): Pet`용 해석기 연결을 선택합니다. RDS 데이터 원본을 선택합니다. 요청 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
{
    "version": "2018-05-29",
    "statements": [
        "select * from Pets"
    ]
}
```

응답 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

Query.listPetsByPriceRange

AWS AppSync 콘솔의 스키마 편집기에서 오른쪽에 있는 `getPet(id: ID!): Pet`용 해석기 연결을 선택합니다. RDS 데이터 원본을 선택합니다. 요청 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
{
    "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)
    }
}
```

응답 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

변형 실행

SQL 문을 사용하여 모든 해석기를 구성했고 Serverless Aurora 데이터 API에 GraphQL API를 연결했으므로 변형 및 쿼리 수행을 시작할 수 있습니다. AWS AppSync 콘솔에서 쿼리 탭을 선택하고 다음을 입력해 애완 동물을 생성합니다.

```
mutation add {
    createPet(input : { type:fish, price:10.0 }){
        id
        type
        price
    }
}
```

응답에는 다음과 같이 id, 유형 및 가격이 포함되어 있어야 합니다.

```
{
  "data": {
    "createPet": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "type": "fish",
      "price": "10.0"
    }
  }
}
```

updatePet 변형을 실행하여 항목을 수정할 수 있습니다.

```
mutation update {
    updatePet(input : {
        id: ID_PLACEHOLDER,
        type:bird,
        price:50.0
    }){
        id
        type
        price
    }
}
```

참고로 앞서 createPet 작업에서 반환된 id를 사용했습니다. 해석기에서 `$util.autoId()`를 사용했기 때문에 이 ID는 레코드에 대해 고유한 값이 될 것입니다. 레코드는 다음과 같이 유사한 방식으로 삭제할 수 있습니다.

```
mutation delete {
    deletePet(input : {id:ID_PLACEHOLDER}){
        id
        type
        price
    }
}
```

첫 번째 변형을 사용해 가격에 대한 값을 다르게 해 레코드를 몇 개 생성한 다음 쿼리를 실행합니다.

쿼리 실행

콘솔의 Queries(쿼리) 탭에서 다음 명령문을 사용해 생성한 레코드를 모두 나열합니다.

```
query allpets {
    listPets {
        id
        type
        price
    }
}
```

이 방법도 좋지만, 다음 GraphQL 쿼리를 사용하여 Query.listPetsByPriceRange에 대한 매핑 템플릿에 `where price > :MIN and price < :MAX`가 있는 SQL WHERE 조건자를 활용해 보겠습니다:

```
query petsByPriceRange {
    listPetsByPriceRange(min:1, max:11) {
        id
        type
        price
    }
}
```

가격이 $1 이상이거나 $10 미만인 레코드만 표시되어야 합니다. 마지막으로, 다음과 같이 개별 레코드를 검색하는 쿼리를 수행할 수 있습니다.

```
query onePet {
```

```
    getPet(id:ID_PLACEHOLDER){
        id
        type
        price
    }
}
```

입력 폐기

개발자는 SQL 인젝션 공격으로부터 보호하기 위해 variableMap을 사용하는 것이 좋습니다. 변수 맵을 사용하지 않는 경우 개발자는 GraphQL 작업의 인수를 삭제해야 합니다. 폐기 방법 중 하나는 데이터 API에 대해 SQL 문을 실행하기 전에 요청 매핑 템플릿에 입력 관련 확인 단계를 제공하는 것입니다. listPetsByPriceRange 예제의 요청 매핑 템플릿을 수정하는 방법을 살펴보겠습니다. 전적으로 사용자 입력에만 의존하는 대신 다음과 같이 수행할 수 있습니다.

```
#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)
    }
}
```

데이터 API에 대해 해석기를 실행할 때 무단 입력을 방지하기 위한 또 다른 방법은 저장 프로시저 및 파라미터화된 입력과 함께 준비된 명령문을 사용하는 것입니다. 예를 들어, listPets에 대한 해석기에서 준비된 명령문으로 select를 실행하는 다음 프로시저를 정의합니다.

```
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
```

이러한 프로시저는 다음 sql 명령을 사용하여 Aurora Serverless 인스턴스에서 생성할 수 있습니다.

```
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"
```

이제 저장 프로시저만 호출하면 되므로 결과적으로 listPets에 대한 해석기 코드가 간단해졌습니다. 최소한 문자열 입력에서는 작은 따옴표를 이스케이프 처리해야 합니다.

```
#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("'", "''"))
    }
}
```

이스케이프 문자열

작은 따옴표는 SQL 문에서 문자열 리터럴의 시작과 끝을 나타냅니다(예: `'some string value'`). 문자열 내에서 하나 이상의 작은 따옴표 문자(')가 있는 문자열 값을 사용할 수 있게 하려면 각 문자열 값을 두 개의 작은 따옴표('')로 바꿔야 합니다. 예를 들어 입력 문자열이 `Nadia's dog`인 경우 SQL 문에 대해 이스케이프 처리합니다.

```
update Pets set type='Nadia''s dog' WHERE id='1'
```

자습서: 파이프라인 해석기

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync에서는 유닛 해석기를 통해 GraphQL 필드를 단일 데이터 소스에 연결하는 간단한 방법을 제공합니다. 그러나 단일 작업을 실행하는 것만으로는 충분하지 않을 수 있습니다. 파이프라인 해석기는 데이터 원본에 대해 작업을 순차적으로 실행하는 기능을 제공합니다. API에서 함수를 생성하고 파이프라인 해석기에 해당 작업을 연결합니다. 각 함수 실행 결과는 실행할 함수가 없을 때까지 다음으로 전송됩니다. 파이프라인 해석기를 사용하여 이제 AWS AppSync에서 직접 보다 복잡한 워크플로를 빌드할 수 있습니다. 이 자습서에서는 사용자가 사진을 게시하고, 친구들이 공유한 사진을 볼 수 있는 간단한 사진 보기 앱을 빌드합니다.

원클릭 설치

AWS AppSync에서 구성된 모든 해석기와 필수 AWS 리소스를 모두 사용하여 GraphQL 엔드포인트를 자동으로 설정하려면 다음 AWS CloudFormation 템플릿을 사용합니다.

이 스택은 계정에 다음 리소스를 생성합니다.

- AWS AppSync가 계정 내 리소스에 액세스할 수 있는 IAM 역할
- DynamoDB 테이블 2개
- Amazon Cognito 사용자 풀 1개
- Amazon Cognito 사용자 풀 그룹 2개
- Amazon Cognito 사용자 풀 사용자 3명
- AWS AppSync API 1개

AWS CloudFormation 스택 생성 프로세스가 종료되면 생성된 Amazon Cognito 사용자 3명 각자에 대해 이메일 한 통을 받습니다. 각 이메일에는 Amazon Cognito 사용자로 AWS AppSync 콘솔에 로그인

하는 데 사용할 수 있는 임시 암호가 들어 있습니다. 자습서의 나머지 부분에서 사용할 수 있도록 암호를 저장해 둡니다.

수동 설정

AWS AppSync 콘솔에서 단계별 프로세스를 직접 수행하려면 아래 설정 프로세스를 따르세요.

AWS AppSync 리소스가 아닌 리소스 설정

API는 DynamoDB 테이블 2개 즉, 사진을 저장하는 사진 테이블과 사용자 간의 관계를 저장하는 친구 테이블과 통신합니다. API는 인증 유형으로 Amazon Cognito 사용자 풀을 사용하도록 구성되어 있습니다. 다음 AWS CloudFormation 스택은 계정에 이러한 리소스를 설정합니다.

AWS CloudFormation 스택 생성 프로세스가 종료되면 생성된 Amazon Cognito 사용자 3명 각자에 대해 이메일 한 통을 받습니다. 각 이메일에는 Amazon Cognito 사용자로 AWS AppSync 콘솔에 로그인하는 데 사용할 수 있는 임시 암호가 들어 있습니다. 자습서의 나머지 부분에서 사용할 수 있도록 암호를 저장해 둡니다.

GraphQL API 생성

AWS AppSync에서 GraphQL API를 생성하려면:

1. AWS AppSync 콘솔을 열고 처음부터 빌드를 선택하고 시작을 선택합니다.
2. API의 이름을 `AppSyncTutorial-PicturesViewer`로 설정합니다.
3. 생성을 선택합니다.

AWS AppSync 콘솔에서 API 키 인증 모드를 사용하여 새 GraphQL API를 생성합니다. 콘솔에서 GraphQL API의 나머지 부분을 설정하고 이 자습서의 나머지 부분에서 이 API에 대한 쿼리를 실행할 수 있습니다.

GraphQL API 구성

방금 생성한 Amazon Cognito 사용자 풀을 사용하여 AWS AppSync API를 구성해야 합니다.

1. [Settings] 탭을 선택합니다.
2. Authorization Type(권한 부여 유형) 섹션에서 Amazon Cognito User Pool(Amazon Cognito 사용자 풀)을 선택합니다.

3. 사용자 풀 구성에서 AWS 리전으로 US-WEST-2를 선택합니다.
4. AppSyncTutorial-UserPool 사용자 풀을 선택합니다.
5. 기본 작업으로 거부를 선택합니다.
6. AppId client regex 필드는 비워둡니다.
7. Save를 선택합니다.

이제 API가 인증 유형으로 Amazon Cognito 사용자 풀을 사용하도록 설정되어 있습니다.

DynamoDB 테이블에 대한 데이터 소스 구성

DynamoDB 테이블을 생성한 후에는 콘솔에서 AWS AppSync GraphQL API로 이동하여 데이터 소스 탭을 선택합니다. 이제 방금 생성한 각 DynamoDB 테이블에 대해 AWS AppSync에서 데이터 소스를 생성합니다.

1. 데이터 원본 탭을 선택합니다.
2. 새로 만들기를 선택하여 새 데이터 원본을 생성합니다.
3. 데이터 원본 이름에 `PicturesDynamoDBTable`을 입력합니다.
4. Data Source Type(데이터 원본 유형)으로 Amazon DynamoDB table(Amazon DynamoDB 테이블)을 선택합니다.
5. 리전으로 US-WEST-2를 선택합니다.
6. 테이블 목록에서 AppSyncTutorial-Pictures DynamoDB 테이블을 선택합니다.
7. 새 역할 생성 또는 기존 역할 사용 섹션에서 기존 역할을 선택합니다.
8. CloudFormation 템플릿에서 방금 생성한 역할을 선택합니다. ResourceNamePrefix를 변경하지 않은 경우 역할 이름은 AppSyncTutorial-DynamoDBRole이어야 합니다.
9. 생성을 선택합니다.

친구 테이블에 대해 동일한 프로세스를 반복합니다. CloudFormation 스택을 생성할 때 ResourceNamePrefix 파라미터를 변경하지 않은 경우 DynamoDB 테이블의 이름은 AppSyncTutorial-Friends가 되어야 합니다.

GraphQL 스키마 생성

이제 데이터 소스가 DynamoDB 테이블에 연결되었으며 GraphQL 스키마를 생성해 보겠습니다. AWS AppSync 콘솔의 스키마 편집기에서 스키마가 다음 스키마와 일치하는지 확인합니다.

```
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!
}
```

스키마를 저장하려면 Save Schema(스키마 저장)를 선택합니다.

일부 스키마 필드에는 @aws_auth 명령으로 주석이 달려 있습니다. API 기본 작업 구성이 DENY로 설정되어 있기 때문에 API에서는 @aws_auth 명령 내에 언급된 그룹의 구성원이 아닌 사용자는 모두 거부합니다. API를 보호하는 방법에 대한 자세한 내용은 보안 페이지를 참조하십시오. 이 경우에는 관리자 사용자만 Mutation.createPicture 및 Mutation.createFriendship 필드에 대한 액세스 권한을 가지고 있으며, Admins 또는 Viewers 그룹의 구성원인 사용자는 Query.getPicturesByOwner 필드에 액세스할 수 있습니다. 다른 모든 사용자는 액세스 권한이 없습니다.

해석기 구성

이제 유효한 GraphQL 스키마와 데이터 원본이 2개 있으므로 해석기를 스키마에 대한 GraphQL 필드에 연결할 수 있습니다. API는 다음과 같은 기능을 제공합니다.

- Mutation.createPicture 필드를 통해 사진 생성

- Mutation.createFriendship 필드를 통해 친구 관계 생성
- Query.getPicture 필드를 통해 사진 가져오기

Mutation.createPicture

AWS AppSync 콘솔의 스키마 편집기에서 오른쪽에 있는 `createPicture(input: CreatePictureInput!): Picture!`용 해석기 연결을 선택합니다. DynamoDBPicturesDynamoDBTable 데이터 소스를 선택합니다. 요청 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
#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)
}
```

응답 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

사진 생성 기능이 완료되었습니다. 임의로 생성되는 UUID를 사진 ID로, Cognito 사용자 이름을 사진의 소유자로 사용하여 사진 테이블에 사진을 저장합니다.

Mutation.createFriendship

AWS AppSync 콘솔의 스키마 편집기에서 오른쪽에 있는 `createFriendship(id: ID!, target: ID!): Boolean`용 해석기 연결을 선택합니다. DynamoDBFriendsDynamoDBTable 데이터 소스를 선택합니다. 요청 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
#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)
    }
}
```

중요: BatchPutItem 요청 템플릿에 DynamoDB 테이블의 정확한 이름이 있어야 합니다. 기본 테이블 이름은 AppSyncTutorial-Friends입니다. 잘못된 테이블 이름을 사용하면 AppSync에서 제공된 역할을 수행하려고 할 때 오류가 발생합니다.

이 자습서에서는 간단한 설명을 위해 친구 관계 요청이 승인되어 친구 관계 항목이 AppSyncTutorialFriends 테이블에 직접 저장된 것처럼 진행합니다.

효율적으로, 관계는 양방향이기 때문에 각 친구 관계에 대해 항목 2개를 저장합니다. 다대다 관계를 나타내기 위한 Amazon DynamoDB 모범 사례에 대한 자세한 내용은 DynamoDB 모범 사례를 참조하세요.

응답 매핑 템플릿 섹션에 다음 템플릿을 추가합니다.

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
true
```

참고: 요청 템플릿에 올바른 테이블 이름이 포함되어 있는지 확인합니다. 기본 이름은 AppSyncTutorial-Friends인데, CloudFormation ResourceNamePrefix 파라미터를 변경한 경우에는 테이블 이름이 약간 다를 수 있습니다.

Query.getPicturesByOwner

이제, 친구 관계 및 사진이 있으므로 사용자가 친구의 사진을 볼 수 있는 기능을 제공해야 합니다. 이러한 요구 사항을 충족하기 위해 먼저 요청자가 소유자의 친구인지 확인하고 마지막으로 사진을 쿼리해야 합니다.

이 기능에는 두 가지 데이터 원본 작업이 필요하기 때문에 함수 2개를 생성해 보겠습니다. 첫 번째 함수 isFriend는 요청자와 소유자가 친구인지 확인합니다. 두 번째 함수 getPicturesByOwner는 제공된 소유자 ID를 사용해 요청된 사진을 가져옵니다. 아래에서 Query.getPicturesByOwner 필드에 대해 제안된 해석기에 대한 실행 흐름을 살펴보겠습니다.

1. Before 매핑 템플릿: 컨텍스트와 필드 입력 파라미터를 준비합니다.
2. isFriend 함수: 요청자가 사진의 소유자인지 확인합니다. 소유자가 아닌 경우 친구 테이블에 대해 DynamoDB GetItem 작업을 수행하여 요청자와 소유자 사용자가 친구인지 확인합니다.
3. getPicturesByOwner 함수: owner-index 글로벌 보조 인덱스에 대한 DynamoDB 쿼리 작업을 사용하여 사진 테이블에서 사진을 검색합니다.
4. After 매핑 템플릿: DynamoDB 속성이 예상되는 GraphQL 유형 필드에 정확하게 매핑되도록 사진 결과를 매핑합니다.

먼저 함수를 생성해 보겠습니다.

isFriend 함수

1. Functions(함수) 탭을 선택합니다.
2. Create Function(함수 생성)을 선택하여 함수를 생성합니다.
3. 데이터 원본 이름에 `FriendsDynamoDBTable`을 입력합니다.
4. 함수 이름으로 isFriend를 입력합니다.
5. 요청 매핑 템플릿의 텍스트 영역 내에서 다음 템플릿을 붙여 넣습니다.

```
#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. 응답 매핑 템플릿의 텍스트 영역 내에서 다음 템플릿을 붙여 넣습니다.

```
#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. 함수 생성을 선택합니다.

결과: isFriend 함수를 생성했습니다.

getPicturesByOwner 함수

1. Functions(함수) 탭을 선택합니다.
2. Create Function(함수 생성)을 선택하여 함수를 생성합니다.
3. 데이터 원본 이름에 `PicturesDynamoDBTable`을 입력합니다.
4. 함수 이름으로 `getPicturesByOwner`를 입력합니다.
5. 요청 매핑 템플릿의 텍스트 영역 내에서 다음 템플릿을 붙여 넣습니다.

```
{
    "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. 응답 매핑 템플릿의 텍스트 영역 내에서 다음 템플릿을 붙여 넣습니다.

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end

$util.toJson($ctx.result)
```

7. 함수 생성을 선택합니다.

결과: getPicturesByOwner 함수를 생성했습니다. 이제 함수를 생성했으므로 Query.getPicturesByOwner 필드에 파이프라인 해석기를 연결합니다.

AWS AppSync 콘솔의 스키마 편집기에서 오른쪽에 있는 `Query.getPicturesByOwner(id: ID!): [Picture]`용 해석기 연결을 선택합니다. 다음 페이지에서는 데이터 원본 드롭다운 목록에 아래에 표시되는 Convert to pipeline resolver(파이프라인 해석기로 변환) 링크를 선택합니다. Before 매핑 템플릿에 다음을 사용합니다.

```
#set($result = { "owner": $ctx.args.id, "callerId": $ctx.identity.username })
$util.toJson($result)
```

after mapping template(사후 매핑 템플릿) 섹션에서 다음을 사용합니다.

```
#foreach($picture in $ctx.result.items)
    ## prepend "src://" to picture.src property
    #set($picture['src'] = "src://${picture['src']}")
#end
$util.toJson($ctx.result.items)
```

Create Resolver(해석기 생성)를 선택합니다. 이제 첫 번째 파이프라인 해석기를 성공적으로 연결했습니다. 동일한 페이지에서 이전에 생성한 함수 2개를 추가합니다. 함수 섹션에서 Add A Function(함수 추가)를 선택하고 첫 번째 함수의 이름을 선택하거나 isFriend를 입력합니다. getPicturesByOwner 함수에 대해 동일한 프로세스를 수행하여 두 번째 함수를 추가합니다. isFriend 함수가 목록 맨 위에 나타나고 그 다음에 getPicturesByOwner 함수가 표시되는지 확인합니다. 위쪽 및 아래쪽 화살표를 사용하여 파이프라이니 내 함수의 실행 순서를 다시 정렬할 수 있습니다.

이제, 파이프라인 해석기가 생성되었고, 함수를 연결했으므로 새로 생성된 GraphQL API를 테스트해 보겠습니다.

GraphQL API 테스트

먼저, 생성한 관리자 사용자를 사용하여 몇 가지 변형을 실행해 사진과 친구 관계를 채워야 합니다. AWS AppSync 콘솔 왼쪽에서 쿼리 탭을 선택합니다.

createPicture 변형

1. AWS AppSync 콘솔에서 쿼리 탭을 선택합니다.
2. Login With User Pools(사용자 풀로 로그인)를 선택합니다.
3. 모달에서 CloudFormation 스택에서 생성한 Cognito 샘플 클라이언트 ID(예: 37solo6mmhh7k4v63cqdfgdg5d)를 입력합니다.
4. 파라미터로 CloudFormation 스택에 전달한 사용자 이름을 입력합니다. 기본값은 nadia입니다.
5. CloudFormation 스택에 파라미터로 입력한 이메일로 전송된 임시 암호를 사용합니다(예: UserPoolUserEmail).
6. Login(로그인)을 선택합니다. 버튼이 Logout nadia로 바뀌거나 CloudFormation 스택을 생성할 때 사용자 이름(즉, UserPoolUsername)으로 바뀌어야 합니다.

몇 가지 createPicture 변경을 전송해 사진 테이블을 채워 보겠습니다. 콘솔 내에서 다음 GraphQL 쿼리를 실행합니다.

```
mutation {
  createPicture(input:{
    owner: "nadia"
    src: "nadia.jpg"
  }) {
    id
    owner
```

```
    src
  }
}
```

응답은 아래와 같아야 합니다.

```
{
  "data": {
    "createPicture": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "owner": "nadia",
      "src": "nadia.jpg"
    }
  }
}
```

사진을 몇 개 더 추가해 보겠습니다.

```
mutation {
  createPicture(input:{
    owner: "shaggy"
    src: "shaggy.jpg"
  }) {
    id
    owner
    src
  }
}
```

```
mutation {
  createPicture(input:{
    owner: "rex"
    src: "rex.jpg"
  }) {
    id
    owner
    src
  }
}
```

nadia를 관리자 사용자로 사용하여 사진을 3개 더 추가했습니다.

createFriendship 변형

친구 관계 항목을 추가해 보겠습니다. 콘솔에서 다음 변형을 실행합니다.

참고: 관리자 사용자로 로그인한 상태여야 합니다(기본 관리자 사용자: nadia).

```
mutation {
  createFriendship(id: "nadia", target: "shaggy")
}
```

응답이 다음과 같아야 합니다.

```
{
  "data": {
    "createFriendship": true
  }
}
```

nadia 및 shaggy는 친구이지만 rex는 그 누구와도 친구가 아닙니다.

getPicturesByOwner 쿼리

이 단계에서는 Cognito 사용자 풀과 이 자습서 초반에 설정한 자격 증명을 사용하여 nadia로 로그인합니다. nadia로 shaggy가 소유하고 있는 사진을 검색합니다.

```
query {
    getPicturesByOwner(id: "shaggy") {
        id
        owner
        src
    }
}
```

nadia와 shaggy는 친구이기 때문에 이 쿼리는 해당하는 사진을 반환해야 합니다.

```
{
  "data": {
    "getPicturesByOwner": [
      {
        "id": "05a16fba-cc29-41ee-a8d5-4e791f4f1079",
        "owner": "shaggy",
        "src": "src://shaggy.jpg"
```

```
      }
    ]
  }
}
```

마찬가지로, nadia가 자신의 사진을 검색하려고 하는 경우에도 성공합니다. 파이프라인 해석기는 이 경우 isFriend GetItem 작업 실행을 피하도록 최적화되어 있습니다. 다음 쿼리를 시도해 보십시오.

```
query {
    getPicturesByOwner(id: "nadia") {
        id
        owner
        src
    }
}
```

(Settings(설정) 창에서) API에 대한 로깅을 활성화한 경우 디버그 수준을 ALL로 설정하고 동일한 쿼리를 다시 실행하면 필드 실행에 대한 로그가 반환됩니다. 로그를 보면 isFriend 함수가 요청 매핑 템플릿 단계에서 초기에 반환되었는지 여부를 확인할 수 있습니다.

```
{
  "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
}
```

earlyReturnedValue 키는 #return 명령으로 반환된 데이터를 나타냅니다.

마지막으로, rex가 Viewers Cognito UserPool의 구성원이긴 하지만 rex는 그 누구와도 친구가 아니기 때문에 shaggy 또는 nadia가 소유한 어떤 사진에도 액세스할 수 없습니다. 콘솔에서 rex로 로그인해 다음 쿼리를 실행해 보십시오.

```
query {
    getPicturesByOwner(id: "nadia") {
        id
        owner
        src
    }
}
```

다음과 같은 권한 없음 오류가 발생합니다.

```
{
  "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"
```

```
    }
  ]
}
```

지금까지 파이프라인 해석기를 사용하여 복잡한 권한 부여를 성공적으로 구현해 보았습니다.

자습서: 델타 동기화

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync의 클라이언트 애플리케이션은 GraphQL 응답을 모바일/웹 애플리케이션의 디스크로 로컬로 캐싱해 데이터를 저장합니다. 버전이 지정된 데이터 원본 및 Sync 작업을 통해 고객은 해석기 하나로 동기화 프로세스를 수행할 수 있습니다. 따라서 클라이언트가 수 많은 레코드가 있을 수 있는 기본 쿼리 하나의 결과로 로컬 캐시를 하이드레이션한 다음 마지막 쿼리 이후 변경된 데이터만 수신할 수 있습니다(델타 업데이트). 클라이언트가 최초 요청을 사용한 캐시의 기본 하이드레이션과 다른 요청의 증분 업데이트를 수행하도록 해 클라이언트 애플리케이션에서 백엔드로 컴퓨팅을 이전할 수 있습니다. 이는 온라인과 오프라인 상태 간에 자주 전환하는 클라이언트 애플리케이션에 훨씬 더 효율적입니다.

Delta Sync를 구현하기 위해 Sync 쿼리는 버전이 지정된 데이터 원본에 대한 Sync 작업을 사용합니다. AWS AppSync 뮤테이션이 버전 지정된 데이터 소스의 항목을 변경하면 해당 변경의 레코드는 델타 테이블에도 저장됩니다. 버전이 지정된 다른 데이터 원본에 대해 서로 다른 델타 테이블(예: 유형당 하나, 도메인 영역당 하나)을 사용하거나 API에 대해 단일 델타 테이블을 사용하도록 선택할 수 있습니다. AppSync에서는 기본 키의 충돌을 피하기 위해 여러 API에 단일 델타 테이블을 사용하지 않을 것을 권장합니다.

또한 Delta Sync 클라이언트는 구독을 인수로 수신할 수 있고, 그런 다음 오프라인-온라인 전환 간에 구독 다시 연결 및 쓰기를 조정합니다. Delta Sync는 지수 백오프, 여러 네트워크 오류 시나리오 간에 지터를 사용한 재시도 및 대기열에 이벤트 저장을 비롯하여 구독을 자동으로 재개해 이를 수행합니다. 그런 다음 대기열의 모든 이벤트를 병합하여 최종적으로 구독을 정상적으로 처리하기 전에 적절한 델타 또는 기본 쿼리가 실행됩니다.

Amplify DataStore를 포함한 클라이언트 구성 옵션에 대한 문서는 Amplify Framework 웹사이트를 참조하십시오. 이 문서에서는 최적의 데이터 액세스를 위해 Delta Sync 클라이언트와 작동하도록 버전이 지정된 DynamoDB 데이터 원본과 Sync 작업을 설정하는 방법을 설명합니다.

원클릭 설치

구성된 모든 해석기와 필수 AWS 리소스를 모두 사용해 AWS AppSync에서 GraphQL 엔드포인트를 자동으로 설정하려면 이 AWS CloudFormation 템플릿을 사용합니다.

이 스택은 계정에 다음 리소스를 생성합니다.

- DynamoDB 테이블 2개(기본 및 델타)
- API 키가 있는 AWS AppSync API 1개
- DynamoDB 테이블에 대한 정책이 연결된 IAM 역할 1개

클라이언트가 오프라인 상태였을 때 누락된 이벤트 저널로 작동하는 두 번째 테이블로 동기화 쿼리를 분할하는 데 테이블 2개가 사용됩니다. 델타 테이블에 대해 쿼리를 효율적으로 유지하기 위해 Amazon DynamoDB TTL을 사용하면 필요에 따라 이벤트를 자동으로 정리됩니다. TTL 시간은 데이터 원본의 필요에 맞게 구성할 수 있습니다(1시간, 1일 등).

Schema

델타 동기화를 시연하기 위해 샘플 애플리케이션은 DynamoDB의 기본 및 델타 테이블을 기반으로 하는 Posts 스키마를 생성합니다. AWS AppSync는 두 테이블에 뮤테이션을 자동으로 기록합니다. 동기화 쿼리는 기본 또는 델타 테이블에서 레코드를 적절하게 가져오고 재연결 로직에서 클라이언트가 이러한 테이블을 어떻게 활용하는지 보여주도록 단일 구독이 정의됩니다.

```
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
}
```

GraphQL 스키마가 표준이긴 하지만 계속 진행하기 전에 첫째, 모든 변형이 자동으로 먼저 기본 테이블에 기록한 다음 델타 테이블에 기록합니다. 기본 테이블은 상태에 대한 중앙 출처인 반면에 델타 테이블은 저널입니다. `lastSync: AWSTimestamp`를 전달하지 않으면 `syncPosts` 쿼리가 기본 테이블에 대해 실행되어 캐시를 하이드레이트하고, 클라이언트가 델타 테이블에서 구성한 TTL 시간보다 오랫동안 오프라인 상태일 때 에지 케이스에 대한 글로벌 캐치업 프로세스로 주기적으로 실행됩니다. `lastSync: AWSTimestamp`를 전달하지 않으면 `syncPosts` 쿼리는 델타 테이블에 대해 실행되고 클라이언트가 마지막 오프라인 상태였던 이후 변경된 이벤트를 검색하는 데 사용됩니다. Amplify 클라이언트는 자동으로 `lastSync: AWSTimestamp` 값을 전달하고 디스크에 적절하게 계속 씁니다.

Post의 _deleted 필드는 삭제 작업에 사용됩니다. 클라이언트가 오프라인 상태이고 레코드가 기본 테이블에서 제거된 경우 이 속성은 클라이언트에 동기화를 수행하여 로컬 캐시에서 항목을 제거하라고 알립니다. 클라이언트가 장기간 오프라인 상태로 유지되고 Delta Sync 쿼리를 사용하여 클라이언트가 이 값을 검색하기 전에 항목이 제거되면 (클라이언트에서 구성 가능한) 기본 쿼리의 전역 캐치업 이벤트가 실행되어 캐시에서 항목을 제거합니다. 존재하는 항목을 삭제하는 동기화 쿼리를 실행할 때 값만 반환하기 때문에 이 필드는 선택 사항으로 표시됩니다.

변형

모든 뮤테이션에 대해 AWS AppSync는 기본 테이블에서 표준 생성/업데이트/삭제 작업을 수행하고 델타 테이블의 변경 사항을 자동으로 기록합니다. 데이터 원본에서 `DeltaSyncTableTTL` 값을 수정하여 레코드를 보관할 시간을 줄이거나 늘릴 수 있습니다. 데이터의 속도가 빠른 조직의 경우 이 시간을

짧게 유지하는 것이 합리적일 수 있습니다. 또는 클라이언트가 장기간 오프라인 상태인 경우에는 이 시간을 늘리는 것이 좋을 수 있습니다.

동기화 쿼리

기본 쿼리는 `lastSync` 값이 지정되지 않은 DynamoDB 동기화 작업입니다. 기본 쿼리는 시작 시 그 이후에는 정기적으로만 실행되기 때문에 이는 다수의 조직에 해당하는 내용입니다.

델타 쿼리는 `lastSync` 값이 지정된 DynamoDB 동기화 작업입니다. (기본 쿼리 주기가 실행되지 않는 한) 델타 쿼리는 클라이언트의 상태가 오프라인에서 다시 온라인으로 전환될 때마다 실행됩니다. 클라이언트는 데이터를 동기화하기 위해 마지막으로 쿼리를 실행한 시간을 자동으로 추적합니다.

델타 쿼리가 실행되면 쿼리의 해석기에서 `ds_pk` 및 `ds_sk`를 사용하여 클라이언트가 마지막으로 동기화를 수행한 이후 변경된 레코드에 대해서만 쿼리합니다. 클라이언트는 적절한 GraphQL 응답을 저장합니다.

동기화 쿼리 실행에 대한 자세한 내용은 동기화 작업 설명서를 참조하십시오.

예

먼저 `createPost` 변형을 호출하여 항목을 만듭니다.

```
mutation create {
  createPost(input: {author: "Nadia", title: "My First Post", content: "Hello World"})
 {
    id
    author
    title
    content
    _version
    _lastChangedAt
    _deleted
  }
}
```

이 변형의 반환 값은 다음과 같습니다.

```
{
  "data": {
    "createPost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
```

```
      "author": "Nadia",
      "title": "My First Post",
      "content": "Hello World",
      "_version": 1,
      "_lastChangedAt": 1574469356331,
      "_deleted": null
    }
  }
}
```

기본 테이블의 내용을 검사하면 다음과 같은 레코드를 볼 수 있습니다.

```
{
  "_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"
  }
}
```

델타 테이블의 내용을 검사하면 다음과 같은 레코드를 볼 수 있습니다.

```
{
  "_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"
  }
}
```

이제 syncPosts 쿼리와 같이 클라이언트가 로컬 데이터 저장소를 하이드레이트하기 위해 실행하는 기본 쿼리를 시뮬레이션할 수 있습니다:

```
query baseQuery {
  syncPosts(limit: 100, lastSync: null, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
      _lastChangedAt
    }
    startedAt
    nextToken
  }
}
```

이 기본 쿼리의 반환 값은 다음과 같습니다.

```
{
```

```
  "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
    }
  }
}
```

나중에 startedAt 값을 저장하여 델타 쿼리를 시뮬레이션하겠지만 먼저 테이블을 변경해야 합니다. updatePost 변형을 사용하여 기존 게시물을 수정하겠습니다.

```
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
  }
}
```

이 변형의 반환 값은 다음과 같습니다.

```
{
  "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
    }
  }
}
```

이제 기본 테이블의 내용을 검사하면 업데이트된 항목을 볼 수 있습니다.

```
{
  "_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"
  }
}
```

이제 델타 테이블의 내용을 검사하면 두 개의 레코드를 볼 수 있습니다.

1. 항목이 생성되었을 때의 레코드
2. 항목이 업데이트되었을 때에 대한 레코드.

새 항목은 다음과 같습니다.

```
{
  "_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"
  }
}
```

이제 델타 쿼리를 시뮬레이션하여 클라이언트가 오프라인 상태일 때 발생한 수정 사항을 검색할 수 있습니다. Base 쿼리에서 반환된 `startedAt` 값을 사용하여 요청해 보겠습니다.

```
query delta {
  syncPosts(limit: 100, lastSync: 1574469602238, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
    }
    startedAt
    nextToken
  }
}
```

이 델타 쿼리의 반환 값은 다음과 같습니다.

```
{
  "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
    }
  }
}
```

구성 및 설정

AWS AppSync를 사용하면 다음을 할 수 있습니다.

- 자주 요청되지만 요청마다 변경될 가능성이 거의 없는 데이터를 캐시합니다. 이렇게 하면 해석기의 부하를 줄일 수 있습니다. 자세한 설명은 the section called “캐싱 및 압축” 섹션을 참조하세요.
- GraphQL 객체의 버전을 관리하여 여러 클라이언트 간의 충돌을 처리하고 방지합니다. 자세한 설명은 the section called “충돌 감지 및 동기화” 섹션을 참조하세요.
- 사용자 지정 도메인 이름을 통해 GraphQL과 실시간 API 모두에서 작동하는 기억하기 쉬운 단일 도메인을 구성할 수 있습니다. 자세한 내용은 사용자 지정 도메인 이름 구성 섹션을 참조하십시오.
- VPC를 통해 GraphQL API에 액세스할 수 있도록 허용합니다. 자세한 내용은 AWS AppSync 프라이빗 API 사용 섹션을 참조하십시오.
- 내부 검사를 활성화하고 쿼리당 쿼리 깊이 및 해석기 제한을 설정합니다. 자세한 정보는 구성 제한 섹션을 참조하십시오.

또한 AWS AppSync에는 로깅, 모니터링 및 추적을 위한 다음과 같은 표준 AWS 도구가 포함되어 있습니다.

- AWS CloudTrail에 로그인
- 아마존을 통한 모니터링 CloudWatch
- AWS X-Ray를 사용하여 추적

캐싱 및 압축

AWS AppSync의 서버 측 데이터 캐싱 기능을 사용하면 고속 인 메모리 캐시에서 데이터를 사용할 수 있어 성능이 향상되고 지연 시간이 줄어듭니다. 이렇게 하면 데이터 원본에 직접 액세스할 필요가 줄어듭니다. 캐싱은 유닛 해석기와 파이프라인 해석기 모두에서 사용할 수 있습니다.

또한 AWS AppSync를 사용하면 페이로드 콘텐츠를 더 빠르게 로드하고 다운로드할 수 있도록 API 응답을 압축할 수 있습니다. 이렇게 하면 잠재적으로 애플리케이션에 가해지는 부담을 줄이는 동시에 데이터 전송 요금도 줄일 수 있습니다. 압축 동작은 구성 가능하며 사용자가 임의로 설정할 수 있습니다.

AWS AppSync API에서 원하는 서버 측 캐싱 및 압축 동작을 정의하는 데 도움이 필요하면 이 섹션을 참조하세요.

인스턴스 유형

AWS AppSync는 AWS AppSync API와 동일한 AWS 계정 및 AWS 리전에서 Amazon ElastiCache for Redis 인스턴스를 호스팅합니다.

다음 ElastiCache for Redis 인스턴스 유형을 사용할 수 있습니다.

small

vCPU 1개, 1.5GiB RAM, 낮음에서 중간 수준의 네트워크 성능

medium

vCPU 2개, 3GiB RAM, 낮음에서 중간 수준의 네트워크 성능

large

vCPU 2개, 12.3GiB RAM, 최대 10기가비트 네트워크 성능

xlarge

vCPU 4개, 25.05GiB RAM, 최대 10기가비트 네트워크 성능

2xlarge

vCPU 8개, 50.47GiB RAM, 최대 10기가비트 네트워크 성능

4xlarge

vCPU 16개, 101.38GiB RAM, 최대 10기가비트 네트워크 성능

8xlarge

vCPU 32개, 203.26GiB RAM, 10기가비트 네트워크 성능(일부 리전에만 사용 가능)

12xlarge

vCPU 48개, 317.77GiB RAM, 10기가비트 네트워크 성능

> **Note**
>
> 과거에는 특정 인스턴스 유형(예: `t2.medium`)을 지정했습니다. 2020년 7월부터 이러한 레거시 인스턴스 유형을 계속 사용할 수는 있지만, 지원이 중단되므로 사용하지 않는 것이 좋습니다. 여기에 설명된 일반 인스턴스 유형을 사용하는 것이 좋습니다.

캐싱 동작

다음은 캐싱과 관련된 동작입니다.

없음

서버 측 캐싱이 없습니다.

전체 요청 캐싱

캐시에 데이터가 없으면 데이터 원본에서 데이터를 가져와 TTL(Time To Live)이 만료될 때까지 캐시를 채웁니다. API에 대한 모든 후속 요청은 캐시에서 반환됩니다. 즉, TTL이 만료되지 않는 한 데이터 원본에 직접 연결되지 않습니다. 이 설정에서 캐싱 키로 `context.arguments` 및 `context.identity` 맵의 내용을 사용합니다.

해석기 단위 캐싱

이 설정을 사용하면 응답을 캐시하기 위해 각 해석기를 명시적으로 옵트인해야 합니다. TTL 및 캐싱 키는 해석기에서 지정할 수 있습니다. 지정할 수 있는 캐싱 키는 최상위 맵 `context.arguments`, `context.source`, `context.identity`, 또는 이러한 맵의 문자열 필드입니다. TTL 값은 필수이지만 캐싱 키는 선택 사항입니다. 캐싱 키를 지정하지 않는 경우 기본값은 `context.arguments`, `context.source`, `context.identity` 맵의 내용입니다.

예를 들면 다음과 같은 조합을 사용할 수 있습니다.

- context.arguments 및 context.source
- context.arguments 및 context.identity.sub
- context.arguments.id 또는 context.arguments.InputType.id
- context.source.id 및 context.identity.sub
- context.identity.claims.username

TTL만 지정하고 캐싱 키는 지정하지 않는 경우 해석기의 동작은 전체 요청 캐싱과 동일합니다.

TTL(Time To Live) 캐시

이 설정은 캐시된 항목이 메모리에 저장되는 시간을 정의합니다. 최대 TTL은 3,600초(1시간)이며, 그 이후에는 항목이 자동으로 삭제됩니다.

캐시 암호화

캐시 암호화는 다음 두 가지 유형으로 제공됩니다. 이러한 설정은 ElastiCache for Redis에서 허용하는 설정과 유사합니다. 암호화 설정은 AWS AppSync API에 캐싱을 처음 활성화할 때만 활성화할 수 있습니다.

- 전송 중 암호화 - AWS AppSync, 캐시 및 데이터 원본(안전하지 않은 HTTP 데이터 원본 제외) 간의 요청은 네트워크 수준에서 암호화됩니다. 엔드포인트에서 데이터를 암호화하고 해독하기 위해 몇 가지 처리가 필요하기 때문에 전송 중 암호화는 성능에 영향을 미칠 수 있습니다.
- 저장 데이터 암호화 - 스왑 작업 중에 메모리에서 디스크에 저장된 데이터는 캐시 인스턴스에서 암호화됩니다. 이 설정도 성능에 영향을 줍니다.

캐시 항목을 무효화하려면 AWS AppSync 콘솔 또는 AWS Command Line Interface(AWS CLI)를 사용하여 플러시 캐시 API 호출을 수행할 수 있습니다.

자세한 내용은 AWS AppSync API 참조의 ApiCache 데이터 유형을 참조하세요.

캐시 제거

AWS AppSync의 서버 측 캐싱을 설정하면 최대 TTL을 구성할 수 있습니다. 이 값은 캐시된 항목이 메모리에 저장되는 시간을 정의합니다. 캐시에서 특정 항목을 제거해야 하는 경우 해석기의 요청 또는 응답에 AWS AppSync의 `evictFromApiCache` 확장 유틸리티를 사용할 수 있습니다. (예를 들어 데이터 원본의 데이터가 변경되었는데 캐시 항목이 유효하지 않은 경우) 캐시에서 항목을 제거하려면 항목의 키를 알아야 합니다. 따라서 항목을 동적으로 제거해야 하는 경우 해석기 단위 캐싱을 사용하고 캐시에 항목을 추가하는 데 사용할 키를 명시적으로 정의하는 것이 좋습니다.

캐시 항목 제거

캐시에서 항목을 제거하려면 `evictFromApiCache` 확장 유틸리티를 사용하세요. 유형 이름과 필드 이름을 지정한 다음 키-값 항목의 객체를 제공하여 제거하려는 항목의 키를 빌드합니다. 객체에서 각 키는 캐시된 해석기의 `cachingKey` 목록에서 사용되는 `context` 객체의 유효한 항목을 나타냅니다. 각 값은 키 값을 구성하는 데 사용되는 실제 값입니다. 캐시된 해석기의 `cachingKey` 목록에 있는 캐싱 키와 동일한 순서로 객체에 항목을 넣어야 합니다.

예를 들어 다음 스키마를 참조하세요.

```
type Note {
  id: ID!
```

```
  title: String
  content: String!
}

type Query {
  getNote(id: ID!): Note
}

type Mutation {
  updateNote(id: ID!, content: String!): Note
}
```

이 예에서는 해석기 단위 캐싱을 활성화한 다음 getNote 쿼리에 활성화할 수 있습니다. 그런 다음 [context.arguments.id]를 구성하도록 캐싱 키를 구성할 수 있습니다.

캐시 키를 생성하기 위해 Note를 가져오려고 하면 AWS AppSync는 getNote 쿼리의 id 인수를 사용하여 서버 측 캐시에서 조회를 수행합니다.

Note를 업데이트할 때는 다음 요청이 백엔드 데이터 원본에서 해당 메모를 가져오도록 특정 메모의 항목을 제거해야 합니다. 이렇게 하려면 요청 핸들러를 만들어야 합니다.

다음 예제에서는 이 방법을 사용하여 제거를 처리하는 방법을 보여 줍니다.

```
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;
```

또는 응답 핸들러에서 제거를 처리할 수도 있습니다.

updateNote 변형이 처리되면 AWS AppSync는 항목 제거를 시도합니다. 항목이 성공적으로 삭제되면 응답에는 삭제된 항목 수를 보여 주는 extensions 객체에 apiCacheEntriesDeleted 값이 포함됩니다.

```
"extensions": {  "apiCacheEntriesDeleted": 1}
```

자격 증명을 기반으로 캐시 항목 제거

context 객체의 여러 값을 기반으로 캐싱 키를 생성할 수 있습니다.

Amazon Cognito 사용자 풀을 기본 인증 모드로 사용하고 Amazon DynamoDB 데이터 원본이 지원하는 다음 스키마를 예로 들어 보겠습니다.

```
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
}
```

Note 객체 유형은 DynamoDB 테이블에 저장됩니다. 테이블에는 Amazon Cognito 사용자 이름을 프라이머리 키로 사용하고 Note의 id(슬러그)를 파티션 키로 사용하는 복합 키가 있습니다. 이 시스템은 여러 사용자가 프라이빗 Note 객체를 호스팅하고 업데이트할 수 있는 다중 테넌트 시스템으로, 절대 공유되지 않습니다.

읽기 중심의 시스템이므로 [context.identity.username, context.arguments.id]로 구성된 캐싱키가 있는 해석기 단위 캐싱을 사용하여 getNote 쿼리를 캐시합니다. Note가 업데이트되면 특정 Note에 대한 항목을 제거할 수 있습니다. 해석기의 cachingKeys 목록에 지정된 것과 동일한 순서로 객체에 구성 요소를 추가해야 합니다.

다음 예제는 이 작업을 보여 줍니다.

```
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;
```

백엔드 시스템은 Note를 업데이트하고 항목을 제거할 수도 있습니다. 다음 변형을 예로 들어 보겠습니다.

```
type Mutation {
  updateNoteFromBackend(id: ID!, content: String!, username: ID!): Note @aws_iam
}
```

항목을 제거할 수 있지만, 캐싱 키의 구성 요소를 cachingKeys 객체에 추가할 수 없습니다.

다음 예제에서 제거는 해석기의 응답으로 발생합니다.

```
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;
```

AWS AppSync 외부에서 백엔드 데이터가 업데이트된 경우 NONE 데이터 원본을 사용하는 변형을 호출하여 캐시에서 항목을 제거할 수 있습니다.

API 응답 압축

AWS AppSync를 사용하면 클라이언트가 압축된 페이로드를 요청할 수 있습니다. 요청할 경우 압축된 콘텐츠를 선호한다는 요청에 대한 응답으로 API 응답이 압축되어 반환됩니다. 압축된 API 응답은 더 빠르게 로드되고, 콘텐츠가 더 빠르게 다운로드되며, 데이터 전송 요금도 줄어들 수 있습니다.

> **Note**
>
> 압축은 2020년 6월 1일 이후에 생성된 모든 새로운 API에서 사용할 수 있습니다.

AWS AppSync는 가능한 한 객체를 압축합니다. 드문 경우이긴 하지만 AWS AppSync는 현재 용량을 비롯한 다양한 요인을 기반으로 압축을 건너뛸 수 있습니다.

AWS AppSync는 GraphQL 쿼리 페이로드 크기를 1,000~10,000,000바이트로 압축할 수 있습니다. 압축을 활성화하려면 클라이언트가 `gzip` 값과 함께 `Accept-Encoding` 헤더를 전송해야 합니다. 응답(`gzip`)에서 `Content-Encoding` 헤더의 값을 확인하여 압축을 확인할 수 있습니다.

AWS AppSync 콘솔의 쿼리 탐색기는 기본적으로 요청의 헤더 값을 자동으로 설정합니다. 응답이 충분히 큰 쿼리를 실행하면 브라우저 개발자 도구를 사용하여 압축을 확인할 수 있습니다.

사용자 지정 도메인 이름 구성

AWS AppSync를 사용하면 사용자 지정 도메인 이름을 통해 GraphQL과 실시간 API 모두에서 작동하는 기억하기 쉬운 단일 도메인을 구성할 수 있습니다.

즉, 계정의 AWS AppSync API와 연결하는 사용자 지정 도메인 이름을 생성하여 원하는 도메인 이름으로 간단하고 기억하기 쉬운 엔드포인트 URL을 활용할 수 있습니다.

AWS AppSync API를 구성하면 두 엔드포인트가 프로비저닝됩니다.

AWS AppSync GraphQL 엔드포인트:

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync 실시간 엔드포인트:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

사용자 지정 도메인 이름을 사용하면 하나의 도메인을 사용하여 두 엔드포인트와 모두 상호 작용할 수 있습니다. 예를 들어 `api.example.com`을 사용자 지정 도메인으로 구성하면 다음 URL을 사용하여 GraphQL 및 실시간 엔드포인트와 상호 작용할 수 있습니다.

AWS AppSync 사용자 지정 도메인 GraphQL 엔드포인트:

```
https://api.example.com/graphql
```

AWS AppSync 사용자 지정 도메인 실시간 엔드포인트:

```
wss://api.example.com/graphql/realtime
```

> **Note**
>
> AWS AppSync API는 사용자 지정 도메인 이름에 대해 TLS 1.2 및 TLS 1.3만 지원합니다.

도메인 이름 등록 및 구성

AWS AppSync API에 대한 사용자 지정 도메인 이름을 설정하려면 등록된 인터넷 도메인 이름이 있어야 합니다. Amazon Route 53 domain registration을 사용하거나 원하는 타사 도메인 등록부를 사용하여 인터넷 도메인을 등록할 수 있습니다. Route 53에 대한 자세한 내용은 Amazon Route 53 개발자 안내서의 Amazon Route 53이란 무엇인가요?를 참조하세요.

API의 사용자 지정 도메인 이름은 하위 도메인의 이름이거나 등록된 인터넷 도메인의 루트 도메인("zone apex"라고도 함) 이름일 수 있습니다. AWS AppSync에서 사용자 지정 도메인 이름을 생성한 후에는 DNS 공급자의 리소스 레코드를 생성하거나 업데이트하여 API 엔드포인트에 매핑해야 합니다. 이렇게 매핑하지 않으면 사용자 지정 도메인 이름이 목적지인 API 요청은 AWS AppSync에 도달할 수 없습니다.

AWS AppSync에서 사용자 지정 도메인 생성

AWS AppSync API에 대한 사용자 지정 도메인 이름을 생성하면 Amazon CloudFront 배포가 설정됩니다. 사용자 지정 도메인 이름을 CloudFront 배포 도메인 이름에 매핑하려면 DNS 레코드를 설정해야 합니다. 이 매핑은 매핑된 CloudFront 배포를 통해 사용자 지정 도메인 이름 AWS AppSync에 바인딩되는 API 요청을 라우팅하는 데 필요합니다. 또한 사용자 지정 도메인 이름에 대한 인증서를 제출해야 합니다.

사용자 지정 도메인 이름을 설정하거나 해당 인증서를 업데이트하려면 CloudFront 배포를 업데이트하고 사용하려는 AWS Certificate Manager(ACM) 인증서를 설명할 수 있는 권한이 있어야 합니다. 이러한 권한을 부여하려면 다음 AWS Identity and Access Management(IAM) 정책 문을 계정의 IAM 사용자, 그룹 또는 역할에 연결하세요.

```
{
  "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 AppSync는 CloudFront 배포에서 SNI(Server Name Indication)를 활용하여 사용자 지정 도메인 이름을 지원합니다. 필요한 인증서 형식 및 최대 인증서 키 길이를 포함하여 CloudFront 배포에서 사용자 지정 도메인 이름을 사용하는 방법에 대한 자세한 내용은 Amazon CloudFront 개발자 안내서의 CloudFront에서 HTTPS 사용을 참조하세요.

사용자 지정 도메인 이름을 API의 호스트 이름으로 설정하려면 API 소유자가 사용자 지정 도메인 이름에 대한 SSL/TLS 인증서를 제공해야 합니다. 인증서를 제공하려면 다음 중 하나를 수행하세요.

- ACM에서 새 인증서를 요청하거나 `us-east-1` AWS 리전(미국 동부(버지니아 북부))의 타사 인증 기관에서 발행한 인증서를 ACM으로 가져옵니다. ACM에 대한 자세한 내용은 AWS Certificate Manager 사용 설명서에서 AWS Certificate Manager이란 무엇인가요?를 참조하세요.
- IAM 서버 인증서를 제공합니다. 자세한 내용은 IAM 사용 설명서에서 IAM에서 서버 인증서 관리를 참조하세요.

AWS AppSync의 와일드카드 사용자 지정 도메인 이름

AWS AppSync는 와일드카드 사용자 지정 도메인 이름을 지원합니다. 와일드카드 사용자 지정 도메인 이름을 구성하려면 와일드카드 문자(*)를 사용자 지정 도메인의 첫 번째 하위 도메인으로 지정하세요. 이는 루트 도메인의 가능한 모든 하위 도메인을 나타냅니다. 예를 들어 `*.example.com`을 와일드카드 사용자 지정 도메인 이름으로 사용하면 `a.example.com`, `b.example.com` 및 `c.example.com` 등이 하위 도메인이 됩니다. 이러한 모든 하위 도메인은 동일한 도메인으로 라우팅됩니다.

AWS AppSync에서 와일드카드 사용자 지정 도메인 이름을 사용하려면 동일한 도메인에서 여러 사이트를 보호할 수 있는 와일드카드 이름을 포함하는 ACM에서 발급한 인증서를 제공해야 합니다. 자세한 내용은 AWS Certificate Manager 사용 설명서의 ACM 인증서 특성을 참조하세요.

충돌 감지 및 동기화

버전이 지정된 데이터 원본

AWS AppSync는 현재 DynamoDB 데이터 원본의 버전 관리를 지원합니다. 충돌 감지, 충돌 해결 및 동기화 작업에는 `Versioned` 데이터 원본이 필요합니다. 데이터 원본에 버전 관리를 활성화하면 AWS AppSync에서 자동으로 다음을 수행합니다.

- 객체 버전 관리 메타데이터로 항목을 향상시킵니다.
- AWS AppSync 변형을 통한 항목의 변경 사항을 델타 테이블에 기록합니다.
- 구성 가능한 시간 동안 기본 테이블에서 삭제된 항목의 "삭제 표시"를 유지합니다.

버전이 지정된 데이터 원본 구성

DynamoDB 데이터 원본의 버전 관리를 활성화할 때 다음 필드를 지정합니다.

`BaseTableTTL`

기본 테이블에서 삭제된 항목의 "삭제 표시"(항목이 삭제되었음을 나타내는 메타데이터 필드)를 유지하는 시간(분)입니다. 항목이 삭제되는 즉시 제거하려면 이 값을 0으로 설정하면 됩니다. 이 필드는 필수 항목입니다.

`DeltaSyncTableName`

AWS AppSync 변형이 있는 항목에 대한 변경 사항이 저장되는 테이블의 이름입니다. 이 필드는 필수 항목입니다.

`DeltaSyncTableTTL`

델타 테이블의 항목을 유지할 시간(분)입니다. 이 필드는 필수 항목입니다.

델타 동기화 테이블

AWS AppSync는 현재 `PutItem`, `UpdateItem` 및 `DeleteItem` DynamoDB 작업을 사용하는 변형에 대해 델타 동기화 로깅을 지원합니다.

AWS AppSync 변형이 버전 지정된 데이터 원본의 항목을 변경하면 해당 변경의 레코드는 증분 업데이트에 최적화된 델타 테이블에 저장됩니다. 버전이 지정된 다른 데이터 원본에 대해 서로 다른 델타 테이블(예: 유형당 하나, 도메인 영역당 하나)을 사용하거나 API에 대해 단일 델타 테이블을 사용하도록

선택할 수 있습니다. AWS AppSync에서는 기본 키의 충돌을 피하기 위해 여러 API에 단일 델타 테이블을 사용하지 않을 것을 권장합니다.

이 테이블에 필요한 스키마는 다음과 같습니다.

ds_pk

파티션 키로 사용되는 문자열 값입니다. 기본 데이터 원본 이름과 변경이 발생한 날짜의 ISO 8601 형식을 연결하여 구성됩니다(예: `Comments:2019-01-01`).

VTL 매핑 템플릿의 `customPartitionKey` 플래그가 파티션 키의 열 이름으로 설정되면(AWS AppSync 개발자 안내서의 DynamoDB에 대한 해석기 매핑 템플릿 참조에서 확인) `ds_pk`의 형식이 변경되고 기본 테이블의 새 레코드에 파티션 키 값을 추가하여 문자열이 구성됩니다. 예를 들어, 기본 테이블의 레코드에 `1a`의 파티션 키 값과 `2b`의 정렬 키 값이 있는 경우 문자열의 새 값은 `Comments:2019-01-01:1a`가 됩니다.

ds_sk

정렬 키로 사용되는 문자열 값입니다. 변경이 발생한 시간의 ISO 8601 형식, 항목의 프라이머리 키, 항목의 버전을 연결하여 구성됩니다. 이 필드의 조합은 델타 테이블에 있는 모든 항목의 고유성을 보장합니다(예: 시간이 `09:30:00`, ID가 `1a`, 버전이 2인 경우 `09:30:00:1a:2`).

VTL 매핑 템플릿의 `customPartitionKey` 플래그가 파티션 키의 열 이름으로 설정되면(AWS AppSync 개발자 안내서의 DynamoDB에 대한 해석기 매핑 템플릿 참조에서 확인) `ds_sk`의 형식이 변경되고 기본 테이블에서 조합 키 값을 정렬 키 값으로 대체하여 문자열이 구성됩니다. 위의 예제를 사용하자면, 기본 테이블의 레코드에 `1a`의 파티션 키 값과 `2b`의 정렬 키 값이 있는 경우 문자열의 새 값은 `09:30:00:2b:3`가 됩니다.

_ttl

델타 테이블에서 항목을 제거해야 할 때 타임스탬프를 Epoch 초 단위로 저장하는 숫자 값입니다. 이 값은 데이터 원본에 구성된 `DeltaSyncTableTTL` 값을 변경이 발생한 순간에 추가하여 결정됩니다. 이 필드는 DynamoDB TTL 속성으로 구성해야 합니다.

기본 테이블과 함께 사용하도록 구성된 IAM 역할에는 델타 테이블에서 작업할 수 있는 권한도 포함되어야 합니다. 이 예제에서는 `Comments`라는 기본 테이블과 `ChangeLog`라는 델타 테이블에 대한 권한 정책이 표시됩니다.

```
{
    "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/*"
            ]
        }
    ]
}
```

버전이 지정된 데이터 원본 메타데이터

AWS AppSync는 사용자를 대신하여 `Versioned` 데이터 원본의 메타데이터 필드를 관리합니다. 이러한 필드를 직접 수정하면 애플리케이션에 오류가 발생하거나 데이터가 손실될 수 있습니다. 이러한 필드는 다음과 같습니다.

`_version`

항목이 변경될 때마다 업데이트되는 단순 증가 카운터입니다.

`_lastChangedAt`

항목이 마지막으로 수정되었을 때 타임스탬프를 Epoch 밀리초 단위로 저장하는 숫자 값입니다.

`_deleted`

항목이 삭제되었음을 나타내는 부울 "삭제 표시" 값입니다. 로컬 데이터 스토어에서 삭제된 항목을 제거하기 위해 애플리케이션에서 사용할 수 있습니다.

`_ttl`

기본 데이터 원본에서 항목을 제거해야 할 때 타임스탬프를 Epoch 초 단위로 저장하는 숫자 값입니다.

ds_pk

델타 테이블의 파티션 키로 사용되는 문자열 값입니다.

ds_sk

델타 테이블의 정렬 키로 사용되는 문자열 값입니다.

gsi_ds_pk

글로벌 보조 인덱스를 파티션 키로 지원하기 위해 생성된 문자열 값 특성입니다. VTL 매핑 템플릿에서 `customPartitionKey` 및 `populateIndexFields` 플래그가 모두 활성화된 경우에만 포함됩니다(AWS AppSync 개발자 안내서의 DynamoDB에 대한 해석기 매핑 템플릿 참조에서 확인). 활성화된 경우 기본 데이터 원본 이름과 변경이 발생한 날짜의 ISO 8601 형식을 연결하여 값이 구성됩니다(예: 기본 테이블의 이름이 Comments인 경우 이 레코드는 `Comments:2019-01-01`로 설정됨).

gsi_ds_sk

글로벌 보조 인덱스를 정렬 키로 지원하기 위해 생성된 문자열 값 특성입니다. VTL 매핑 템플릿에서 `customPartitionKey` 및 `populateIndexFields` 플래그가 모두 활성화된 경우에만 포함됩니다(AWS AppSync 개발자 안내서의 DynamoDB에 대한 해석기 매핑 템플릿 참조에서 확인). 활성화된 경우 변경이 발생한 시간의 ISO 8601 형식, 기본 테이블에 있는 항목의 파티션 키, 기본 테이블에 있는 항목의 정렬 키, 항목의 버전을 연결하여 값이 구성됩니다(예: 시간이 `09:30:00`, 파티션 키 값이 `1a`, 정렬 키 값이 `2b`, 버전이 `3`인 경우 `09:30:00:1a#2b:3`).

이러한 메타데이터 필드는 기본 데이터 원본에 있는 항목의 전체 크기에 영향을 줍니다. AWS AppSync는 애플리케이션을 설계할 때 버전이 지정된 데이터 원본 메타데이터를 위해 500바이트 이상의 최대 프라이머리 키 크기 스토리지를 예약할 것을 권장합니다. 클라이언트 애플리케이션에서 이 메타데이터를 사용하려면 GraphQL 유형 및 변형 선택 집합에 `_version`, `_lastChangedAt` 및 `_deleted` 필드를 포함합니다.

충돌 감지 및 해결

AWS AppSync에서 동시 쓰기가 발생하는 경우 충돌 감지 및 충돌 해결 전략을 구성하여 업데이트를 적절하게 처리할 수 있습니다. 충돌 감지는 변형이 데이터 원본에 실제 기록된 항목과 충돌하는지 판단합니다. `conflictDetection` 필드에 대한 SyncConfig의 값을 `VERSION`으로 설정하면 충돌 감지가 활성화됩니다.

충돌 해결은 충돌이 감지될 경우 수행되는 작업입니다. 이는 SyncConfig의 충돌 핸들러 필드를 설정하여 결정됩니다. 다음과 같은 세 가지 충돌 해결 전략이 있습니다.

- OPTIMISTIC_CONCURRENCY
- AUTOMERGE
- LAMBDA

이러한 각 충돌 해결 전략은 아래에 자세히 설명되어 있습니다.

버전은 쓰기 작업 중에 AppSync에 의해 자동으로 증가하며, 버전이 활성화된 데이터 원본으로 구성된 해석기 외부 또는 클라이언트에서 수정하면 안 됩니다. 이렇게 하면 시스템의 일관성 동작이 변경되어 데이터가 손실될 수 있습니다.

낙관적 동시성

낙관적 동시성은, 버전이 지정된 데이터 원본에 대해 AWS AppSync가 제공하는 충돌 해결 전략입니다. 충돌 해석기가 낙관적 동시성으로 설정된 경우, 수신 중인 변형의 버전이 실제 객체 버전과 다른 것으로 감지되면 충돌 핸들러가 수신 중인 요청을 거부합니다. GraphQL 응답 내에, 최신 버전 서버의 기존 항목이 제공됩니다. 그런 다음 클라이언트는 이 충돌을 로컬로 처리하고, 항목 버전이 업데이트된 변형을 다시 시도해야 합니다.

Automerge

Automerge를 사용하면 개발자가 다른 전략으로는 처리할 수 없었던 충돌을 수동으로 병합하는 클라이언트 측 로직을 작성하지 않고도 충돌 해결 전략을 쉽게 구성할 수 있습니다. Automerge는 충돌을 해결하기 위해 데이터를 병합할 때 엄격한 규칙 세트를 준수합니다. Automerge의 원칙은 GraphQL 필드의 기본 데이터 유형을 중심으로 합니다. 내용은 다음과 같습니다.

- 스칼라 필드의 충돌: GraphQL 스칼라 또는 컬렉션이 아닌 필드(목록, 집합, 맵). 스칼라 필드의 수신 값을 거부하고 서버에 있는 값을 선택합니다.
- 목록의 충돌: GraphQL 유형과 데이터베이스 유형은 목록입니다. 수신 목록을 서버의 기존 목록과 연결합니다. 수신 중인 변형의 목록 값이 서버의 목록 끝에 추가됩니다. 중복 값은 유지됩니다.
- 집합의 충돌: GraphQL 유형은 목록이고 데이터베이스 유형은 집합입니다. 서버의 기존 집합과 수신 집합을 사용하여 집합 공용 구조체를 적용합니다. 이는 중복 항목이 없음을 의미하는 집합의 속성을 준수합니다.
- 수신 중인 변형이 항목에 새 필드를 추가하거나 `null` 값이 있는 필드에 대해 만들어진 경우, 이를 기존 항목에 병합합니다.
- 맵의 충돌: 데이터베이스의 기본 데이터 유형이 맵(키-값 문서)인 경우, 맵의 각 속성을 구문 분석하고 처리할 때 위의 규칙을 적용합니다.

Automerge는 업데이트된 버전의 요청을 자동으로 검색, 병합 및 재시도하여 클라이언트가 충돌하는 데이터를 수동으로 병합할 필요가 없도록 합니다.

Automerge가 스칼라 유형에서 충돌을 처리하는 방법의 예를 보여줍니다. 우선, 다음 레코드를 사용하겠습니다.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 4
}
```

현재 수신 중인 변형이 항목을 업데이트하려고 시도할 수 있으나, 클라이언트가 아직 서버와 동기화되지 않았기 때문에 이전 버전으로 업데이트됩니다. 다음과 같은 형태입니다.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 55,
  "_version" : 2
}
```

수신 중인 요청이 이전 버전인 2임을 알 수 있습니다. 이 흐름 동안 Automerge는 'jersey' 필드를 '55'로 업데이트하기를 거부하여 데이터를 병합하고 '5' 값을 유지하므로, 서버에 이 항목이 다음 이미지와 같이 저장됩니다.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 5 # version is incremented every time automerge performs a merge that is
 stored on the server.
}
```

버전 5에서 위 항목의 상태를 감안할 때, 이번에는 수신 중인 변형이 다음 이미지로 항목을 변형하려고 시도한다고 가정해 봅시다.

```
{
  "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
}
```

수신 중인 변형에서 세 가지를 관심 있게 봐야 합니다. 스칼라라는 이름이 변경되었지만 “interests” 집합과 “points” 목록이라는 두 개의 새로운 필드가 추가되었습니다. 이 시나리오에서는 버전 불일치로 인해 충돌이 감지됩니다. Automerge는 속성을 준수하고, 스칼라이기 때문에 이름 변경을 거부하고 충돌하지 않는 필드에 추가합니다. 그러면 서버에 저장된 항목이 다음과 같이 표시됩니다.

```
{
  "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
}
```

항목 이미지는 버전 6으로 업데이트된 가운데, 이번에는 수신 중인 변형(다른 버전 불일치)이 항목을 다음과 같이 변환하려고 한다고 가정합시다.

```
{
  "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
}
```

여기에서 “interests”에 대해 수신 중인 필드를 보면, 서버에 하나의 중복 값이 있고 두 개의 새로운 값이 있습니다. 이 경우 기본 데이터 유형이 집합이므로 Automerge는 서버에 있는 값과 수신 중인 요청의 값을 결합하여 중복을 제거합니다. 마찬가지로 하나의 중복 값과 하나의 새로운 값이 있는 “points” 필드에는 충돌이 있습니다. 그러나 여기에서는 기본 데이터 유형이 목록이므로, Automerge는 수신 중인 요청의 모든 값을 서버에 이미 있는 값의 끝에 추가하기만 합니다. 그 결과, 서버에 저장된 병합 이미지는 다음과 같이 나타납니다.

```
{
```

```
  "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
}
```

이제 서버에 저장된 항목이 버전 8에서 다음과 같이 표시된다고 가정해 봅시다.

```
{
  "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
}
```

그러나 이번에도 버전 불일치로 수신 중인 요청이 다음 이미지로 항목을 업데이트하려고 시도합니다.

```
{
  "id" : 1,
  "name" : "Nadia",
  "stats": {
      "ppg": "25.7",
      "rpg": "6.9"
  }
  "_version" : 3
}
```

이제 이 시나리오에서는 서버에 이미 있는 필드가 없음을 알 수 있습니다(interests, points, jersey). 또한 맵 “stats” 내의 “ppg” 값이 편집되고, 새로운 값 “rpg”가 추가되며, “apg”가 생략되고 있습니다. Automerge는 생략된 필드를 유지하므로(참고: 필드를 제거하려는 경우 일치하는 버전으로 요청을 다시 시도해야 함) 손실되는 법이 없습니다. 또한 맵 내의 필드에 동일한 규칙을 적용하므로 “ppg”에 대한

변경은 거부되지만 “apg”는 보존되고 새 필드인 “rpg”가 추가됩니다. 그 결과 이제 서버에 저장된 항목이 다음과 같이 표시됩니다.

```
{
  "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

충돌 해결 옵션:

- RESOLVE: 기존 항목을 응답 페이로드에 제공된 새 항목으로 바꿉니다. 한 번에 하나의 항목에만 동일한 작업을 다시 시도할 수 있습니다. 현재 DynamoDB PutItem & UpdateItem에 지원됩니다.
- REJECT: 변형을 거부하고 GraphQL 응답의 기존 항목에 오류를 반환합니다. 현재 DynamoDB PutItem, UpdateItem & DeleteItem에 지원됩니다.
- REMOVE: 기존 항목을 제거합니다. 현재 DynamoDB DeleteItem에 지원됩니다.

Lambda 호출 요청

AWS AppSync DynamoDB 해석기가 LambdaConflictHandlerArn에 지정된 Lambda 함수를 호출합니다. 데이터 원본에 대해 구성된 동일한 service-role-arn을 사용합니다. 호출의 페이로드 구조는 다음과 같습니다.

```
{
    "newItem": { ... },
    "existingItem": {... },
    "arguments": { ... },
    "resolver": { ... },
    "identity": { ... }
```

```
}
```

필드는 다음과 같이 정의됩니다.

newItem

미리 보기 항목(변형이 성공한 경우)입니다.

existingItem

현재 DynamoDB 테이블에 있는 항목입니다.

arguments

GraphQL 변형의 인수.

resolver

AWS AppSync 해석기에 대한 정보

identity

호출자에 대한 정보. API 키로 액세스하면 이 필드가 null로 설정됩니다.

페이로드의 예:

```
{
    "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"
    }
}
```

Lambda 호출 응답

PutItem 및 UpdateItem 충돌 해결

변형 RESOLVE. 응답은 다음 형식이어야 합니다.

```
{
    "action": "RESOLVE",
    "item": { ... }
}
```

item 필드는 기본 데이터 원본의 기존 항목을 대체하는 데 사용할 객체를 나타냅니다. item에 포함된 경우 기본 키와 동기화 메타데이터는 무시됩니다.

변형 REJECT. 응답은 다음 형식이어야 합니다.

```
{
    "action": "REJECT"
}
```

DeleteItem 충돌 해결

항목을 REMOVE합니다. 응답은 다음 형식이어야 합니다.

```
{
    "action": "REMOVE"
}
```

변형 REJECT. 응답은 다음 형식이어야 합니다.

```
{
    "action": "REJECT"
}
```

아래의 예제 Lambda 함수는 호출하는 사람과 해석기 이름을 확인합니다. jeffTheAdmin에서 만든 경우 DeletePost 해석기의 객체를 REMOVE하거나, Update/Put 해석기의 새 항목과 충돌을 RESOLVE합니다. 그렇지 않으면 변형이 REJECT됩니다.

```
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;
}
```

오류

`ConflictUnhandled`

충돌 감지는 버전 불일치를 찾아내고, 충돌 핸들러는 변형을 거부합니다.

예: 낙관적 동시성 충돌 핸들러를 사용한 충돌 해결. 또는 `REJECT`와 함께 반환된 Lambda 충돌 핸들러.

`ConflictError`

충돌을 해결하려고 하면 내부 오류가 발생합니다.

예: Lambda 충돌 핸들러가 잘못된 응답을 반환했습니다. 또는 제공된 리소스 `LambdaConflictHandlerArn`을 찾을 수 없어 Lambda 충돌 핸들러를 호출할 수 없습니다.

`MaxConflicts`

충돌 해결을 위한 최대 재시도 횟수에 도달했습니다.

예: 동일한 객체에 대한 동시 요청이 너무 많습니다. 충돌이 해결되기 전에 객체가 다른 클라이언트에 의해 새 버전으로 업데이트됩니다.

`BadRequest`

클라이언트는 메타데이터 필드(`_version`, `_ttl`, `_lastChangedAt`, `_deleted`)를 업데이트하려고 합니다.

예: 클라이언트는 업데이트 변형으로 객체의 _버전을 업데이트하려고 시도합니다.

`DeltaSyncWriteError`

델타 동기화 레코드를 쓰지 못했습니다.

예: 변형이 성공했지만 델타 동기화 테이블에 쓰려고 할 때 내부 오류가 발생했습니다.

`InternalFailure`

내부 오류가 발생했습니다.

CloudWatch Logs

AWS AppSync API가 로깅 설정이 Field-Level Logs `enabled`로 설정되고 Field-Level Logs의 로그 수준이 `ALL`로 설정된 CloudWatch Logs를 활성화한 경우, AWS AppSync가 로그 그룹에 충돌 감지 및 해

결 정보를 방출합니다. 로그 메시지 형식에 대한 자세한 내용은 충돌 탐지 및 동기화 로깅 설명서를 참조하십시오.

동기화 작업

버전이 지정된 데이터 원본은 DynamoDB 테이블에서 모든 결과를 검색한 다음 마지막 쿼리(델타 업데이트) 이후에 변경된 데이터만 수신할 수 있는 Sync 작업을 지원합니다. AWS AppSync가 Sync 작업에 대한 요청을 수신하면 요청에 지정된 필드를 사용하여 기본 테이블 또는 델타 테이블에 액세스해야 하는지 여부를 결정합니다.

- `lastSync` 필드가 지정되지 않으면 기본 테이블에서 `Scan`이 수행됩니다.
- `lastSync` 필드가 지정되었지만 값이 `current moment - DeltaSyncTTL` 이전인 경우 기본 테이블에서 `Scan`이 수행됩니다.
- `lastSync` 필드가 지정되고 값이 `current moment - DeltaSyncTTL` 시점 또는 그 이후인 경우 델타 테이블에서 `Query`가 수행됩니다.

AWS AppSync는 모든 Sync 작업에 대해 응답 매핑 템플릿으로 `startedAt` 필드를 반환합니다. 이 `startedAt` 필드는 로컬로 저장하고 다른 요청에 사용할 수 있는 Sync 작업이 시작된 시간(Epoch 밀리초)입니다. 페이지 매김 토큰이 요청에 포함된 경우, 이 값은 결과의 첫 페이지에 대한 요청에 의해 반환된 값과 동일합니다.

Sync 매핑 템플릿 형식에 대한 자세한 내용은 매핑 템플릿 참조를 참조하십시오.

모니터링 및 로깅

AWS AppSync GraphQL API를 모니터링하고 요청과 관련된 문제를 디버깅하는 데 도움이 되도록 Amazon Logs에 로깅을 활성화할 수 있습니다. CloudWatch

설정 및 구성

GraphQL API에서 자동 로깅을 켜려면 콘솔을 사용하십시오. AWS AppSync

1. 에 AWS Management Console 로그인하고 콘솔을 엽니다. AppSync
2. API 페이지에서 GraphQL API의 이름을 선택합니다.
3. API 홈페이지의 탐색 창에서 설정 을 선택합니다.
4. 로깅 아래에서 다음을 수행합니다.

a. 로그 활성화를 켭니다.

b. 자세한 요청 수준 로깅을 보려면 상세 콘텐츠 포함 아래의 확인란을 선택합니다(선택 사항).

c. 필드 해석기 로그 수준에서 원하는 필드 수준 로깅 수준(없음, 오류 또는 모두)을 선택합니다(선택 사항).

d. 기존 역할 생성 또는 사용에서 새 역할을 선택하여 로그를 쓸 수 있는 새 역할 AWS Identity and Access Management (IAM) AWS AppSync 을 생성합니다. CloudWatch 또는 기존 역할을 선택하여 AWS 계정에 있는 기존 IAM 역할의 Amazon 리소스 이름(ARN)을 선택합니다.

5. 저장을 선택합니다.

수동 IAM 역할 구성

기존 IAM 역할을 사용하기로 선택한 경우 해당 역할은 로그를 쓰는 데 필요한 권한을 AWS AppSync 부여해야 합니다. CloudWatch 이를 수동으로 구성하려면 로그를 작성할 때 역할을 AWS AppSync 수임할 수 있는 서비스 역할 ARN을 제공해야 합니다.

IAM 콘솔에서 다음 정의를 포함하는 `AWSAppSyncPushToCloudWatchLogsPolicy`라는 이름의 새 정책을 생성합니다.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

그런 다음 이름을 AWSAppSyncPushToCloudWatchLogsRole사용하여 새 역할을 생성하고 새로 만든 정책을 역할에 연결합니다. 다음과 같도록 이 역할의 신뢰 관계를 편집합니다.

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

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

역할 ARN을 복사하여 AWS AppSync GraphQL API에 대한 로깅을 설정할 때 사용합니다.

CloudWatch 메트릭

CloudWatch 지표를 사용하여 HTTP 상태 코드 또는 지연 시간으로 인해 발생할 수 있는 특정 이벤트를 모니터링하고 이에 대한 알림을 제공할 수 있습니다. 다음과 같은 지표가 생성됩니다.

지표 목록

`4XXError`

잘못된 클라이언트 구성으로 인해 유효하지 않은 요청으로 인한 오류가 있습니다. 일반적으로 이러한 오류는 GraphQL 처리 외부에서 위치에 상관없이 발생합니다. 예를 들어 요청에 잘못된 JSON 페이로드 또는 잘못된 쿼리가 포함되어 있거나 서비스가 제한되거나 권한 부여 설정이 잘못 구성된 경우 이 오류가 발생할 수 있습니다.

단위: 수. Sum 통계를 사용하여 오류의 총 발생 횟수를 가져옵니다.

`5XXError`

GraphQL 쿼리 실행 중 발생한 오류입니다. 예를 들어 비어 있거나 잘못된 스키마에 대해 쿼리를 간접적으로 호출할 때 이 오류가 발생할 수 있습니다. Amazon Cognito 사용자 풀 ID 또는 AWS 지역이 유효하지 않은 경우에도 발생할 수 있습니다. 또는 요청을 처리하는 동안 문제가 AWS AppSync 발생하는 경우에도 이 문제가 발생할 수 있습니다.

단위: 수. Sum 통계를 사용하여 오류의 총 발생 횟수를 가져옵니다.

`Latency`

클라이언트로부터 요청을 AWS AppSync 받은 시점과 클라이언트에게 응답을 반환하는 시점 사이의 시간. 여기에는 응답이 최종 장치에 도달하는 데 발생한 네트워크 지연 시간은 포함되지 않습니다.

단위: 밀리초. Average 통계를 사용하여 예상 지연 시간을 평가합니다.

`Requests`

리전별로 계정의 모든 API가 처리한 요청(쿼리 + 변형)의 수입니다

단위: 수. 특정 리전에서 처리된 모든 요청의 수입니다.

`TokensConsumed`

토큰은 `Request`가 소비하는 리소스 양(처리 시간 및 사용된 메모리)을 기준으로 `Requests`에 할당됩니다. 일반적으로 각 `Request`에서는 토큰 하나를 소비합니다. 그러나 많은 양의 리소스를 소비하는 `Request`에는 필요에 따라 추가 토큰이 할당됩니다.

단위: 수. 특정 리전에서 처리된 요청에 할당된 토큰의 수입니다.

`NetworkBandwidthOutAllowanceExceeded`

> **Note**
>
> AWS AppSync 콘솔의 캐시 설정 페이지에서 Cache Health Metrics 옵션을 사용하여 이 캐시 관련 상태 지표를 활성화할 수 있습니다.

처리량이 집계된 대역폭 제한을 초과하여 네트워크 패킷이 삭제되었습니다. 이는 캐시 구성의 병목 현상을 진단하는 데 유용합니다. 지표에 를 지정하여 특정 API에 대한 데이터를 기록합니다`API_Id`. `appsyncCacheNetworkBandwidthOutAllowanceExceeded`

단위: 수. ID로 지정된 API의 대역폭 제한을 초과한 후 삭제된 패킷 수입니다.

`EngineCPUUtilization`

> **Note**
>
> AWS AppSync 콘솔의 캐시 설정 페이지에서 Cache Health Metrics 옵션을 사용하여 이 캐시 관련 상태 지표를 활성화할 수 있습니다.

Redis 프로세스에 할당된 CPU 사용률 (백분율). 이는 캐시 구성의 병목 현상을 진단하는 데 유용합니다. 지표에 를 지정하여 특정 API에 대한 데이터를 기록합니다`API_Id`. `appsyncCacheEngineCPUUtilization`

단위: 퍼센트. ID로 지정된 API에 대해 Redis 프로세스에서 현재 사용 중인 CPU 백분율입니다.

실시간 구독

모든 지표는 하나의 차원(GraphQLAPIId)으로 방출됩니다. 즉 모든 지표가 GraphQL API ID와 결합됩니다. 다음 메트릭은 Pure보다 GraphQL 구독과 관련이 있습니다. WebSockets

지표 목록

`ConnectRequests`

성공한 시도와 실패한 시도를 모두 AWS AppSync포함한 WebSocket 연결 요청 수.

단위: 수. Sum 통계를 사용하여 총 연결 요청 수를 가져옵니다.

`ConnectSuccess`

에 성공한 WebSocket 연결 수. AWS AppSync 구독 없이 연결할 수 있습니다.

단위: 수. Sum 통계를 사용하여 성공한 연결의 총 발생 횟수를 가져옵니다.

`ConnectClientError`

클라이언트측 오류로 AWS AppSync 인해 거부된 WebSocket 연결 수입니다. 이는 서비스가 제한되어 있거나 권한 부여 설정이 잘못 구성되었음을 뜻할 수 있습니다.

단위: 수. Sum 통계를 사용하여 클라이언트 측 연결 오류의 총 발생 횟수를 가져옵니다.

`ConnectServerError`

연결을 처리하는 AWS AppSync 동안 발생한 오류 수입니다. 이 오류는 대개 예기치 않은 서버 측에 문제가 생겼을 때 발생합니다.

단위: 수. Sum 통계를 사용하여 서버 측 연결 오류의 총 발생 횟수를 가져옵니다.

`DisconnectSuccess`

에서 성공적으로 WebSocket 연결이 끊긴 횟수. AWS AppSync

단위: 수. Sum 통계를 사용하여 성공한 연결 해제의 총 발생 횟수를 가져옵니다.

`DisconnectClientError`

연결을 WebSocket 끊는 AWS AppSync 동안 발생한 클라이언트 오류의 수입니다.

단위: 수. Sum 통계를 사용하여 연결 해제 오류의 총 발생 횟수를 가져옵니다.

DisconnectServerError

연결을 끊는 AWS AppSync 동안 발생한 서버 오류의 수입니다. WebSocket

단위: 수. Sum 통계를 사용하여 연결 해제 오류의 총 발생 횟수를 가져옵니다.

SubscribeSuccess

을 (를) 통해 성공적으로 등록된 구독 수입니다. AWS AppSync WebSocket 구독 없는 연결은 가능하지만 연결 없는 구독은 불가능합니다.

단위: 수. Sum 통계를 사용하여 성공한 구독의 총 발생 횟수를 가져옵니다.

SubscribeClientError

클라이언트측 오류로 AWS AppSync 인해 거부된 구독 수입니다. JSON 페이로드가 잘못되었거나 서비스가 제한되거나 권한 부여 설정이 잘못 구성된 경우 이러한 오류가 발생할 수 있습니다.

단위: 수. Sum 통계를 사용하여 클라이언트 측 구독 오류의 총 발생 횟수를 가져옵니다.

SubscribeServerError

구독을 처리하는 동안 발생한 오류 수입니다. AWS AppSync 이 오류는 대개 예기치 않은 서버 측에 문제가 생겼을 때 발생합니다.

단위: 수. Sum 통계를 사용하여 서버 측 구독 오류의 총 발생 횟수를 가져옵니다.

UnsubscribeSuccess

성공적으로 처리된 구독 취소 요청의 수입니다.

단위: 수. Sum 통계를 사용하여 성공한 구독 취소 요청의 총 발생 횟수를 가져옵니다.

UnsubscribeClientError

클라이언트측 오류로 AWS AppSync 인해 거부된 구독 취소 요청 수.

단위: 수. Sum 통계를 사용하여 클라이언트 측 구독 취소 요청 오류의 총 발생 횟수를 가져옵니다.

UnsubscribeServerError

구독 취소 요청을 처리하는 AWS AppSync 동안 발생한 오류 수입니다. 이 오류는 대개 예기치 않은 서버 측에 문제가 생겼을 때 발생합니다.

단위: 수. Sum 통계를 사용하여 서버 측 구독 취소 요청 오류의 총 발생 횟수를 가져옵니다.

PublishDataMessageSuccess

성공적으로 게시된 구독 이벤트 메시지의 수입니다.

단위: 수. Sum 통계를 사용하여 성공적으로 게시된 구독 이벤트 메시지의 총 개수를 가져옵니다.

PublishDataMessageClientError

클라이언트 측 오류로 인해 게시하지 못한 구독 이벤트 메시지의 수입니다.

Unit: 수. Sum 통계를 사용하여 클라이언트 측 구독 이벤트 게시 오류의 총 발생 횟수를 가져옵니다.

PublishDataMessageServerError

구독 이벤트 메시지를 게시하는 AWS AppSync 동안 발생한 오류 수입니다. 이 오류는 대개 예기치 않은 서버 측에 문제가 생겼을 때 발생합니다.

단위: 수. Sum 통계를 사용하여 서버 측 게시 구독 이벤트 오류의 총 발생 횟수를 가져옵니다.

PublishDataMessageSize

게시된 구독 이벤트 메시지의 크기입니다.

단위: 바이트

ActiveConnections

1분 동안 클라이언트에서 WebSocket 연결한 동시 연결 수입니다. AWS AppSync

단위: 수. Sum 통계를 사용하여 열린 연결의 총 수를 가져옵니다.

ActiveSubscriptions

1분 동안 클라이언트의 동시 구독 수입니다.

단위: 수. Sum 통계를 사용하여 활성 구독의 총 수를 가져옵니다.

ConnectionDuration

연결이 열린 상태로 유지되는 시간입니다.

단위: 밀리초. 평균 통계를 사용하여 연결 기간을 평가합니다.

OutboundMessages

측정된 메시지 수가 성공적으로 게시되었습니다. 측정된 메시지 1개는 전송된 데이터의 5kB에 해당합니다.

단위: 수. Sum 통계를 사용하면 총 게시된 측정 대상 메시지 수를 알 수 있습니다.

InboundMessageSuccess

성공적으로 처리된 인바운드 메시지 수입니다. 변형에 의해 간접 호출된 각 구독 유형은 하나의 인바운드 메시지를 생성합니다.

단위: 수. Sum 통계를 사용하면 총 처리된 인바운드 메시지 수를 알 수 있습니다.

InboundMessageError

잘못된 API 요청 (예: 240kB 구독 페이로드 크기 제한 초과) 으로 인해 처리에 실패한 인바운드 메시지 수입니다.

단위: 수. Sum 통계를 사용하면 API 관련 처리에 실패한 총 인바운드 메시지 수를 알 수 있습니다.

InboundMessageFailure

의 오류로 인해 처리에 실패한 인바운드 메시지 수입니다. AWS AppSync

단위: 수. 합계 통계를 사용하여 AWS AppSync관련 처리 실패가 발생한 인바운드 메시지의 총 수를 가져올 수 있습니다.

InboundMessageDelayed

지연된 인바운드 메시지 수. 인바운드 메시지 속도 할당량 또는 아웃바운드 메시지 속도 할당량이 위반되면 인바운드 메시지가 지연될 수 있습니다.

단위: 수. 합계 통계를 사용하면 지연된 인바운드 메시지의 총 수를 확인할 수 있습니다.

InboundMessageDropped

삭제된 인바운드 메시지 수. 인바운드 메시지 속도 할당량 또는 아웃바운드 메시지 속도 할당량이 위반되면 인바운드 메시지가 삭제될 수 있습니다.

단위: 수. 합계 통계를 사용하면 삭제된 인바운드 메시지의 총 수를 확인할 수 있습니다.

InvalidationSuccess

`$extensions.invalidateSubscriptions()`의 변형으로 인해 성공적으로 무효화(구독 취소)된 구독의 수입니다.

단위: 수. Sum 통계를 사용하여 구독 취소에 성공한 총 구독 수를 가져옵니다.

InvalidationRequestSuccess

성공적으로 처리된 무효화 요청 수입니다.

단위: 수. Sum 통계를 사용하면 총 처리된 무효화 요청 수를 알 수 있습니다.

InvalidationRequestError

잘못된 API 요청으로 인해 처리에 실패한 무효화 요청의 수입니다.

단위: 수. Sum 통계를 사용하면 API 관련 처리에 실패한 총 무효화 요청 수를 알 수 있습니다.

InvalidationRequestFailure

의 오류로 인해 처리에 실패한 무효화 요청 수입니다. AWS AppSync

단위: 수. 합계 통계를 사용하여 AWS AppSync관련 처리 실패가 발생한 무효화 요청의 총 수를 구할 수 있습니다.

InvalidationRequestDropped

무효화 요청 할당량을 초과했을 때 무효화 요청 수가 삭제되었습니다.

단위: 수. Sum 통계를 사용하여 총 삭제된 무효화 요청 수를 확인할 수 있습니다.

인바운드 메시지와 아웃바운드 메시지 비교

돌연변이가 실행되면 해당 변형에 대한 @aws_subscribe 지시문이 있는 구독 필드가 호출됩니다. 각 구독 호출은 하나의 인바운드 메시지를 생성합니다. 예를 들어 @aws_subscribe 에서 두 개의 구독 필드가 동일한 변형을 지정하는 경우 해당 변형이 호출되면 두 개의 인바운드 메시지가 생성됩니다.

아웃바운드 메시지 1개는 클라이언트에 전송되는 데이터 5kB에 WebSocket 해당합니다. 예를 들어 10개의 클라이언트에 15kB의 데이터를 보내면 30개의 아웃바운드 메시지가 생성됩니다 (15kB * 클라이언트 10개/메시지당 5kB = 메시지 30개).

인바운드 또는 아웃바운드 메시지의 할당량 증가를 요청할 수 있습니다. 자세한 내용은 AWS 일반 참조 가이드의 AWS AppSync 엔드포인트 및 할당량 및 Service Quotas User Guide의 할당량 증가 요청 지침을 참조하십시오.

향상된 지표

향상된 지표는 AWS AppSync 요청 및 오류 수, 지연 시간, 캐시 적중/실패 등 API 사용 및 성능에 대한 세분화된 데이터를 내보냅니다. 모든 향상된 지표 데이터는 사용자 CloudWatch 계정으로 전송되며 전송할 데이터 유형을 구성할 수 있습니다.

> **Note**
>
> 향상된 지표를 사용할 경우 추가 요금이 적용됩니다. 자세한 내용은 Amazon CloudWatch 요금의 세부 모니터링 요금 티어를 참조하십시오.

이러한 지표는 AWS AppSync 콘솔의 다양한 설정 페이지에서 찾을 수 있습니다. API 설정 페이지의 향상된 지표 섹션에서는 다음 항목을 활성화하거나 비활성화할 수 있습니다.

1. Resolver 메트릭 동작: 이 옵션은 리졸버에 대한 추가 메트릭이 수집되는 방식을 제어합니다. 전체 요청 확인자 지표 (요청 중인 모든 해석기에 대해 활성화된 지표) 또는 해석자별 측정치 (구성이 활성화되도록 설정된 확인자에만 활성화된 지표) 를 활성화하도록 선택할 수 있습니다. 다음과 같은 옵션을 사용할 수 있습니다.

지표	지표 차원	지표 이름	단위	설명
리졸버당 GraphQL 오류	API_ID, 리졸버	그래프 QL 오류	개수	리졸버당 발생한 GraphQL 오류 수입니다.
리졸버당 요청 수	API_ID, 리졸버	요청	개수	요청 중에 발생한 호출 수입니다. 이는 리졸버별로 기록됩니다.
리졸버당 지연 시간	API_ID, 리졸버	지연 시간	밀리초	리졸버 호출을 완료하는 데 걸리는 시간. 지연 시간은 밀리초 단위로 측정되며 리졸버별로 기록됩니다.
리졸버당 캐시 히트	API_ID, 리졸버	CacheHit	개수	요청 중 캐시 적중 횟수 캐시를 사용하는 경우에만 발생합니다.

				캐시 히트는 리졸버별로 기록됩니다.
리졸버당 캐시 미스 횟수	API_ID, 리졸버	CacheMiss	개수	요청 중 발생한 캐시 누락 횟수 이는 캐시를 사용하는 경우에만 발생합니다. 캐시 미스는 리졸버별로 기록됩니다.

2. 데이터 소스 지표 동작: 이 옵션은 데이터 소스에 대한 추가 지표가 수집되는 방식을 제어합니다. 전체 요청 데이터 소스 지표 (요청 시 모든 데이터 소스에 대해 활성화된 지표) 또는 데이터 소스별 지표 (구성이 활성화로 설정된 데이터 소스에만 활성화된 지표) 를 활성화하도록 선택할 수 있습니다. 다음과 같은 옵션을 사용할 수 있습니다.

지표	지표 차원	지표 이름	단위	설명
데이터 소스별 요청	API_ID, 데이터 소스	요청	개수	요청 중에 발생한 호출 수. 요청은 데이터 소스별로 기록됩니다. 전체 요청이 활성화된 경우 각 데이터 원본에는 고유한 항목이 있습니다. CloudWatch
데이터 소스별 지연 시간	API_ID, 데이터 소스	지연 시간	밀리초	데이터 소스 호출을 완료하는 데 걸리는 시간. 지연 시간은 데이터 소스별로 기록됩니다.

데이터 소스별 오류	API_ID, 데이터 소스	그래프 QL 오류	개수	데이터 소스 호출 중에 발생한 오류 수입니다.

3. 작업 메트릭: GraphQL 작업 수준 메트릭을 활성화합니다.

지표	메트릭 치수	지표 이름	단위	설명
작업별 요청	API_ID, 오퍼레이션	요청	개수	지정된 GraphQL 작업이 호출된 횟수입니다.
작업별 GraphQL 오류	API_ID, 오퍼레이션	그래프 QL 오류	개수	지정된 GraphQL 작업 중에 발생한 GraphQL 오류 수입니다.

CloudWatch 로그

기존 또는 새 GraphQL API에 대해 두 가지 유형의 로깅 즉, 요청 수준 및 필드 수준 로깅을 구성할 수 있습니다.

요청 수준 로그

요청 수준 로깅(상세 콘텐츠 포함)이 구성되면 다음 정보가 로깅됩니다.

- 소비된 토큰 수
- 요청 및 응답 HTTP 헤더
- 요청에서 실행 중인 GraphQL 쿼리
- 전체 작업 요약
- 등록된 기존 및 새 GraphQL 구독

필드 수준 로그

필드 수준 로깅이 구성되면 다음 정보가 로깅됩니다.

- 각 필드에 대한 소스 및 인수와 함께 생성된 요청 매핑

- 각 필드에 대해 변환된 응답 매핑(필드 해석의 결과로 데이터가 포함됨)
- 각 필드에 대한 추적 정보

로깅을 켜면 CloudWatch 로그를 AWS AppSync 관리합니다. 이 프로세스에는 로그 그룹 및 로그 스트림을 생성하는 작업과 이러한 로그로 로그 스트림에 보고하는 작업이 포함됩니다.

GraphQL API에서 로깅을 켜고 AWS AppSync 요청하면 로그 그룹 아래에 로그 그룹과 로그 스트림을 생성합니다. 로그 그룹은 `/aws/appsync/apis/{graphql_api_id}` 형식에 따라 명명됩니다. 각 로그 그룹 내에서 로그는 로그 스트림으로 나뉩니다. 로깅된 데이터가 보고되면 로그는 마지막 이벤트 시간으로 정렬됩니다.

모든 로그 이벤트에는 해당 요청의 x-amzn - 태그가 지정됩니다. RequestId 이렇게 하면 로그 이벤트를 CloudWatch 필터링하여 해당 요청에 대해 기록된 모든 정보를 가져올 수 있습니다. 모든 GraphQL AWS AppSync 요청의 응답 RequestId 헤더에서 가져올 수 있습니다.

필드 수준 로깅은 다음 로그 수준으로 구성됩니다.

- 없음 - 캡처되는 필드 수준 로그가 없습니다.
- 오류 - 오류가 있는 필드에 대한 다음 정보만 로깅합니다.
    - 서버 응답의 오류 섹션
    - 필드 수준 오류
    - 오류 필드에 대해 해석되어 생성된 요청/응답 함수
- 모두 - 쿼리의 모든 필드에 대해 다음 정보를 로깅합니다.
    - 필드 수준 추적 정보
    - 각 필드에 대해 해석되어 생성된 요청/응답 함수

모니터링의 장점

로깅 및 지표를 사용해 GraphQL 쿼리를 식별 및 최적화하고, 문제를 해결할 수 있습니다. 예를 들어, 쿼리의 각 필드에 대해 로깅된 추적 정보를 사용하여 지연 시간 문제를 디버깅할 수 있습니다. 이를 시연하기 위해 GraphQL 쿼리에서 중첩된 해석기를 하나 이상 사용한다고 가정해 보겠습니다. CloudWatch 로그의 샘플 필드 작업은 다음과 비슷할 수 있습니다.

```
{
    "path": [
        "singlePost",
```

```
        "authors",
        0,
        "name"
    ],
    "parentType": "Post",
    "returnType": "String!",
    "fieldName": "name",
    "startOffset": 416563350,
    "duration": 11247
}
```

이는 GraphQL 스키마에 해당할 수 있고, 다음과 유사할 수 있습니다.

```
type Post {
  id: ID!
  name: String!
  authors: [Author]
}

type Author {
  id: ID!
  name: String!
}

type Query {
  singlePost(id:ID!): Post
}
```

앞선 로그 결과에서 path는 `singlePost()` 쿼리 실행에서 반환된 데이터의 단일 항목을 보여 줍니다. 이 예에서는 첫 번째 인덱스(0)의 이름 필드를 나타냅니다. startOffset은 GraphQL 쿼리 작업의 시작으로부터 오프셋을 제공합니다. duration은 필드를 해석하는 데 걸리는 총 시간입니다. 이러한 값은 특정 데이터 원본의 데이터가 예상보다 느리게 실행되는 이유를 찾거나 특정 필드가 전체 쿼리 실행 속도를 늦추는 문제를 해결하는 데 유용할 수 있습니다. 예를 들어, Amazon DynamoDB 테이블에 대해 프로비저닝된 처리량을 늘리거나 전반적인 작업 수행을 저하하는 특정 필드를 쿼리에서 제거하도록 선택할 수 있습니다.

2019년 5월 8일부터 로그 이벤트를 완전히 구조화된 JSON으로 AWS AppSync 생성합니다. 이를 통해 CloudWatch Logs Insights 및 Amazon Service와 같은 로그 분석 OpenSearch 서비스를 사용하여 GraphQL 요청의 성능과 스키마 필드의 사용 특성을 이해할 수 있습니다. 예를 들어, 성능 문제의 근본 원인일 수 있는 지연 시간이 긴 해석기를 쉽게 식별할 수 있습니다. 또한, 스키마에서 가장 높은/낮은 빈도로 사용되는 필드를 식별하고 GraphQL 필드 사용을 중단하는 경우의 영향을 평가할 수 있습니다.

충돌 감지 및 동기화 로깅

Field resolver CloudWatch 로그 수준이 모두로 설정된 상태로 구성된 로그 로깅이 AWS AppSync API에 있는 경우 충돌 감지 및 해결 정보를 로그 그룹에 AWS AppSync 내보냅니다. 이를 통해 AWS AppSync API가 충돌에 어떻게 대응했는지 세밀하게 파악할 수 있습니다. 응답을 해석하는 데 도움이 되도록 다음 정보가 로그에 제공됩니다.

지표 목록

`conflictType`

충돌이 버전 불일치 때문인지, 고객이 제공한 조건 때문인지 자세한 정보를 제공합니다.

`conflictHandlerConfigured`

요청 시 해석기에 구성된 충돌 핸들러를 지시합니다.

`message`

충돌이 감지되고 해결된 과정에 대한 정보를 제공합니다.

`syncAttempt`

궁극적으로 요청을 거부하기 전에 데이터를 동기화하기 위해 서버가 시도한 시도 횟수입니다.

`data`

구성된 충돌 핸들러가 `Automerge`인 경우 이 필드가 채워져 각 필드에 대해 `Automerge`가 어떤 결정을 취했는지 보여줍니다. 다음과 같은 작업이 제공될 수 있습니다.

- REJECTED - `Automerge`가 서버의 값으로 대체하기 위해 수신 중인 필드 값을 거부하는 경우.
- ADDED - 서버에 기존 값이 없기 때문에 `Automerge`가 수신 중인 필드에 추가하는 경우.
- APPENDED - `Automerge`가 서버에 있는 목록의 값에 수신 중인 값을 추가하는 경우.
- MERGED - `Automerge`가 수신 중인 값을 서버에 있는 집합의 값에 병합하는 경우.

토큰 수를 사용하여 요청 최적화

1,500KB-초 이하의 메모리와 vCPU 시간을 소비하는 요청에는 하나의 토큰이 할당됩니다. 리소스 소비량이 1,500KB-초보다 큰 요청은 추가 토큰을 수신합니다. 예를 들어 요청에 3,350KB/초가 소요되는 경우 요청에 토큰 3개 (다음 정수 값으로 반올림) 를 AWS AppSync 할당합니다. 기본적으로 배포된 지역에 따라 계정의 API에 초당 최대 5,000개 또는 10,000개의 요청 토큰을 AWS AppSync 할당합니다.

AWS API에서 각각 초당 평균 2개의 토큰을 사용하는 경우 요청은 각각 초당 2,500개 또는 5,000개로 제한됩니다. 할당된 양보다 더 많은 초당 토큰이 필요한 경우 요청 토큰 비율의 기본 할당량을 늘려달라는 요청을 제출할 수 있습니다. 자세한 내용은 AWS 일반 참조 가이드의AWS AppSync 엔드포인트 및 할당량 및 Service Quotas 사용 설명서의 할당량 증가 요청을 참조하십시오.

요청당 토큰 수가 많다는 것은 요청을 최적화하고 API 성능을 개선할 기회가 있음을 의미할 수도 있습니다. 요청당 토큰 수를 늘릴 수 있는 요인은 다음과 같습니다.

- GraphQL 스키마의 크기 및 복잡성.
- 요청 및 응답 매핑 템플릿의 복잡성.
- 요청당 해석기 간접 호출 수.
- 해석기에서 반환된 데이터의 양.
- 다운스트림 데이터 원본의 지연 시간.
- 병렬 또는 일괄 호출이 아닌 연속적인 데이터 원본 호출이 필요한 스키마 및 쿼리 설계.
- 로깅 구성, 특히 필드 수준 및 상세 로그 콘텐츠.

> **Note**
>
> AWS AppSync 지표와 로그 외에도 클라이언트는 응답 헤더를 통해 요청에 사용된 토큰 수에 액세스할 수 있습니다. `x-amzn-appsync-TokensConsumed`

로그 유형 참조

RequestSummary

- requestId: 요청에 대한 고유 식별자입니다.
- graphQLAPIId: 요청을 수행하는 GraphQL API의 ID입니다.
- statusCode: HTTP 상태 코드 응답입니다.
- 지연 시간: E 요청 nd-to-end 지연 시간 (나노초) (정수).

```
{
     "logType": "RequestSummary",
     "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
     "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4",
```

```
     "statusCode": 200,
     "latency": 242000000
}
```

ExecutionSummary

- requestId: 요청에 대한 고유 식별자입니다.
- graphQLAPIId: 요청을 수행하는 GraphQL API의 ID입니다.
- startTime: 요청에 대한 GraphQL 처리의 시작 타임스탬프입니다. RFC 3339 형식으로 표시됩니다.
- endTime: 요청에 대한 GraphQL 처리의 종료 타임스탬프입니다. RFC 3339 형식으로 표시됩니다.
- duration: 경과된 총 GraphQL 처리 시간입니다. 나노초 단위의 정수로 표시됩니다.
- 버전: 의 스키마 버전입니다. ExecutionSummary
- parsing:
    - startOffset: 간접 호출과 관련하여 구문 분석에 대한 시작 오프셋입니다. 나노초 단위의 정수로 표시됩니다.
    - duration: 구문 분석에 소비된 시간입니다. 나노초 단위의 정수로 표시됩니다.
- validation:
    - startOffset: 간접 호출과 관련하여 검증에 대한 시작 오프셋입니다. 나노초 단위의 정수로 표시됩니다.
    - duration: 검증을 수행하는 데 소비된 시간입니다. 나노초 단위의 정수로 표시됩니다.

```
{
    "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"
```

```
}
```

추적

- requestId: 요청에 대한 고유 식별자입니다.
- graphQLAPIId: 요청을 수행하는 GraphQL API의 ID입니다.
- startOffset: 간접 호출과 관련하여 필드 해석에 대한 시작 오프셋입니다. 나노초 단위의 정수로 표시됩니다.
- duration: 필드 해석에 소비된 시간입니다. 나노초 단위의 정수로 표시됩니다.
- fieldName: 해석되는 필드의 이름입니다.
- parentType: 해석되는 필드의 상위 유형입니다.
- returnType: 해석되는 필드의 반환 유형입니다.
- path: 응답의 루트에서 시작하고 해석되는 필드로 종료되는 경로 세그먼트의 목록입니다.
- resolverArn: 필드 해석에 사용되는 해석기의 ARN입니다. 중첩 필드에는 표시되지 않을 수 있습니다.

```
{
    "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"
}
```

로그 인사이트를 통한 CloudWatch 로그 분석

다음은 실행 가능한 통찰을 GraphQL 작업의 성능과 상태로 가져오기 위해 실행할 수 있는 쿼리의 예제입니다. 이 예시는 CloudWatch Logs Insights 콘솔에서 샘플 쿼리로 사용할 수 있습니다. CloudWatch

콘솔에서 Logs Insights를 선택하고 GraphQL API의 AWS AppSync 로그 그룹을 선택한 다음 샘플 쿼리에서 쿼리를 AWS AppSync 선택합니다.

다음 쿼리는 최대 토큰이 소비된 상위 10개의 GraphQL 요청을 반환합니다.

```
filter @message like "Tokens Consumed"
| parse @message "* Tokens Consumed: *" as requestId, tokens
| sort tokens desc
| display requestId, tokens
| limit 10
```

다음 쿼리는 최대 지연 시간이 있는 상위 10개의 해석기를 반환합니다.

```
  fields resolverArn, duration
| filter logType = "Tracing"
| limit 10
| sort duration desc
```

다음 쿼리는 가장 자주 호출되는 해석기를 반환합니다.

```
  fields ispresent(resolverArn) as isRes
| stats count() as invocationCount by resolverArn
| filter isRes and logType = "Tracing"
| limit 10
| sort invocationCount desc
```

다음 쿼리는 템플릿 매핑에 가장 많은 오류가 있는 해석기를 반환합니다.

```
  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
```

다음 쿼리는 해석기 지연 시간 통계를 반환합니다.

```
  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
```

다음 쿼리는 필드 지연 시간 통계를 반환합니다.

```
  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
```

CloudWatch Logs Insights 쿼리 결과를 대시보드로 내보낼 CloudWatch 수 있습니다.

서비스로 OpenSearch 로그를 분석하세요.

Amazon OpenSearch Service로 AWS AppSync 로그를 검색, 분석 및 시각화하여 성능 병목 현상과 운영 문제의 근본 원인을 식별할 수 있습니다. 최대 지연 시간과 오류가 있는 해석기를 식별할 수 있습니다. 또한 대시보드를 사용하여 강력한 시각화가 포함된 OpenSearch 대시보드를 만들 수 있습니다. OpenSearch 대시보드는 Service에서 사용할 수 있는 오픈 소스 데이터 시각화 및 탐색 도구입니다. OpenSearch OpenSearch 대시보드를 사용하여 GraphQL 작업의 성능과 상태를 지속적으로 모니터링할 수 있습니다. 예를 들어, GraphQL 요청의 P90 지연 시간을 시각화하고 각 해석기의 P90 지연 시간을 드릴다운하는 대시보드를 생성할 수 있습니다.

OpenSearch 서비스를 사용할 때는 인덱스를 검색하기 위한 필터 패턴으로 “cwl*”를 사용하십시오. OpenSearch OpenSearch 서비스는 Logs에서 스트리밍된 로그를 접두사 “CloudWatch cwl-”로 인덱싱합니다. AWS AppSync API 로그를 OpenSearch 서비스로 전송된 다른 CloudWatch 로그와 구별하려면 검색에 필터 표현식을 추가하는 것이 좋습니다. `graphQLAPIID.keyword=`*`YourGraphQLAPIID`*

로그 형식 마이그레이션

2019년 5월 8일 또는 그 이후에 AWS AppSync 생성되는 로그 이벤트는 완전히 구조화된 JSON으로 포맷됩니다. 2019년 5월 8일 이전의 GraphQL 요청을 분석하려면 샘플에서 제공되는 스크립트를 사용하여 이전 로그를 완전히 구조화된 JSON으로 마이그레이션할 수 있습니다. GitHub 2019년 5월 8일 이전의 로그 형식을 사용해야 하는 경우 다음 설정을 통해 지원 티켓을 생성합니다. Type(유형)을 Account Management(계정 관리)로 설정한 다음 Category(범주)를 General Account Question(일반 계정 질문)으로 설정합니다.

또한 지표 필터를 사용하여 로그 데이터를 수치 CloudWatch 지표로 변환하여 그래프로 표시하거나 경보를 설정할 수 있습니다. CloudWatch

AWS X-Ray를 사용하여 추적

AWS X-Ray를 사용하여 AWS AppSync에서 실행되는 요청을 추적할 수 있습니다. X-Ray를 사용할 수 있는 모든 AWS 리전에서 AWS AppSync와 함께 X-Ray를 사용할 수 있습니다. X-Ray는 전체 GraphQL 요청에 대한 자세한 개요를 제공합니다. 이를 통해 API 및 기본 해석기와 데이터 원본의 지연 시간을 분석할 수 있습니다. X-Ray 서비스 맵을 사용하여 X-Ray와 통합된 모든 AWS 서비스를 포함한 요청의 지연 시간을 확인할 수 있습니다. 또한 샘플링 규칙을 구성하여 지정한 기준에 따라 어떤 요청을 어떤 샘플링 속도로 기록할지를 X-Ray에 지정할 수도 있습니다.

X-Ray의 샘플링에 대한 자세한 내용은 AWS X-Ray 콘솔의 샘플링 규칙 구성을 참조하세요.

설정 및 구성

AWS AppSync 콘솔을 통해 GraphQL API에 대한 AppSync 추적을 활성화할 수 있습니다.

1. AWS AppSync 콘솔에 로그인합니다.
2. 탐색 창에서 설정을 선택합니다.
3. X-Ray에서 Enable X-Ray(X-Ray 활성화)를 설정합니다.
4. Save를 선택합니다. 이제 X-Ray 추적이 API에 대해 활성화됩니다.

AWS CLI 또는 AWS CloudFormation을 사용하는 경우 `xrayEnabled` 속성을 `true`로 설정하여 새 AWS AppSync API를 만들거나 기존 AWS AppSync API를 업데이트할 수도 있습니다.

AWS AppSync API에 대해 X-Ray 추적이 활성화되면 적절한 권한을 가진 계정에 AWS Identity and Access Management 서비스 연결 역할이 자동으로 생성됩니다. 이를 통해 AWS AppSync에서 X-Ray로 추적을 안전하게 보낼 수 있습니다.

X-Ray로 API 추적

샘플링

샘플링 규칙을 사용하여 AWS AppSync에 기록할 데이터의 양을 제어하고, 코드를 수정하거나 재배포하는 대신 즉석에서 샘플링 동작을 수정할 수 있습니다. 예를 들어, 이 규칙은 API ID `3n572shhcpfokwhdnq1ogu59v6`을 사용하여 GraphQL API에 대한 요청을 샘플링합니다.

- 규칙 이름 - `test-sample`
- 우선 순위 - `10`

- 리저버 크기 - 10
- 고정 비율 - 10
- 서비스 이름 - *
- 서비스 유형 - AWS::AppSync::GraphQLAPI
- HTTP 메서드 - *
- 리소스 ARN - arn:aws:appsync:us-west-2:123456789012:apis/3n572shhcpfokwhdnq1ogu59v6
- 호스트 - *

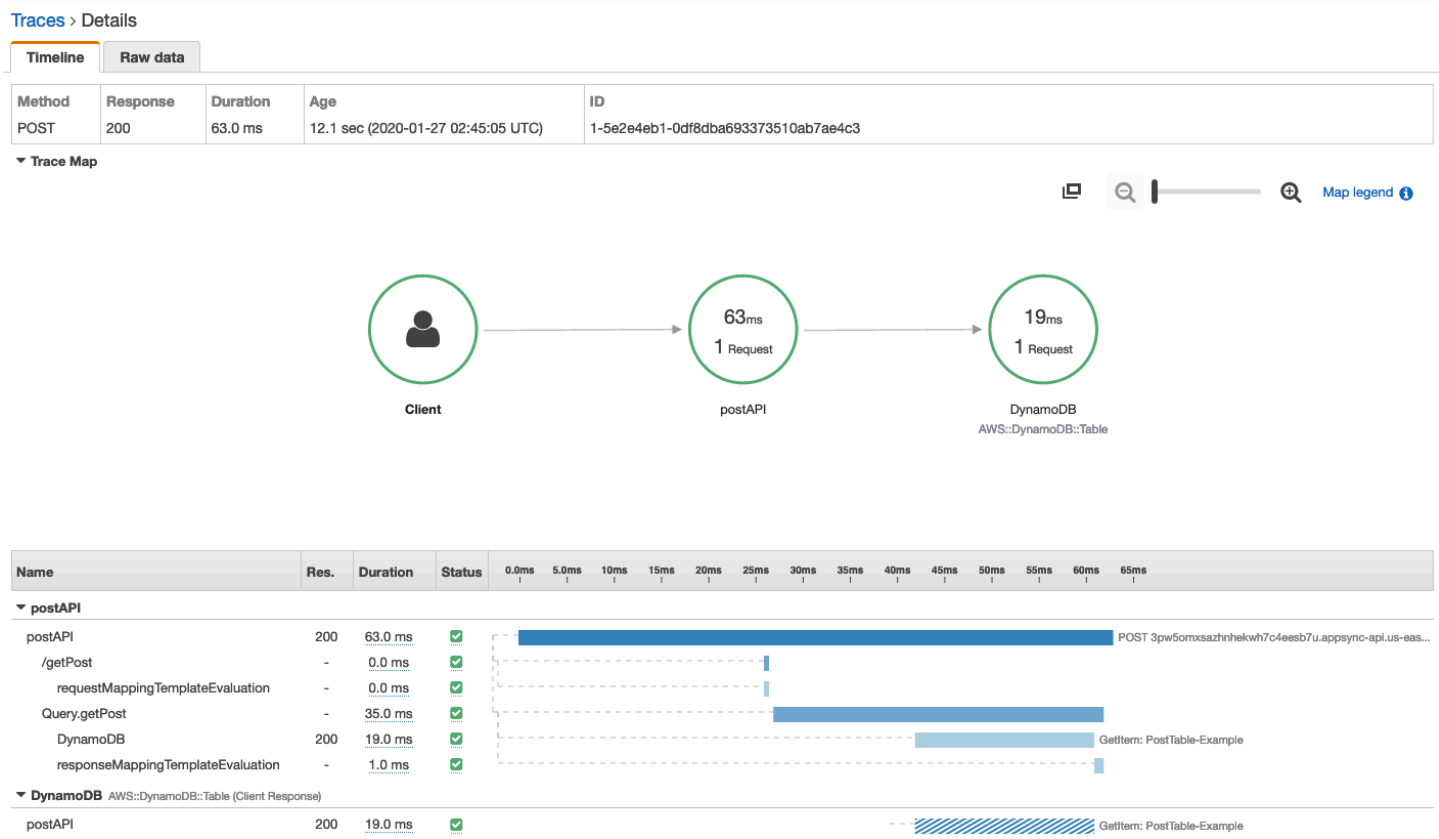
추적 이해

GraphQL API에 대한 X-Ray 추적을 활성화하면 X-Ray 추적 세부 정보 페이지를 사용하여 API 요청에 대한 자세한 지연 시간 정보를 확인할 수 있습니다. 다음 예제에서는 이 특정 요청에 대한 서비스 맵과 함께 추적 보기를 보여 줍니다. 이 요청은 Post 유형으로 postAPI라는 API에 대해 수행되었으며, 이 API의 데이터는 PostTable-Example이라는 Amazon DynamoDB 테이블에 포함되어 있습니다.

다음 추적 이미지는 다음 GraphQL 쿼리에 해당합니다.

```
query getPost {
    getPost(id: "1") {
      id
      title
    }
}
```

getPost 쿼리에 대한 해석기는 기본 DynamoDB 데이터 소스를 사용합니다. 다음 추적 보기는 DynamoDB에 대한 호출과 쿼리 실행의 여러 부분에 대한 지연 시간을 보여 줍니다.



- 앞의 이미지에서 `/getPost`는 해석 중인 요소의 전체 경로를 나타냅니다. 이 경우 `getPost`는 루트 `Query` 유형의 필드이므로 경로에서 루트 바로 뒤에 나타납니다.
- `requestMappingTemplateEvaluation`은 AWS AppSync가 쿼리에서 이 요소에 대한 요청 매핑 템플릿을 평가하는 데 소요된 시간을 나타냅니다.
- `Query.getPost`는 유형 및 필드(`Type.field` 형식)를 나타냅니다. 여기에는 API 구조와 추적 중인 요청에 따라 여러 하위 세그먼트가 포함될 수 있습니다.
  - DynamoDB는 이 해석기에 연결된 데이터 원본을 나타냅니다. 여기에는 필드를 확인하기 위해 소요된 DynamoDB에 대한 네트워크 호출의 지연 시간이 포함되어 있습니다.
  - `responseMappingTemplateEvaluation`은 AWS AppSync가 쿼리에서 이 요소에 대한 응답 매핑 템플릿을 평가하는 데 소요된 시간을 나타냅니다.

X-Ray에서 추적을 볼 때 하위 세그먼트를 선택하고 세부 정보 보기를 탐색하여 AWS AppSync 세그먼트의 하위 세그먼트에 대한 추가 컨텍스트 및 메타데이터 정보를 얻을 수 있습니다.

깊이 중첩되거나 복잡한 쿼리의 경우, AWS AppSync가 X-Ray에 전달하는 세그먼트가 세그먼트 설명서에서 허용하는 최대 크기보다 클 수 있습니다. 허용되는 최대 크기는 AWS X-Ray 세그먼트 설명서에 정의되어 있습니다. X-Ray는 제한을 초과하는 세그먼트를 표시하지 않습니다.

AWS CloudTrail을 사용하여 AWS AppSync API 호출 로깅

AWS AppSync는 AWS AppSync에서 사용자, 역할 또는 AWS 서비스가 수행한 작업에 대한 레코드를 제공하는 서비스인 AWS CloudTrail과 통합됩니다. CloudTrail은 AWS AppSync에 대한 모든 API 호출을 이벤트로 캡처합니다. 캡처되는 호출에는 AWS AppSync 콘솔의 호출과 AWS AppSync API에 대한 코드의 호출이 포함됩니다. CloudTrail에서 수집한 정보를 사용하여 AWS AppSync에 전송된 요청, 요청자의 IP 주소, 요청을 전송한 사용자, 요청이 전송된 시간 및 추가 세부 정보를 확인할 수 있습니다.

추적을 생성하여 AWS AppSync에 대한 이벤트를 포함한 CloudTrail 이벤트를 Amazon Simple Storage Service(S3) 버킷에 지속적으로 전송할 수 있습니다. 추적을 구성하지 않은 경우에도 CloudTrail 콘솔에서 최신 이벤트를 볼 수 있습니다.

> **Important**
>
> 현재 일부 GraphQL 작업은 로깅되지 않습니다. AppSync는 쿼리 및 변형 작업을 CloudTrail에 로깅하지 않습니다.

CloudTrail에 대한 자세한 내용은 AWS CloudTrail 사용 설명서를 참조하세요.

CloudTrail의 AWS AppSync 정보

CloudTrail은 계정 생성 시 AWS 계정에서 사용되도록 설정됩니다. 이벤트 기록의 CloudTrail 콘솔에서 AWS 계정의 최신 이벤트를 확인, 검색 및 다운로드할 수 있습니다. 자세한 내용은 AWS CloudTrail 사용 설명서에서 CloudTrail 이벤트 기록을 사용하여 이벤트 보기를 참조하세요.

AWS AppSync에 대한 이벤트를 포함하여 AWS 계정에 이벤트를 지속적으로 기록하려면 추적을 생성합니다. 콘솔에서 추적을 생성하면 기본적으로 모든 AWS 리전에 추적이 적용됩니다. 추적은 AWS 파티션에 있는 모든 리전의 이벤트를 로깅하고 지정된 Amazon S3 버킷으로 로그 파일을 전송합니다. 또는 CloudTrail 로그에서 수집된 이벤트 데이터를 추가 분석 및 처리하도록 다른 AWS 서비스를 구성할 수 있습니다. 자세한 내용은 AWS CloudTrail User Guide의 다음 섹션을 참조하세요.

- AWS 계정에 대한 추적 생성
- CloudTrail 로그와 AWS 서비스 통합
- CloudTrail에 대한 Amazon SNS 알림 구성
- 여러 리전에서 CloudTrail 로그 파일 수신
- 여러 계정에서 CloudTrail 로그 파일 수신

CloudTrail은 모든 AWS AppSync API 작업을 로깅합니다. 예를 들어 `CreateGraphqlApi`, `CreateDataSource` 및 `ListResolvers` API에 대한 호출은 CloudTrail 로그 파일에 항목을 생성합니다. 이러한 작업 및 기타 작업은 AWS AppSync API 참조에 문서화되어 있습니다.

모든 이벤트 및 로그 항목에는 요청을 생성한 사용자에 대한 정보가 들어 있습니다. 자격 증명 정보를 이용하면 다음을 판단하는 데 도움이 됩니다.

- 요청을 루트로 했는지 아니면 AWS Identity and Access Management(IAM) 사용자 자격 증명으로 했는지.
- 역할 또는 페더레이션 사용자에 대한 임시 보안 자격 증명을 사용하여 요청이 생성되었는지 여부.
- 다른 AWS 서비스에서 요청했는지 여부.

자세한 내용은 AWS CloudTrail User Guide의 CloudTrail userIdentity Element를 참조하세요.

AWS AppSync 로그 파일 항목 이해

CloudTrail은 하나 이상의 로그 항목이 포함된 로그 파일로 이벤트를 전송합니다. 이벤트는 모든 소스로부터의 단일 요청을 나타내며 요청 작업, 작업 날짜와 시간, 요청 파라미터 등에 대한 정보가 들어 있습니다. 로그 파일은 퍼블릭 API 호출의 정렬된 스택 기록이 아니므로 특정 순서로 나타나지 않습니다.

다음 예제의 CloudTrail 로그 항목은 `CreateApiKey` 작업을 보여줍니다.

```
{
  "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"
    }
  ]
}
```

다음 예제의 CloudTrail 로그 항목은 `ListApiKeys` 작업을 보여줍니다.

```
{
  "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"
    }
  ]
}
```

다음 예제의 CloudTrail 로그 항목은 DeleteApiKey 작업을 보여줍니다.

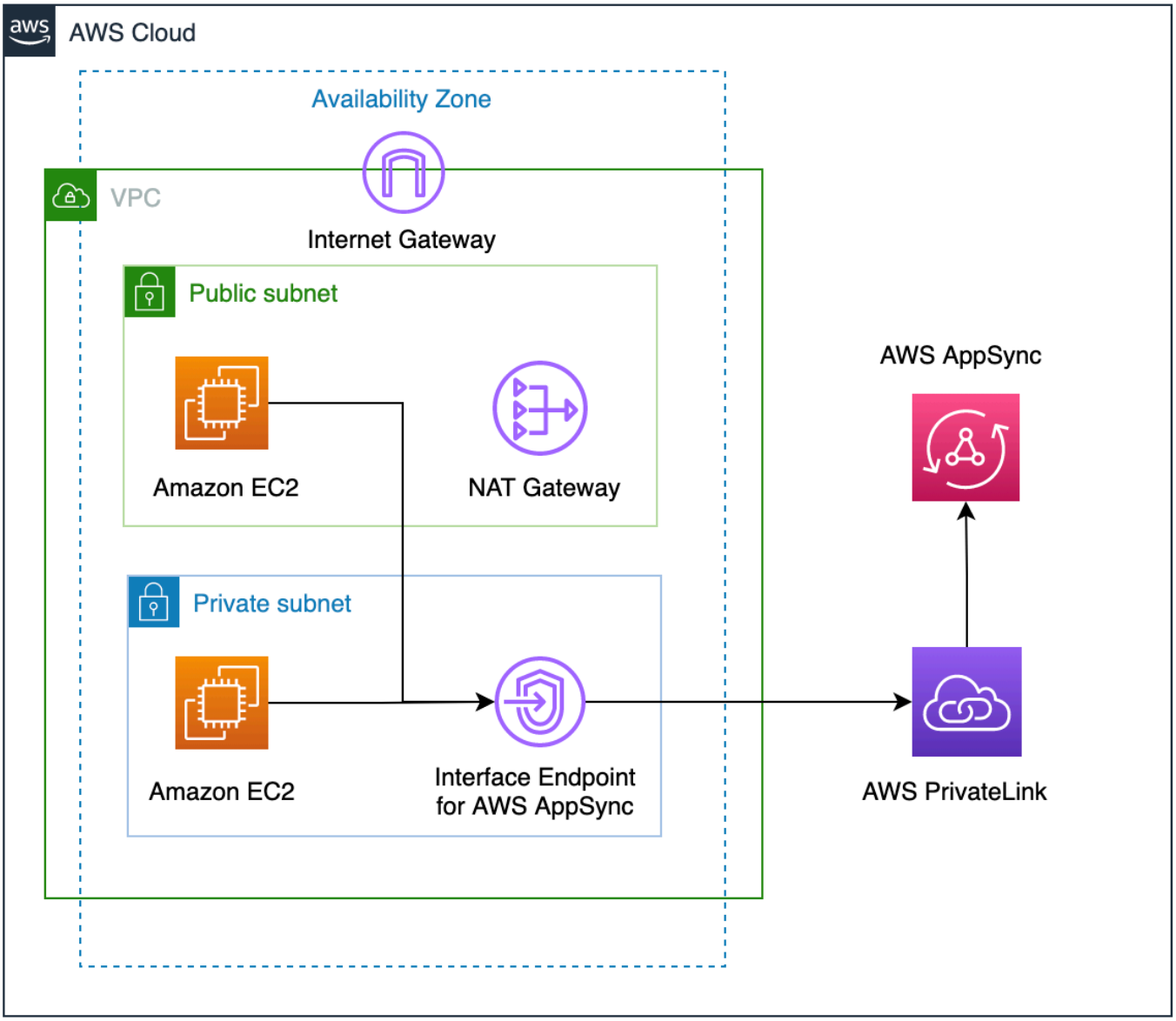
```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/Alice",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "Alice"
    },
    "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"
    }
  ]
}
```

AWS AppSync 프라이빗 API 사용

Amazon Virtual Private Cloud(VPC)를 사용하면 VPC에서만 액세스할 수 있는 AWS AppSync 프라이빗 API를 생성할 수 있습니다. 프라이빗 API를 사용하면 데이터를 공개적으로 노출하지 않으면서 내부 애플리케이션에 대한 API 액세스를 제한하고 GraphQL 및 Realtime 엔드포인트에 연결할 수 있습니다.

VPC와 AWS AppSync 서비스 간에 프라이빗 연결을 설정하려면 인터페이스 VPC 엔드포인트를 생성해야 합니다. 인터페이스 엔드포인트는 인터넷 게이트웨이, NAT 디바이스, VPN 연결 또는 AWS Direct Connect 연결 없이 비공개로 AWS AppSync API에 액세스할 수 있도록 지원하는 AWS PrivateLink에 의해 구동됩니다. VPC의 인스턴스는 AWS AppSync API와 통신하는 데 퍼블릭 IP 주소를 필요로 하지 않습니다. VPC와 AWS AppSync 간의 트래픽은 AWS 네트워크를 벗어나지 않습니다.



프라이빗 API 기능을 활성화하기 전에 몇 가지 사항을 추가로 고려해야 합니다.

- 프라이빗 DNS 기능이 활성화된 상태로 AWS AppSync에 VPC 인터페이스 엔드포인트를 설정하면 VPC의 리소스가 AWS AppSync에서 생성한 API URL을 사용하여 다른 AWS AppSync 퍼블릭 API를 간접적으로 호출할 수 없게 됩니다. 이는 퍼블릭 API에 대한 요청이 인터페이스 엔드포인트를 통해 라우팅되는데, 퍼블릭 API에 허용되지 않는 방식이기 때문입니다. 이 시나리오에서 퍼블릭 API를 간접적으로 호출하려면 퍼블릭 API에서 사용자 지정 도메인 이름을 구성하는 것이 좋습니다. 그렇게 하면 퍼블릭 API를 간접적으로 호출하는 데 VPC의 리소스가 사용될 수 있습니다.

- AWS AppSync 프라이빗 API는 VPC에서만 사용할 수 있습니다. AWS AppSync 콘솔 쿼리 편집기는 브라우저의 네트워크 구성이 트래픽을 VPC로 라우팅할 수 있는 경우(예: VPN 또는 AWS Direct Connect를 통한 연결)에만 API에 도달할 수 있습니다.
- AWS AppSync의 VPC 인터페이스 엔드포인트를 사용하면 동일한 AWS 계정 및 리전의 모든 프라이빗 API에 액세스할 수 있습니다. 프라이빗 API에 대한 액세스를 추가로 제한하려면 다음 옵션을 고려해 보세요.
  - 필요한 관리자만 AWS AppSync에 대한 VPC 엔드포인트 인터페이스를 생성할 수 있도록 합니다.
  - VPC 엔드포인트 사용자 지정 정책을 사용하여 VPC의 리소스에서 간접적으로 호출할 수 있는 API를 제한합니다.
  - VPC에 있는 리소스의 경우, 리소스에 API에 대한 축소된 범위의 역할을 부여함으로써 IAM 권한 부여를 사용해 AWS AppSync API를 간접적으로 호출하는 것이 좋습니다.
- IAM 보안 주체를 제한하는 정책을 만들거나 사용할 때는 메서드의 `authorizationType`을 `AWS_IAM` 또는 `NONE`으로 설정해야 합니다.

AWS AppSync 프라이빗 API 생성

다음 단계는 AWS AppSync 서비스에서 프라이빗 API를 만드는 방법을 보여 줍니다.

> **Warning**
>
> API를 생성하는 동안에만 프라이빗 API 기능을 활성화할 수 있습니다. 생성한 이후에는 AWS AppSync API 또는 AWS AppSync 프라이빗 API에서 이 설정을 수정할 수 없습니다.

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
   - 대시보드에서 API 생성을 선택합니다.
2. 처음부터 API 설계를 선택하고 다음을 선택합니다.
3. 프라이빗 API 섹션에서 프라이빗 API 기능 사용을 선택합니다.
4. 나머지 옵션을 구성하고 API의 데이터를 검토한 다음 생성을 선택합니다.

AWS AppSync 프라이빗 API를 사용하려면 먼저 VPC의 AWS AppSync에 인터페이스 엔드포인트를 구성해야 합니다. 단, 프라이빗 API와 VPC가 동일한 AWS 계정 및 리전에 있어야 합니다.

AWS AppSync에 대한 인터페이스 엔드포인트 생성

Amazon VPC 콘솔 또는 AWS Command Line Interface(AWS CLI)를 사용하여 AWS AppSync에 대한 인터페이스 엔드포인트를 생성할 수 있습니다. 자세한 내용은 Amazon VPC 사용 설명서의 인터페이스 엔드포인트 생성을 참조하세요.

Console

1. AWS Management Console에 로그인한 다음 Amazon VPC 콘솔의 엔드포인트 페이지를 엽니다.
2. 엔드포인트 생성을 선택합니다.
   a. 서비스 카테고리 필드에서 AWS 서비스가 선택되어 있는지 확인합니다.
   b. 서비스 테이블에서 `com.amazonaws.{region}.appsync-api`를 선택합니다. 유형 열 값이 `Interface`인지 확인합니다.
   c. VPC 필드에서 VPC와 서브넷을 선택합니다.
   d. 인터페이스 엔드포인트에 대한 프라이빗 DNS 기능을 활성화하려면 DNS 이름 활성화 확인란을 선택합니다.
   e. 보안 그룹 필드에서 보안 그룹을 하나 이상 선택합니다.
3. 엔드포인트 생성을 선택합니다.

CLI

`create-vpc-endpoint` 명령을 사용하여 엔드포인트 네트워크 인터페이스와 연결할 VPC ID, VPC 엔드포인트 유형(인터페이스), 서비스 이름, 엔드포인트를 사용할 서브넷 및 보안 그룹을 지정합니다. 예:

```
$ 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
```

프라이빗 DNS 옵션을 사용하려면 VPC의 `enableDnsHostnames` 및 `enableDnsSupportattributes` 값을 설정해야 합니다. 자세한 내용은 Amazon VPC 사용 설명서의 VPC에 대한 DNS 지원 보기 및 업데이트를 참조하세요. 인터페이스 엔드포인트에 대해 프라이빗

DNS 기능을 활성화하면 아래 형식으로 기본 퍼블릭 DNS 엔드포인트를 사용하여 AWS AppSync API GraphQL 및 Real-Time 엔드포인트에 요청을 보낼 수 있습니다.

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

서비스 엔드포인트에 대한 자세한 내용은 AWS 일반 참조에서 서비스 엔드포인트 및 할당량을 참조하세요.

인터페이스 엔드포인트와의 서비스 상호 작용에 대한 자세한 내용은 Amazon VPC 사용 설명서의 인터페이스 엔드포인트를 통해 서비스 액세스를 참조하세요.

AWS CloudFormation을 사용하여 엔드포인트를 생성하고 구성하는 방법에 대한 자세한 내용은 AWS CloudFormation 사용 설명서의 AWS::EC2::VPCEndpoint 리소스를 참조하세요.

고급 예제

인터페이스 엔드포인트에 대해 프라이빗 DNS 기능을 활성화하면 아래 형식으로 기본 퍼블릭 DNS 엔드포인트를 사용하여 AWS AppSync API GraphQL 및 Real-Time 엔드포인트에 요청을 보낼 수 있습니다.

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

인터페이스 VPC 엔드포인트 퍼블릭 DNS 호스트 이름을 사용할 경우 API를 간접적으로 호출하는 기본 URL의 형식은 다음과 같습니다.

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.
{region}.vpce.amazonaws.com/graphql
```

AZ에 엔드포인트를 배포한 경우 AZ 전용 DNS 호스트 이름을 사용할 수도 있습니다.

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}-{az_id}.appsync-api.
{region}.vpce.amazonaws.com/graphql.
```

VPC 엔드포인트 퍼블릭 DNS 이름을 사용하려면 AWS AppSync API 엔드포인트 호스트 이름을 `Host` 또는 `x-appsync-domain` 헤더로 요청에 전달해야 합니다. 다음 예제에서는 샘플 스키마 시작 가이드에서 생성했던 `TodoAPI`를 사용합니다.

```
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"}}}'
```

다음 예제에서는 샘플 스키마 시작 가이드에서 생성한 Todo 앱을 사용합니다. 샘플 Todo API를 테스트하기 위해 프라이빗 DNS를 사용하여 API를 간접적으로 호출하겠습니다. 원하는 명령줄 도구를 사용할 수 있으며, 이 예제에서는 curl을 사용하여 쿼리와 변형을 전송하고 wscat를 사용하여 구독을 설정합니다. 이 예제를 에뮬레이션하려면 아래 명령의 대괄호 { } 안의 값을 AWS 계정의 해당 값으로 대체하세요.

변형 작업 테스트 - **`createTodo`** 요청

```
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"}}}'
```

변형 작업 테스트 - **`createTodo`** 응답

```
{
    "data": {
        "createTodo": {
            "id": "<todo-id>",
            "name": "My first GraphQL task",
            "where": "Day 1",
            "when": "Friday Night",
            "description": "Learn more about GraphQL"
        }
    }
}
```

쿼리 작업 테스트 - **`listTodos`** 요청

```
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"}}}'
```

쿼리 작업 테스트 - **`listTodos`** 요청

```
{
  "data": {
    "listTodos": {
      "items": [
        {
          "description": "Learn more about GraphQL",
          "id": "<todo-id>",
          "name": "My first GraphQL task",
          "when": "Friday night",
          "where": "Day 1"
        }
      ]
    }
  }
}
```

구독 작업 테스트 - **`createTodo`** 변형 구독

AWS AppSync에서 GraphQL 구독을 설정하려면 실시간 WebSocket 클라이언트 빌드를 참조하세요. VPC의 Amazon EC2 인스턴스에서 wscat를 사용하여 AWS AppSync 프라이빗 API 구독 엔드포인트를 테스트할 수 있습니다. 아래 예에서는 권한 부여를 위해 `API KEY`를 사용합니다.

```
$ 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"}
```

또는 VPC 엔드포인트 도메인 이름을 사용하고 wscat 명령에서 호스트 헤더를 지정하여 WebSocket을 설정해야 합니다.

```
$ 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"}
```

아래 변형 코드를 실행하세요.

```
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"}}}'
```

이후 구독이 시작되고 아래와 같이 메시지 알림이 표시됩니다.

```
< {"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"}}}}
```

IAM 정책을 사용하여 퍼블릭 API 생성 제한

AWS AppSync에서는 프라이빗 API와 함께 사용할 수 있는 IAM `Condition` 설명을 지원합니다. 프라이빗 및 퍼블릭 API를 생성할 수 있는 IAM 역할 및 사용자를 제어하는 `appsync:CreateGraphqlApi` 작업을 위해 IAM 정책 설명과 함께 `visibility` 필드를 포함할 수 있습니다. 이렇게 하면 IAM 관리자는 사용자가 프라이빗 GraphQL API만 생성할 수 있도록 하는 IAM 정책을 정의할 수 있습니다. 퍼블릭 API를 생성하려는 사용자에게는 권한 없음 메시지가 표시됩니다.

예를 들어 IAM 관리자는 다음과 같은 IAM 정책 설명을 생성하여 프라이빗 API 생성을 허용할 수 있습니다.

```
{
    "Sid": "AllowPrivateAppSyncApis",
    "Effect": "Allow",
    "Action": "appsync:CreateGraphqlApi",
    "Resource": "*",
    "Condition": {
        "ForAnyValue:StringEquals": {
            "appsync:Visibility": "PRIVATE"
        }
    }
}
```

또한 IAM 관리자는 다음과 같은 서비스 제어 정책을 추가하여 AWS 조직의 모든 사용자가 프라이빗 API 이외의 AWS AppSync API를 생성하지 못하도록 할 수 있습니다.

```
{
    "Sid": "BlockNonPrivateAppSyncApis",
    "Effect": "Deny",
    "Action": "appsync:CreateGraphqlApi",
    "Resource": "*",
    "Condition": {
        "ForAnyValue:StringNotEquals": {
            "appsync:Visibility": "PRIVATE"
        }
    }
}
```

AWS AppSync를 사용하여 GraphQL 실행 복잡성, 쿼리 깊이 및 내부 검사 구성

AWS AppSync를 사용하면 단일 쿼리로 내부 검사 기능을 활성화 또는 비활성화하고 중첩 수준 및 해석기 수에 제한을 설정할 수 있습니다.

내부 검사 기능 사용

> **Tip**
>
> GraphQL의 내부 검사에 대한 자세한 내용은 GraphQL 재단 웹 사이트의 이 문서를 참조하십시오.

기본적으로 GraphQL에서는 내부 검사를 사용하여 스키마 자체를 쿼리하여 스키마의 유형, 필드, 쿼리, 변형, 구독 등을 검색할 수 있습니다. 이는 GraphQL 서비스가 데이터를 구성하고 처리하는 방법을 학습하는 데 중요한 기능입니다. 그러나 내부 검사 시에는 다음과 같은 고려해야 할 사항이 있습니다. 내부 검사를 비활성화하면 도움이 되는 사용 사례가 있을 수 있습니다. 예를 들어 필드 이름이 민감하거나 숨겨질 수 있는 경우 또는 전체 API 스키마를 소비자에게 문서화하지 않도록 설정하는 경우가 이에 해당합니다. 이러한 경우 내부 검사를 통해 스키마 데이터를 게시하면 프라이빗 데이터가 의도적으로 유출될 수 있습니다.

이러한 일이 발생하지 않도록 내부 검사를 비활성화할 수 있습니다. 이렇게 하면 권한이 없는 당사자가 스키마의 내부 검사 필드를 사용하는 것을 방지할 수 있습니다. 하지만 개발팀이 서비스의 데이터가 처리되는 방식을 익히는 데 내부 검사가 유용하다는 점이 중요합니다. 내부적으로는 내부 검사를 활성화한 상태로 유지하면서 프로덕션 코드에서는 추가 보안 계층으로 사용하지 않도록 설정하는 것이 유용할 수 있습니다. 이를 처리하는 또 다른 방법은 권한 부여 메서드를 추가하는 것입니다. AWS AppSync 역시 이 메서드를 제공합니다. 자세한 내용은 권한 부여 섹션을 참조하십시오.

AWS AppSync를 통해 API 수준에서 내부 검사를 활성화하거나 비활성화할 수 있습니다. 내부 검사를 활성화 또는 비활성화하려면 다음을 따르십시오.

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
2. API 페이지에서 GraphQL API의 이름을 선택합니다.
3. API 홈페이지의 탐색 창에서 설정 을 선택합니다.
4. API 구성에서 편집을 선택합니다.
5. 내부 검사 쿼리에서 다음 작업을 수행합니다.

- 내부 검사 쿼리 활성화를 켜거나 끕니다.

6. 저장을 선택합니다.

내부 검사를 활성화하면(기본 동작) 내부 검사 시스템이 정상적으로 작동합니다. 예를 들어 아래 이미지는 스키마에서 사용 가능한 모든 유형을 처리하는 __schema 필드를 보여줍니다.

이 기능을 비활성화하면 응답에 검증 오류가 대신 표시됩니다.

쿼리 깊이 제한 구성

작업 중에 API가 작동하는 방식을 더 세밀하게 제어해야 하는 경우가 있을 수 있습니다. 이러한 제어 중 하나는 쿼리가 처리할 수 있는 중첩 수준의 양에 제한을 추가하는 것입니다. 기본적으로 쿼리는 중

첩 수준을 무제한으로 처리할 수 있습니다. 쿼리를 지정된 양의 중첩 수준으로 제한하면 프로젝트의 성능과 유연성에 영향을 미칠 수 있습니다. 다음 쿼리를 실행합니다.

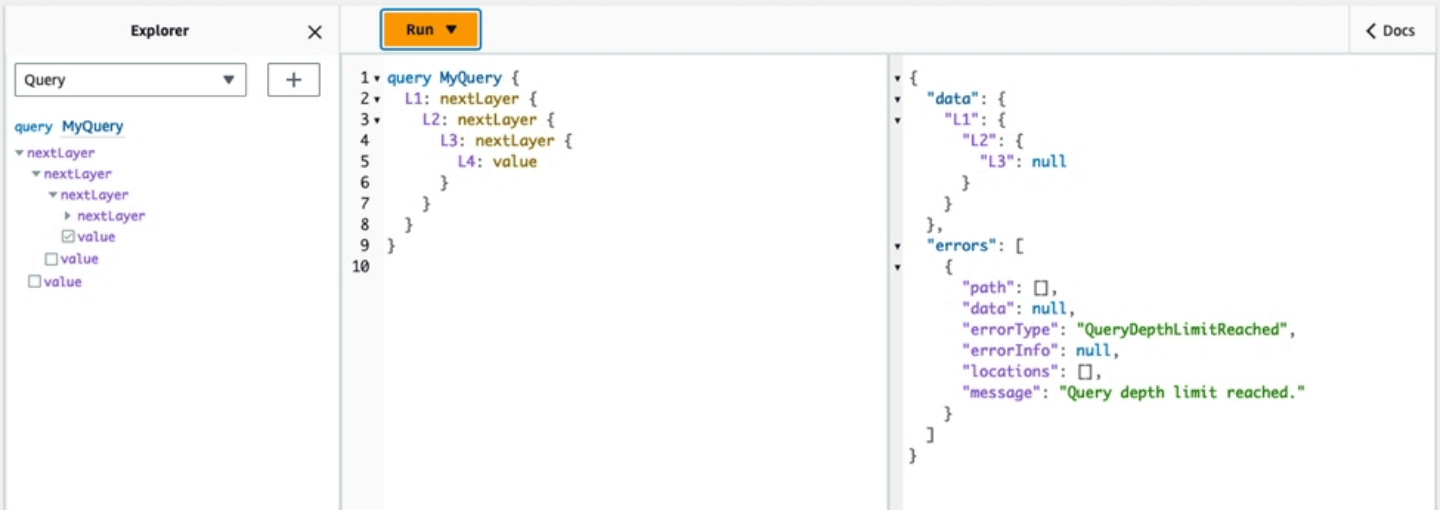
```
query MyQuery {
  L1: nextLayer {
    L2: nextLayer {
      L3: nextLayer {
        L4: value
      }
    }
  }
}
```

프로젝트에서 L1 또는 L2에 대한 쿼리를 특정 목적으로 제한하도록 요청할 수도 있습니다. 기본적으로 L1부터 L4까지의 전체 쿼리는 이를 제어할 수 있는 방법 없이 처리됩니다. 제한을 설정하면 쿼리가 지정된 수준 이상의 항목에 액세스하는 것을 방지할 수 있습니다.

쿼리 깊이 제한을 추가하려면 다음 작업을 수행합니다.

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
2. API 페이지에서 GraphQL API의 이름을 선택합니다.
3. API 홈페이지의 탐색 창에서 설정 을 선택합니다.
4. API 구성에서 편집을 선택합니다.
5. 쿼리 깊이에서 다음 작업을 수행합니다.
   a. 쿼리 깊이 활성화를 켜거나 끕니다.
   b. 최대 깊이에서 깊이 제한을 설정합니다. 이 값은 1부터 75 사이일 수 있습니다.
6. 저장을 선택합니다.

제한이 설정된 경우 상한을 넘으면 QueryDepthLimitReached 오류가 발생합니다. 예를 들어 아래 이미지는 2의 깊이 제한이 적용된 쿼리가 세 번째(L3) 및 네 번째(L4) 수준의 제한을 초과하는 모습을 보여줍니다.



참고로 스키마에서 필드를 여전히 Null 허용 또는 Null 허용 불가로 표시할 수 있습니다. Null 허용 불가 필드에 `QueryDepthLimitReached` 오류가 발생하면 해당 오류는 Null 허용 첫 번째 상위 필드에 발생합니다.

해석기 수 제한 구성

각 쿼리가 처리할 수 있는 해석기 수도 제어할 수 있습니다. 쿼리 깊이와 마찬가지로 이 양에도 제한을 설정할 수 있습니다. 세 개의 해석기가 포함된 다음 쿼리를 예로 들어 보겠습니다.

```
query MyQuery {
  resolver1: resolver
  resolver2: resolver
  resolver3: resolver
}
```

기본적으로 각 쿼리는 최대 10,000개의 해석기를 처리할 수 있습니다. 위 예시에서는 `resolver1`, `resolver2`, `resolver3`이 처리됩니다. 하지만 프로젝트에서는 각 쿼리를 총 하나 또는 두 개의 해석기를 처리하도록 제한을 요청할 수도 있습니다. 제한을 설정하면 첫 번째(`resolver1`) 또는 두 번째(`resolver2`) 해석기처럼 특정 수를 초과하는 해석기를 처리하지 않도록 쿼리에 지시할 수 있습니다.

해석기 수 제한을 추가하려면 다음 작업을 수행합니다.

1. AWS Management Console에 로그인한 다음 AppSync 콘솔을 엽니다.
2. API 페이지에서 GraphQL API의 이름을 선택합니다.
3. API 홈페이지의 탐색 창에서 설정 을 선택합니다.
4. API 구성에서 편집을 선택합니다.

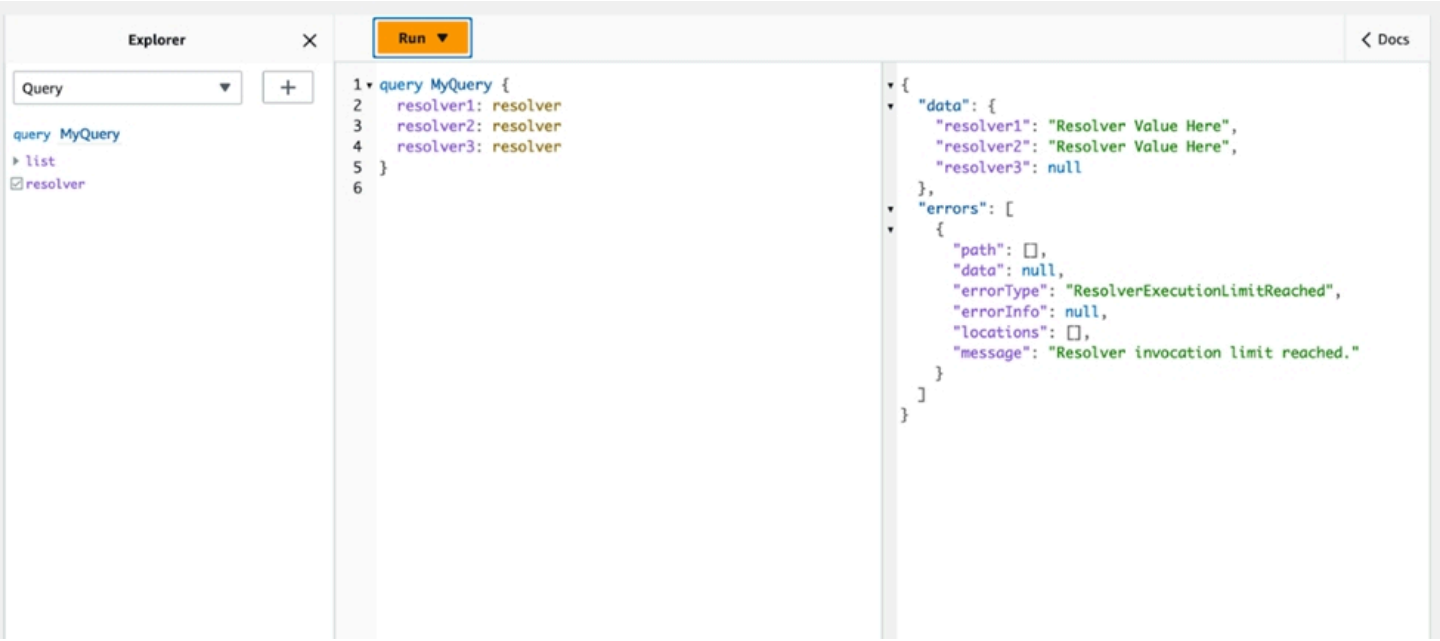
5. 해석기 수 제한에서 다음 작업을 수행합니다.

   a. 해석기 수 활성화를 켭니다.

   b. 최대 해석기 수에서 개수 제한을 설정합니다. 이 값은 1부터 10000 사이일 수 있습니다.

6. 저장을 선택합니다.

쿼리 깊이 제한과 마찬가지로 구성된 해석기 제한을 초과하면 추가 해석기에서 `ResolverExecutionLimitReached` 오류가 발생하여 쿼리가 종료됩니다. 아래 이미지에서 해석기 수 제한이 2인 쿼리가 세 개의 해석기를 처리하려고 합니다. 제한 때문에 세 번째 해석기에서 오류가 발생하고 실행되지 않습니다.



에서 환경 변수 사용 AWS AppSync

코드를 업데이트하지 않고도 환경 변수를 사용하여 AWS AppSync 리졸버 및 함수의 동작을 조정할 수 있습니다. 환경 변수는 API 구성과 함께 저장된 문자열 쌍으로, 해석기와 함수가 런타임에 활용할 수 있도록 제공됩니다. 이는 초기 설정 중에만 사용할 수 있지만 실행 중에는 해석기와 함수에서 사용해야 하는 구성 데이터를 참조해야 하는 상황에 특히 유용합니다. 환경 변수를 사용하면 코드에 구성 데이터가 노출되므로 이러한 값을 하드 코딩할 필요가 줄어듭니다.

> **Note**
>
> 데이터베이스 보안을 강화하려면 환경 변수 대신 Secrets Manager 또는 AWS Systems Manager 매개변수 저장소를 사용하여 자격 증명이나 민감한 정보를 저장하는 것이 좋습니다.

이 기능을 활용하려면 AWS AppSync HTTP 데이터 소스를 사용한 AWS 서비스 호출을 참조하십시오.

환경 변수가 제대로 작동하려면 몇 가지 동작과 규칙을 따라야 합니다.

- JavaScript 리졸버/함수와 VTL 템플릿 모두 환경 변수를 지원합니다.
- 함수 호출 전에는 환경 변수가 평가되지 않습니다.
- 환경 변수는 문자열 값만 지원합니다.
- 환경 변수에 정의된 모든 값은 문자열 리터럴로 간주되며 확장되지 않습니다.
- 변수 평가는 함수 코드에서 수행하는 것이 가장 좋습니다.

환경 변수 구성(콘솔)

변수를 생성하고 키-값 쌍을 정의하여 AWS AppSync GraphQL API의 환경 변수를 구성할 수 있습니다. 리졸버와 함수는 런타임에 환경 변수의 키 이름을 사용하여 값을 검색합니다. 콘솔에서 환경 변수를 설정하려면: AWS AppSync

1. 에 AWS Management Console 로그인하고 AppSync콘솔을 엽니다.
2. API 페이지에서 GraphQL API의 이름을 선택합니다.
3. API 홈페이지의 탐색 창에서 설정 을 선택합니다.
4. 환경 변수에서 환경 변수 추가를 선택합니다.
5. Add environment variable(환경 변수 추가)을 선택합니다.
6. 키와 값을 입력합니다.
7. 필요한 경우 5단계와 6단계를 반복하여 키 값을 더 추가합니다. 키 값을 제거해야 하는 경우 제거 옵션과 제거할 키를 선택합니다.
8. 제출을 선택합니다.

Tip

키와 값을 생성할 때 따라야 하는 몇 가지 규칙이 있습니다.

- 키는 문자로 시작해야 합니다.
- 키는 2자 이상이어야 합니다.

- 키에는 문자, 숫자, 밑줄 문자 (_) 만 사용할 수 있습니다.
- 값은 최대 512자까지 입력할 수 있습니다.
- GraphQL API에서 최대 50개의 키-값 쌍을 구성할 수 있습니다.

환경 변수 구성(API)

API를 사용하여 환경 변수를 설정하려면 다음을 사용할 수 있습니다. `PutGraphqlApiEnvironmentVariables` 해당 CLI 명령은 입니다. `put-graphql-api-environment-variables`

API를 사용하여 환경 변수를 검색하려면 다음을 사용할 수 있습니다. `GetGraphqlApiEnvironmentVariables` 해당 CLI 명령은 입니다. `get-graphql-api-environment-variables`

명령에는 API ID와 환경 변수 목록이 포함되어야 합니다.

```
aws appsync put-graphql-api-environment-variables \
  --api-id "<api-id>" \
  --environment-variables '{"key1":"value1","key2":"value2", …}'
```

다음 예시에서는 `put-graphql-api-environment-variables` 명령을 `abcdefghijklmnopqrstuvwxyz` 사용하는 ID로 API에 두 개의 환경 변수를 설정합니다.

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true"}'
```

`put-graphql-api-environment-variables`명령으로 환경 변수를 적용하면 환경 변수 구조의 내용이 덮어쓰여진다는 점에 유의하십시오. 즉, 기존 환경 변수는 손실됩니다. 새 환경 변수를 추가할 때 기존 환경 변수를 유지하려면 요청에 새 키-값 쌍과 함께 기존 키-값 쌍을 모두 포함하십시오. 위의 예를 사용하여 `"EMPTY":""` 추가하려는 경우 다음과 같이 할 수 있습니다.

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true", "EMPTY":""}'
```

현재 구성을 검색하려면 다음 `get-graphql-api-environment-variables` 명령을 사용합니다.

```
aws appsync get-graphql-api-environment-variables --api-id "<api-id>"
```

위 예제를 사용하여 다음 명령을 사용할 수 있습니다.

```
aws appsync get-graphql-api-environment-variables --api-id "abcdefghijklmnopqrstuvwxyz"
```

결과에는 환경 변수 목록이 해당 키 값과 함께 표시됩니다.

```
{
    "environmentVariables": {
        "USER_TABLE": "users_prod",
        "DEBUG": "true",
        "EMPTY": ""
    }
}
```

환경 변수 (CFN) 구성

아래 템플릿을 사용하여 환경 변수를 만들 수 있습니다.

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  GraphQLApiWithEnvVariables:
    Type: "AWS::AppSync::GraphQLApi"
    Properties:
      Name: "MyApiWithEnvVars"
      AuthenticationType: "AWS_IAM"
      EnvironmentVariables:
        EnvKey1: "non-empty"
        EnvKey2: ""
```

환경 변수 및 병합된 API

소스 API에 정의된 환경 변수는 병합된 API에서도 사용할 수 있습니다. 병합된 API의 환경 변수는 읽기 전용이며 업데이트할 수 없습니다. 병합이 성공하려면 환경 변수 키가 모든 소스 API에서 고유해야 합니다. 키가 중복되면 항상 병합에 실패합니다.

환경 변수 검색

함수 코드에서 환경 변수를 검색하려면 리졸버 및 함수의 `ctx.env` 객체에서 값을 검색하십시오. 다음은 이러한 동작의 몇 가지 예시입니다.

Publishing to Amazon SNS

이 예시에서는 HTTP 리졸버가 Amazon SNS 주제에 메시지를 전송합니다. 주제의 ARN은 GraphQL API를 정의하는 스택을 생성하고 주제를 배포한 후에만 알 수 있습니다.

```
/**
 * 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

이 예제에서 API가 스테이징용으로 배포되었거나 이미 프로덕션 단계에 있는 경우 DynamoDB 테이블의 이름이 다릅니다. 리졸버 코드는 변경할 필요가 없습니다. 환경 변수 값은 API가 배포된 위치에 따라 업데이트됩니다.

```
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
      },
    ],
  };
}
```

인증 및 권한 부여

이 단원에서는 애플리케이션에 대한 보안 및 데이터 보호 구성을 위한 옵션을 설명합니다.

권한 부여 유형

애플리케이션이 AWS AppSync GraphQL API와 상호 작용하도록 승인할 수 있는 다섯 가지 방법이 있습니다. AWS AppSync API 또는 CLI 호출에서 다음 권한 부여 유형 값 중 하나를 지정하여 사용할 권한 부여 유형을 지정합니다.

- **`API_KEY`**

  API 키를 사용하는 경우

- **`AWS_LAMBDA`**

  AWS Lambda 함수 사용용.

- **`AWS_IAM`**

  AWS Identity and Access Management (IAM) 권한을 사용하는 경우.

- **`OPENID_CONNECT`**

  OpenID Connect 공급자를 사용하는 경우

- **`AMAZON_COGNITO_USER_POOLS`**

  Amazon Cognito 사용자 풀을 사용하는 경우

이러한 기본 권한 부여 유형은 대부분의 개발자에게 유용합니다. 추가 고급 사용 사례를 보려면 콘솔, CLI 및 AWS CloudFormation을 통해 추가 권한 부여 모드를 추가할 수 있습니다. 추가 권한 부여 모드의 경우 위에 나열된 값 (즉,,, `API_KEY` `AWS_LAMBDA` `AWS_IAM`OPENID_CONNECT, 및AMAZON_COGNITO_USER_POOLS) 을 사용하는 권한 부여 유형을 AWS AppSync 제공합니다.

`API_KEY`, `AWS_LAMBDA` 또는 `AWS_IAM`을 주 또는 기본 권한 부여 유형으로 지정하면 추가 권한 부여 모드 중 하나로 다시 지정할 수 없습니다. 마찬가지로, 추가 권한 부여 모드 내에서는 `API_KEY`, `AWS_LAMBDA` 또는 `AWS_IAM`을 복제할 수 없습니다. 여러 개의 Amazon Cognito 사용자 풀 및 OpenID Connect 공급자를 사용할 수 있습니다. 하지만 기본 권한 부여 모드와 추가 권한 부여 모드 중 하나 간에 중복 Amazon Cognito 사용자 풀 또는 OpenID Connect 공급자를 사용할 수 없습니다. 해당 구성 정

규식을 사용하여 Amazon Cognito 사용자 풀 또는 OpenID Connect 공급자에 대해 다른 클라이언트를 지정할 수 있습니다.

API_KEY 권한 부여

인증되지 않은 API에는 인증된 API보다 더 엄격한 조절이 필요합니다. 인증되지 않은 GraphQL 엔드포인트에 대한 조절을 제어하는 한 가지 방법은 API 키를 사용하는 것입니다. API 키는 인증되지 않은 GraphQL 엔드포인트를 생성할 때 AWS AppSync 서비스에서 생성되는 애플리케이션의 하드 코딩된 값입니다. 콘솔, CLI 또는 AWS AppSync API 참조에서 API 키를 교체할 수 있습니다.

Console

1. 에 로그인하고 콘솔을 엽니다. AWS Management Console AppSync
   a. API 대시보드에서 GraphQL API를 선택합니다.
   b. 사이드바에서 설정을 선택합니다.
2. 기본 권한 부여 모드에서 API 키를 선택합니다.
3. API 키 테이블에서 API 키 추가를 선택합니다.

   테이블에 새 API 키가 생성됩니다.

   - 이전 API 키를 삭제하려면 테이블에서 API 키를 선택한 다음 삭제를 선택합니다.
4. 페이지 하단에서 [Save(저장하기)]를 선택합니다.

CLI

1. 아직 구성하지 않았다면 AWS CLI에 대한 액세스를 구성하십시오. 자세한 정보는 구성 기본 사항을 참조하세요.
2. `update-graphql-api` 명령을 실행하여 GraphQL API 객체를 생성합니다.

   이 특정 명령에 대해 두 개의 파라미터를 입력해야 합니다.

   1. GraphQL API의 `api-id`.
   2. API의 새 `name`. 동일한 `name`을 사용할 수 있습니다.
   3. `API_KEY`로 사용될 `authentication-type`.

> **Note**
>
> 필수로 구성해야 하지만 일반적으로 CLI 구성 값으로 기본 설정되는 Region과 같은 다른 파라미터도 있습니다.

예를 들어 명령은 다음과 같을 수 있습니다.

```
aws appsync update-graphql-api --api-id abcdefghijklmnopqrstuvwxyz --name
 TestAPI --authentication-type API_KEY
```

CLI에서 출력이 반환됩니다. 다음은 JSON의 예입니다.

```
{
    "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"
    }
}
```

API 키는 최대 365일 동안 구성할 수 있으며 기존 만료 날짜를 해당일부터 최장 365일 더 연장할 수 있습니다. API 키는 개발용 또는 퍼블릭 API를 노출해도 안전한 사용 사례에 권장됩니다.

클라이언트에서 API 키는 헤더 `x-api-key`가 지정합니다.

예를 들어, `API_KEY`가 `'ABC123'`인 경우 다음과 같이 `curl` 을 통해 GraphQL 쿼리를 보낼 수 있습니다.

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "x-api-key:ABC123" -d
 '{ "query": "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

AWS_LAMBDA 권한 부여

AWS Lambda 함수를 사용하여 자체 API 권한 부여 로직을 구현할 수 있습니다. Lambda 함수를 기본 또는 보조 권한 부여자로 사용할 수 있지만, Lambda 권한 부여 함수는 API당 하나만 있을 수 있습니다. 권한 부여를 위해 Lambda 함수를 사용하는 경우 다음이 적용됩니다.

- API에 `AWS_LAMBDA` 및 `AWS_IAM` 권한 부여 모드가 활성화되어 있는 경우 SigV4 서명을 `AWS_LAMBDA` 권한 부여 토큰으로 사용할 수 없습니다.
- API에 `AWS_LAMBDA` 및 `OPENID_CONNECT` 권한 부여 모드 또는 `AMAZON_COGNITO_USER_POOLS` 권한 부여 모드가 활성화되어 있는 경우 OIDC 토큰을 `AWS_LAMBDA` 인증 토큰으로 사용할 수 없습니다. 참고로 OIDC 토큰은 보유자 체계일 수 있습니다.
- Lambda 함수는 해석기에 대해 5MB를 초과하는 컨텍스트 데이터를 반환해서는 안 됩니다.

예를 들어 권한 부여 토큰이 `'ABC123'`인 경우 다음과 같이 curl을 통해 GraphQL 쿼리를 보낼 수 있습니다.

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "Authorization:ABC123" -d
 '{ "query":
        "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

Lambda 함수는 각 쿼리 또는 변형 전에 호출됩니다. API ID 및 인증 토큰을 기반으로 반환 값을 캐시할 수 있습니다. 기본적으로 캐싱은 켜져 있지 않지만 API 수준에서 활성화하거나 함수의 반환 값에 `ttlOverride` 값을 설정하여 활성화할 수 있습니다.

필요한 경우 함수를 호출하기 전에 권한 부여 토큰의 유효성을 검사하는 정규식을 지정할 수 있습니다. 이러한 정규식은 함수를 호출하기 전에 권한 부여 토큰의 형식이 올바른지 확인하는 데 사용됩니다. 이 정규식과 일치하지 않는 토큰을 사용하는 모든 요청은 자동으로 거부됩니다.

권한 부여에 사용되는 Lambda 함수의 호출을 AWS AppSync 허용하려면 기본 정책을 `appsync.amazonaws.com` 적용해야 합니다. 이 작업은 콘솔에서 자동으로 수행되지만 AWS AppSync 콘솔은 AWS AppSync 정책을 제거하지 않습니다. Lambda 함수에 정책을 연결하는 방법에 대한 자세한 내용은 개발자 안내서의 리소스 기반 정책을 참조하십시오. AWS Lambda

지정하는 Lambda 함수는 다음과 같은 형태의 이벤트를 수신합니다.

```
{
    "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
    }
}
```

event객체에는 애플리케이션 클라이언트에서 요청을 통해 전송된 헤더가 포함되어 있습니다. AWS AppSync

권한 부여 함수는 최소한 isAuthorized 요청이 승인되었는지 여부를 나타내는 불리언을 반환해야 합니다. AWS AppSync Lambda 인증 함수에서 반환된 다음 키를 인식합니다.

함수 목록

isAuthorized(부울, 필수)

authorizationToken의 값이 GraphQL API를 호출할 수 있는 권한이 있는지 여부를 나타내는 부울 값입니다.

이 값이 참이면 GraphQL API가 계속 실행됩니다. 이 값이 거짓이면 UnauthorizedException이 발생합니다.

deniedFields(문자열 목록, 선택 사항)

해석기에서 값이 반환된 경우에도 문자열 목록이 null로 강제 변경됩니다.

각 항목은 arn:aws:appsync:*us-east-1*:*111122223333*:apis/*GraphQLApiId*/types/*TypeName*/fields/*FieldName* 형식 또는 축약된 *TypeName*.*FieldName* 형식의 정규화된 필드 ARN입니다. 두 API가 Lambda 함수 권한 부여자를 공유하는 경우 전체 ARN 양식을 사용해야 하며 두 API 간에 공통 유형과 필드 간에 모호성이 있을 수 있습니다.

`resolverContext`(JSON 객체, 선택 사항)

해석기 템플릿에 `$ctx.identity.resolverContext`로 표시되는 JSON 객체입니다. 예를 들어 해석기에서 다음 구조를 반환하는 경우:

```
{
  "isAuthorized":true
  "resolverContext": {
    "banana":"very yellow",
    "apple":"very green"
  }
}
```

해석기 템플릿의 `ctx.identity.resolverContext.apple` 값은 `"very green"`입니다. `resolverContext` 객체는 키-값 쌍만 지원합니다. 중첩 키는 지원되지 않습니다.

> **Warning**
>
> 이 JSON 객체의 총 크기는 5MB를 초과할 수 없습니다.

`ttlOverride`(정수, 선택 사항)

응답을 캐시해야 하는 시간(초)입니다. 반환된 값이 없으면 API의 값이 사용됩니다. 이 값이 0이면 응답이 캐시되지 않습니다.

Lambda 권한 부여자의 제한 시간은 10초입니다. API의 성능을 확장하려면 가능한 한 최단 시간 내에 실행되도록 함수를 설계하는 것이 좋습니다.

여러 AWS AppSync API가 단일 인증 Lambda 함수를 공유할 수 있습니다. 교차 계정 권한 부여자 사용은 허용되지 않습니다.

여러 API 간에 권한 부여 함수를 공유하는 경우, 축약된 형식의 필드 이름(*typename*.*fieldname*)이 의도치 않게 필드를 숨길 수 있다는 점에 유의하세요. `deniedFields` 필드의 모호성을 없애려면 `arn:aws:appsync:`*region*`:`*accountId*`:apis/`*GraphQLApiId*`/types/`*typeName*`/fields/`*fieldName* 형식으로 명확한 필드 ARN을 지정하세요.

Lambda 함수를 AWS AppSync의 기본 권한 부여 모드로 추가하려면 다음을 따르세요.

Console

1. AWS AppSync 콘솔에 로그인하고 업데이트하려는 API로 이동합니다.
2. API의 설정 페이지로 이동합니다.

   API 수준 권한 부여를 AWS Lambda으로 변경합니다.
3. API 호출을 승인할 대상 AWS 리전 및 Lambda ARN을 선택합니다.

   > **Note**
   >
   > 적절한 보안 주체 정책이 자동으로 추가되어 AWS AppSync 에서 Lambda 함수를 호출할 수 있습니다.
4. 필요한 경우 응답 TTL 및 토큰 검증 정규식을 설정합니다.

AWS CLI

1. 사용 중인 Lambda 함수에 다음 정책을 연결합니다.

```
aws lambda add-permission --function-name "my-function" --statement-id "appsync"
 --principal appsync.amazonaws.com --action lambda:InvokeFunction --output text
```

> **Important**
>
> 함수 정책을 단일 GraphQL API로 고정하려면 다음 명령을 실행하세요.
>
> ```
> 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. 지정된 Lambda 함수 ARN을 권한 부여자로 사용하도록 AWS AppSync API를 업데이트하십시오.

```
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**
>
> 토큰 정규식과 같은 기타 구성 옵션을 포함할 수도 있습니다.

다음 예는 Lambda 함수가 AWS AppSync 권한 부여 메커니즘으로 사용될 때 가질 수 있는 다양한 인증 및 결함 상태를 보여주는 Lambda 함수를 설명합니다.

```
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 {}
```

SigV4 및 OIDC 토큰 권한 부여 제한 방지

다음 방법을 사용하여 특정 권한 부여 모드가 활성화된 경우 SigV4 서명 또는 OIDC 토큰을 Lambda 권한 부여 토큰으로 사용할 수 없는 문제를 방지할 수 있습니다.

AWS AppSync의 API에 대해 `AWS_IAM` 및 `AWS_LAMBDA` 권한 부여 모드가 활성화되어 있을 때 SigV4 서명을 Lambda 권한 부여 토큰으로 사용하려면 다음을 따르세요.

- 새 Lambda 권한 부여 토큰을 생성하려면 SigV4 서명에 임의의 접미사 또는 접두사를 추가하세요.
- 원본 SigV4 서명을 검색하려면 Lambda 권한 부여 토큰에서 임의의 접두사 또는 접미사를 제거하여 Lambda 함수를 업데이트하세요. 그런 다음 원본 SigV4 서명을 인증에 사용합니다.

의 API에 대해 권한 부여 모드 또는 `AMAZON_COGNITO_USER_POOLS` 및 권한 `AWS_LAMBDA` 부여 모드가 활성화되어 있을 때 OIDC 토큰을 Lambda 인증 토큰으로 사용하려면 다음을 수행하십시오 AWS AppSync. `OPENID_CONNECT`

- 새 Lambda 권한 부여 토큰을 생성하려면 OIDC 토큰에 임의의 접미사 또는 접두사를 추가하세요. Lambda 권한 부여 토큰에는 보유자 체계 접두사를 포함할 수 없습니다.
- 원본 OIDC 토큰을 검색하려면 Lambda 권한 부여 토큰에서 임의의 접두사 또는 접미사를 제거하여 Lambda 함수를 업데이트하세요. 그런 다음 원본 OIDC 토큰을 인증에 사용합니다.

AWS_IAM 권한 부여

이 권한 부여 유형은 GraphQL API에 대해 AWS 서명 버전 4 서명 프로세스를 적용합니다. IAM(Identity and Access Management) 액세스 정책을 권한 부여 유형과 연결할 수 있습니다. 애플리케이션에서는 액세스 키(액세스 키 ID와 비밀 액세스 키로 구성됨)를 사용하거나 Amazon Cognito 연동 자격 증명에서 제공한 단기 임시 보안 인증을 사용하여 이 연결을 활용할 수 있습니다.

모든 데이터 작업을 수행하기 위한 액세스 권한이 있는 역할이 필요한 경우:

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

```
      {
         "Effect": "Allow",
         "Action": [
            "appsync:GraphQL"
         ],
         "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/*"
         ]
      }
   ]
}
```

AppSync콘솔의 기본 API 목록 페이지에서 API 이름 바로 아래에서 찾을 `YourGraphQLApiId` 수 있습니다. 또한 CLI `aws appsync list-graphql-apis`를 사용하여 검색할 수도 있습니다.

특정 GraphQL 작업으로 액세스를 제한하려는 경우 루트 `Query`, `Mutation` 및 `Subscription` 필드에 대해 액세스를 제한할 수 있습니다.

```
{
   "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>"
         ]
     }
   ]
}
```

예를 들어, 다음 스키마를 가정하고 모든 게시글 가져오기에 대한 액세스를 제한하려고 합니다.

```
schema {
```

```
   query: Query
   mutation: Mutation
}

type Query {
   posts:[Post!]!
}

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

(예를 들어 Amazon Cognito 자격 증명 풀에 연결할 수 있는) 역할에 해당하는 IAM 정책은 다음과 같습니다.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
            "appsync:GraphQL"
            ],
            "Resource": [
                "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/
Query/fields/posts"
            ]
        }
    ]
}
```

OPENID_CONNECT 권한 부여

이 권한 부여 유형은 OIDC 호환 서비스에서 제공하는 OpenID 연결(OIDC) 토큰을 강제로 적용합니다. 애플리케이션에서는 OIDC 공급자가 액세스 제어를 위해 정의한 사용자 및 권한을 활용합니다.

발급자 URL은 제공하는 유일한 필수 구성 값입니다 AWS AppSync (예:`https://auth.example.com`). 이 URL은 HTTPS를 통해 주소를 지정할 수 있어야 합니다. AWS AppSync 발급자 `/.well-known/openid-configuration` URL에 추가하고 OpenID Connect Discovery 사양에 따라 `https://auth.example.com/.well-known/openid-configuration` OpenID 구성을 찾습니다. 이 URL에서 RFC5785 호환 JSON 문서를 검색해야 합니다. 이 JSON 문서에는 서명 키가

있는 JSON 웹 `jwks_uri` 키 세트 (JWKS) 문서를 가리키는 키가 포함되어야 합니다. AWS AppSync JWKS에는 및 의 JSON 필드가 포함되어야 합니다. `kty` `kid`

AWS AppSync 다양한 서명 알고리즘을 지원합니다.

서명 알고리즘
RS256
RS384
RS512
PS256
PS384
PS512
HS256
HS384
HS512
ES256
ES384
ES512

RSA 알고리즘을 사용하는 것이 좋습니다. 공급자가 발행하는 토큰에는 토큰이 발행된 시간(`iat`)이 포함되어 있어야 하며 토큰이 인증된 시간(`auth_time`)이 포함될 수 있습니다. 추가 확인을 위해 발행된 시간에 대한 TTL 값(`iat`TTL)과 OpenID Connect 구성의 인증 시간(`auth`TTL)을 제공할 수 있습니다. 공급자가 여러 애플리케이션을 인증하면 클라이언트 ID별로 인증하는 데 사용되는 정규식(`clientId`)도 제공할 수 있습니다. `clientId`OpenID Connect 구성에 이 있는 경우 가 토큰의 `aud` 또는 클레임과 `clientId` 일치하도록 요구하여 `azp` 클레임을 AWS AppSync 검증합니다.

여러 클라이언트 ID를 검증하려면 정규식에서는 'or'인 파이프라인 연산자('|')를 사용합니다. 예를 들어 OIDC 애플리케이션에 0A1S2D, 1F4G9H, 1J6L4B, 6GS5MG 등의 클라이언트 ID를 가진 클라이언

트 4개가 있고 처음 세 개의 클라이언트 ID만 검증하는 경우 클라이언트 ID 필드에 1F4G9H|1J6L4B|6GS5MG를 입력합니다.

AMAZON_COGNITO_USER_POOLS 권한 부여

이 권한 부여 유형은 Amazon Cognito 사용자 풀에서 제공한 OIDC 토큰을 적용합니다. 응용 프로그램은 다른 AWS 계정의 사용자 풀과 사용자 풀 모두에 있는 사용자와 그룹을 활용하고 이를 GraphQL 필드와 연결하여 액세스를 제어할 수 있습니다.

Amazon Cognito 사용자 풀을 사용하는 경우 사용자가 속한 그룹을 생성할 수 있습니다. 이 정보는 애플리케이션이 GraphQL 작업을 전송할 때 인증 헤더로 AWS AppSync 보내는 JWT 토큰으로 인코딩됩니다. 스키마에 GraphQL 명령을 사용하여 어떤 그룹이 필드의 어떤 해석기를 호출할 수 있는지 제어하여 고객에게 보다 정확하게 제어되는 액세스 권한을 제공할 수 있습니다.

예를 들어 다음과 같은 GraphQL 스키마가 있다고 가정하겠습니다.

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

type Query {
   posts:[Post!]!
}

type Mutation {
   addPost(id:ID!, title:String!):Post!
}
...
```

Amazon Cognito 사용자 풀에 블로거 및 독자라는 두 가지 그룹이 있고 새 항목을 추가할 수 없도록 독자를 제한하려는 경우 스키마는 다음과 같아야 합니다.

```
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"])
}
...
```

단, 액세스에 대한 특정 전략을 기본값으로 설정하려는 경우 @aws_auth 지시문을 생략할 수 있습니다. grant-or-deny 콘솔이나 다음 CLI 명령을 통해 GraphQL API를 생성할 때 사용자 풀 구성에서 grant-or-deny 전략을 지정할 수 있습니다.

```
$ 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"}'
```

추가 권한 부여 모드 사용

추가 권한 부여 모드를 추가하면 AWS AppSync GraphQL API 수준 (즉, authenticationType 객체에 직접 구성할 수 있는 필드) 에서 권한 부여 설정을 직접 구성할 수 있으며 GraphqlApi 이 설정은 스키마에서 기본값으로 작동합니다. 다시 말해서, 특정 지시문이 없는 모든 유형은 API 수준 권한 부여 설정을 전달해야 합니다.

스키마 수준에서 스키마에 대한 지시문을 사용하여 추가 권한 부여 모드를 지정할 수 있습니다 스키마의 개별 필드에서 권한 부여 모드를 지정할 수 있습니다. 예를 들어, API_KEY 권한 부여의 경우 스키마 객체 유형 정의/필드에서 @aws_api_key를 사용합니다. 스키마 필드 및 객체 유형 정의에서는 다음 지시문이 지원됩니다.

- @aws_api_key - 필드에 API_KEY 권한이 부여되도록 지정합니다.
- @aws_iam - 필드에 AWS_IAM 권한이 부여되도록 지정합니다.
- @aws_oidc - 필드에 OPENID_CONNECT 권한이 부여되도록 지정합니다.
- @aws_cognito_user_pools - 필드에 AMAZON_COGNITO_USER_POOLS 권한이 부여되도록 지정합니다.
- @aws_lambda - 필드에 AWS_LAMBDA 권한이 부여되도록 지정합니다.

@aws_auth 지시문을 추가 권한 부여 모드와 함께 사용할 수 없습니다. @aws_auth는 추가 권한 부여 모드가 없는 AMAZON_COGNITO_USER_POOLS 권한 부여의 컨텍스트에서만 작동합니

다. 하지만 샘플 인수를 사용하여 @aws_auth 지시문 대신에 @aws_cognito_user_pools 지시문을 사용할 수 있습니다. 두 지시문 간의 주요 차이점은 모든 필드와 객체 유형 정의에서 @aws_cognito_user_pools를 지정할 수 있다는 것입니다.

추가 권한 부여 모드가 작동하는 방식과 스키마에서 추가 권한 부여 모드를 지정할 수 있는 방법을 이해하기 위해 다음 스키마를 살펴보겠습니다.

```
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!
}
...
```

이 스키마의 경우 AWS AppSync GraphQL AWS_IAM API의 기본 권한 부여 유형이라고 가정합니다. 다시 말해서, 지시문이 없는 필드는 AWS_IAM을 사용하여 보호됩니다. 예를 들어, Query 유형에서

getPost 필드의 경우에도 마찬가지입니다. 스키마 지시문을 사용하면 두 개 이상의 권한 부여 모드를 사용할 수 있습니다. 예를 들어, AWS AppSync GraphQL API에서 추가 권한 부여 모드로 API_KEY 구성하고 지시문을 사용하여 필드를 표시할 수 있습니다 (예: @aws_api_key getAllPosts 이 예제에서는). 지시문은 필드 수준에서 작동하므로 API_KEY에 Post 유형에 대한 액세스 권한도 부여해야 합니다. 지시문으로 Post 유형의 각 필드에 표시하거나 @aws_api_key 지시문으로 Post 유형에 표시하여 이렇게 할 수 있습니다.

Post 유형의 필드에 대한 액세스를 한층 더 제한하려면 다음과 같이 Post 유형의 개별 필드에 대해 지시문을 사용할 수 있습니다.

예를 들어, @aws_iam 지시문을 사용하여 restrictedContent 필드를 Post 유형에 추가하고 이 필드에 대한 액세스를 제한할 수 있습니다. AWS_IAM 권한이 있는 요청은 restrictedContent에 액세스할 수 있지만, API_KEY 요청은 액세스할 수 없습니다.

```
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
}
...
```

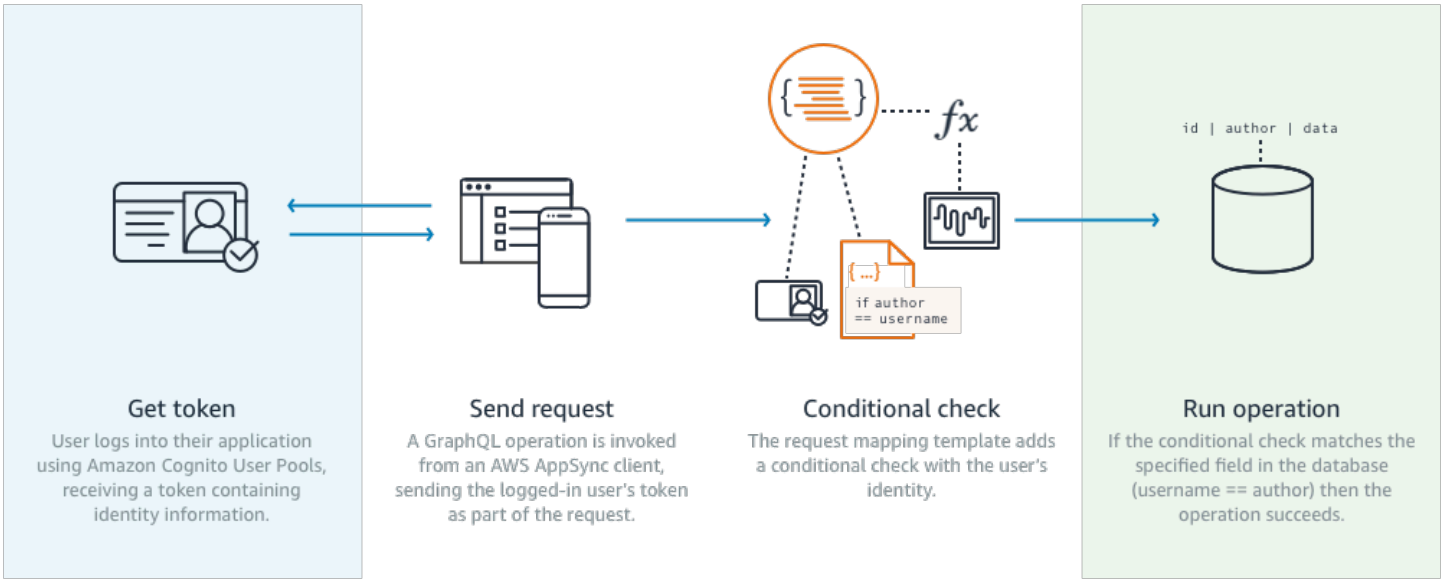
세분화된 액세스 제어

앞선 정보는 특정 GraphQL 필드에 대한 액세스 제한 또는 부여 방법을 설명합니다. 특정 조건을 기준으로(예를 들어, 호출하는 사용자 및 해당 사용자가 데이터를 소유하는지 여부를 기준으로) 데이터에 대한 액세스 제어를 설정하려는 경우 해석기에서 매핑 템플릿을 사용할 수 있습니다. 또한 필터링 정보에서 설명하는 보다 복잡한 비즈니스 로직을 수행할 수도 있습니다.

이 섹션에서는 DynamoDB 해석기 매핑 템플릿을 사용하여 데이터에 대한 액세스 제어를 설정하는 방법을 보여줍니다.

계속 진행하기 전에 의 매핑 템플릿에 익숙하지 않다면 DynamoDB의 AWS AppSync Resolver 매핑 템플릿 참조와 Resolver 매핑 템플릿 참조를 검토해 보는 것이 좋습니다.

DynamoDB를 사용하는 다음 예에서는 이전 블로그 게시 스키마를 사용하고 게시글을 작성한 사용자만 해당 게시글을 편집할 수 있도록 허용한다고 가정합니다. 평가 프로세스는 사용자가 예를 들어, Amazon Cognito 사용자 풀을 사용하여 애플리케이션에서 자격 증명을 얻은 다음 GraphQL 작업의 일부로 해당 자격 증명을 전달하도록 하는 과정입니다. 그런 다음 매핑 템플릿은 조건문에서 자격 증명(예: 사용자 이름)의 값을 대체하고 이 값을 데이터베이스의 값과 비교합니다.



이 기능을 추가하려면 다음과 같이 GraphQL 필드 `editPost`를 추가합니다.

```
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!
}
...
```

`editPost`의 해석기 매핑 템플릿(이 단원의 끝에 나오는 예 참조)은 데이터 스토어에 대한 로직 점검을 수행해 게시글을 생성한 사용자만 해당 게시글을 편집하도록 해야 합니다. 이 작업은 편집 작업이기 때문에 DynamoDB에서는 `UpdateItem`에 해당합니다. 이 작업을 수행하기 전에 사용자 자격 검증을

통해 전달된 컨텍스트로 조건 검사를 수행할 수 있습니다. 이 컨텍스트는 다음 값을 가지는 Identity 객체에 저장됩니다.

```
{
   "accountId" : "12321434323",
   "cognitoIdentityPoolId" : "",
   "cognitoIdentityId" : "",
   "sourceIP" : "",
   "caller" : "ThisistheprincipalARN",
   "username" : "username",
   "userArn" : "Sameasabove"
}
```

DynamoDBUpdateItem 호출 시 이 객체를 사용하려면 비교를 위해 테이블에 사용자 자격 증명 정보를 저장해야 합니다. 먼저, addPost 변형이 생성자를 저장해야 합니다. 그 다음으로, editPost 변형이 업데이트 전에 조건 검사를 수행해야 합니다.

다음은 사용자 ID를 Author열로 저장하는 addPost에 대한 해석기 코드의 예시입니다.

```
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;
```

Author 속성은 애플리케이션에서 가져온 Identity 객체의 속성으로 채워집니다.

마지막으로, 다음은 게시글을 작성한 사용자가 요청한 경우에만 블로그 게시글의 내용을 업데이트하는 editPost의 해석기 코드의 예시입니다.

```
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;
```

이 예시에서는 UpdateItem보다는 모든 값을 덮어쓰는 PutItem을 사용하지만, 동일한 개념이 condition 문 블록에 적용됩니다.

필터링 정보

데이터 원본의 응답을 제어할 수 없는 경우가 있을 수 있지만 데이터 원본에 쓰기 또는 읽기 성공에 대한 불필요한 정보를 클라이언트에게 보내고 싶지 않습니다. 이러한 경우 응답 매핑 템플릿을 사용하여 정보를 필터링할 수 있습니다.

예를 들어, 블로그 게시물 DynamoDB 테이블에 대한 적절한 인덱스(예: Author에 대한 인덱스)가 없다고 가정해 보겠습니다. 다음 해석기를 사용할 수 있습니다.

```
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;
}
```

요청 핸들러는 발신자가 게시물을 만든 작성자가 아니더라도 항목을 가져옵니다. 이렇게 하면 모든 데이터가 반환되지 않도록 응답 핸들러는 발신자가 항목 작성자와 일치하는지 확인합니다. 호출자가 이 검사와 일치하지 않는 경우 null 응답만 반환됩니다.

데이터 원본 액세스

AWS AppSync ID 및 액세스 관리 (IAM) 역할 및 액세스 정책을 사용하여 데이터 소스와 통신합니다. 기존 역할을 사용하는 경우 역할을 수임하려면 신뢰 정책을 추가해야 합니다. AWS AppSync 신뢰 관계는 아래와 같아야 합니다.

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

해당 역할에 대한 액세스 정책을 축소하여 권한을 필요한 최소한의 원본 집합에만 허용하는 것이 중요합니다. AppSync 콘솔을 사용하여 데이터 원본을 만들고 역할을 만들면 이 작업이 자동으로 수행됩니다. 하지만 IAM 콘솔의 기본 제공 샘플 템플릿을 사용하여 콘솔 외부에서 역할을 생성하는 경우 리소스에 대한 권한 범위가 자동으로 축소되지 않으므로 애플리케이션을 프로덕션으로 이동하기 전에 이 작업을 수행해야 합니다. AWS AppSync

권한 부여 사용 사례

보안 단원에서는 API 보호를 위한 다양한 권한 부여 모드에 대해 알아보았으며 개념과 흐름을 이해할 수 있도록 세분화된 권한 부여 메커니즘을 소개했습니다. AWS AppSync에서는 GraphQL 해석기 매핑 템플릿 사용을 통해 데이터에 대한 모든 논리적 작업을 수행할 수 있으므로 사용자 자격 증명, 조건문 및 데이터 삽입을 결합하여 매우 유연한 방식으로 데이터 읽기/쓰기를 보호할 수 있습니다.

AWS AppSync 해석기 편집에 대해 잘 모르는 경우 프로그래밍 가이드를 살펴보세요.

개요

지금까지는 시스템의 데이터에 대한 액세스 권한 부여가 액세스 제어 매트릭스를 통해 수행되었으며, 이 매트릭스에서 행(리소스)과 열(사용자/역할)의 교차점이 부여되는 권한이었습니다.

AWS AppSync는 사용자 고유 계정의 리소스를 사용하여 자격 증명(사용자/역할) 정보를 GraphQL 요청 및 응답에 컨텍스트 객체(해석기에서 사용할 수 있음)로서 삽입합니다. 다시 말해서 해석기 로직에 따라 쓰기 또는 읽기 작업에 대한 권한을 적절히 부여할 수 있습니다. 예를 들어 이름이 지정된 특정 사용자 또는 그룹만 특정 데이터베이스 행을 읽고 쓸 수 있는 등 이 로직이 리소스 수준인 경우 해당 "권한 부여 메타데이터"를 저장해야 합니다. AWS AppSync는 데이터를 저장하지 않으므로 권한을 계산할 수 있도록 이 권한 부여 메타데이터를 리소스와 함께 저장해야 합니다. 권한 부여 메타데이터는 일반적으로 소유자 또는 사용자/그룹 목록 같은 DynamoDB 테이블에 속성(열)입니다. 예를 들면 Readers 및 Writers 속성이 있을 수 있습니다.

다시 말해, 상위 수준에서는 데이터 원본의 개별 항목을 읽으려는 경우 해석기를 데이터 원본에서 읽어온 후 응답 템플릿에서 조건부 `#if () ... #end` 문을 수행할 수 있다는 뜻입니다. 이 확인은 일반적으로 읽기 작업에서 반환되는 권한 부여 메타데이터와 비교하여 멤버십을 확인하기 위해 `$context.identity`의 사용자 또는 그룹 값을 사용합니다. `Scan` 또는 `Query` 테이블에서 반환되는 목록과 같이 여러 레코드의 경우, 유사한 사용자 또는 그룹 값을 사용하여 작업의 일부로서 조건부 확인을 데이터 원본으로 전송합니다.

데이터를 작성할 때와 마찬가지로 조건문을 작업(예: `PutItem` 또는 `UpdateItem`)에 적용하여 변형을 만드는 사용자나 그룹이 권한을 가지고 있는지 여부를 확인합니다. 이 조건문은 `$context.identity`의 값 하나를 여러 번 다시 사용하여 해당 리소스에 대한 권한 부여 메타데이터와 비교합니다. 요청 및 응답 템플릿 모두에 대해 클라이언트의 사용자 지정 헤더를 사용하여 검증 확인을 수행할 수도 있습니다.

데이터 읽기

위에 요약된 바와 같이 확인을 수행할 권한 부여 메타데이터를 리소스와 함께 저장하거나 GraphQL 요청(자격 증명, 헤더 등)에 전달해야 합니다. 이를 설명하기 위해 아래에 나오는 DynamoDB 테이블이 있다고 가정하겠습니다.

ID	Data	PeopleCanAccess	GroupsCanAccess	Owner
123	{my: data,...}	[Mary, Joe]	[Admins, Editors]	Nadia

기본 키는 `id`이며 액세스해야 할 데이터는 `Data`입니다. 다른 열은 승인을 위해 수행할 수 있는 확인의 예입니다. DynamoDB에 대한 해석기 매핑 템플릿 참조에 설명된 대로 `Owner`는 `String`이고 `PeopleCanAccess` 및 `GroupsCanAccess`는 `String Sets`입니다.

해석기 매핑 템플릿 개요의 다이어그램은 응답 템플릿에 컨텍스트 객체는 물론 데이터 원본의 결과도 포함됨을 보여줍니다. 개별 항목의 GraphQL 쿼리의 경우 응답 템플릿을 사용하여 사용자에게 이러한

결과를 보도록 허용되었는지 여부를 확인하거나 아니면 권한 부여 오류 메시지를 반환할 수 있습니다. 이를 때로는 "권한 부여 필터"라고도 합니다. 스캔 또는 쿼리를 사용하는 GraphQL 쿼리 반환 목록의 경우, 요청 템플릿에 대한 확인을 수행하고 권한 부여 조건이 충족되는 경우에만 데이터를 반환하는 것이 더 바람직합니다. 구현은 다음과 같습니다.

1. GetItem - 개별 레코드에 대한 권한 부여 확인입니다. `#if() ... #end` 문을 사용하여 수행됩니다.
2. 스캔/쿼리 작업 - 권한 부여 확인이 `"filter":{"expression":...}` 문입니다. 일반적으로 같음(`attribute = :input`)을 확인하거나 목록에 값(`contains(attribute, :input)`)이 있는지 여부를 확인합니다.

#2에서는 두 명령문의 `attribute`가 위의 예제의 `Owner`와 같은 테이블 내 레코드 열 이름을 나타냅니다. # 기호를 사용하여 이 이름을 별칭으로 지정하고 `"expressionNames":{...}`를 사용할 수 있지만, 필수는 아닙니다. `:input`은 데이터베이스 속성과 비교할 값에 대한 참조로, `"expressionValues":{...}`에서 정의합니다. 다음과 같은 예제가 표시됩니다.

사용 사례: 소유자에게 읽기 허용

위의 테이블을 사용할 때, 개별 읽기 작업(`Owner == Nadia`)에 대해 `GetItem`일 때만 데이터를 반환하려는 경우 템플릿은 다음과 같습니다.

```
#if($context.result["Owner"] == $context.identity.username)
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

여기에 언급된 몇 가지 사항은 나머지 단원에서 재사용될 것입니다. 먼저 이 확인은 `$context.identity.username`을 사용하는데, 이 이름은 Amazon Cognito 사용자 풀이 사용되는 경우 친숙한 사용자 등록 이름이며 IAM이 사용되는 경우 사용자 자격 증명(Amazon Cognito 연동 자격 증명 포함)입니다. 고유한 'Amazon Cognito 자격 증명' 값(여러 위치에서 로그인을 연동할 때 유용함)과 같이 소유자에 대해 저장할 다른 값도 있으며, 해석기 매핑 템플릿 컨텍스트 참조에서 사용 가능한 옵션을 검토해야 합니다.

두 번째, `$util.unauthorized()`를 사용한 else 조건 확인 응답은 완전히 선택 사항이지만 GraphQL API를 설계할 때 모범 사례로서 권장됩니다.

사용 사례: 특정 액세스 하드코딩

```
// 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
```

사용 사례: 결과 목록 필터링

이전 예에서는 단일 항목을 반환하므로 `$context.result`에 대해 확인을 직접 수행할 수 있었지만, 스캔과 같은 일부 작업은 `$context.result.items`에서 여러 항목을 반환합니다. 이 경우에는 권한 부여 필터를 수행하여 사용자에게 보기가 허용된 결과만 반환해야 합니다. `Owner` 필드에서 현재 Amazon Cognito IdentityID가 레코드에 대해 설정되었다고 가정해 보겠습니다. 이제 다음 응답 매핑 템플릿을 사용하여 사용자가 소유하는 레코드만 표시하도록 필터링할 수 있습니다.

```
#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)
```

사용 사례: 여러 사람에게 읽기 허용

또 한 가지 널리 사용되는 권한 부여 옵션은 사람 그룹에 데이터 읽기를 허용하는 것입니다. 아래 예에서는 GraphQL 쿼리를 실행하는 사용자가 `"filter":{"expression":...}` 세트에 나열되는 경우에만 PeopleCanAccess가 테이블 스캔에서 값을 반환합니다.

```
{
    "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)
        }
    }
}
```

사용 사례: 그룹에 읽기 허용

마지막 사용 사례와 마찬가지로, 이 사용 사례에서는 하나 이상의 그룹 내 사람들만 데이터베이스의 특정 항목을 읽을 수 있습니다. `"expression": "contains()"` 작업의 사용도 비슷하지만, 이것은 설정된 멤버십에서 사용자가 속해 있을 수 있는 모든 그룹의 논리 OR입니다. 여기서는 사용자가 속한 각 그룹에 대해 아래 $expression 문을 작성한 다음 이 문을 필터에 전달합니다.

```
#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)
    }
}
```

데이터 쓰기

변형에 대한 데이터 쓰기는 항상 요청 매핑 템플릿에서 제어됩니다. DynamoDB 데이터 원본의 경우, 핵심은 해당 테이블의 권한 부여 메타데이터를 확인하는 적정 `"condition": {"expression"...}"`을 사용하는 것입니다. 보안 단원에 테이블의 `Author` 필드를 확인할 수 있는 예제를 제공했습니다. 이 단원의 사용 사례에서는 더 많은 사용 사례를 살펴봅니다.

사용 사례: 여러 소유자

앞의 예제 테이블 다이어그램을 사용하면서, `PeopleCanAccess` 목록을 가정해 보겠습니다.

```
{
    "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)
        }
    }
}
```

사용 사례: 그룹에 새 레코드 생성 허용

```
#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)
    }
}
```

사용 사례: 그룹에 기존 레코드 업데이트 허용

```
#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)
    }
}
```

퍼블릭 및 프라이빗 레코드

또한 조건부 필터를 사용하여 데이터를 프라이빗, 퍼블릭 또는 다른 부울 확인으로 표시하도록 선택할 수도 있습니다. 이렇게 하면 조건 필터를 응답 템플릿 내부에서 권한 부여 필터의 일부로서 결합할 수 있습니다. 이 확인을 사용하는 것은 그룹 멤버십을 제어하지 않고 데이터를 임시로 숨기거나 뷰에서 제거할 수 있는 좋은 방법입니다.

예를 들어 DynamoDB 테이블의 각 항목에 대해 `yes` 또는 `no` 값을 사용하여 `public`이라는 속성을 추가했다고 가정하겠습니다. 다음 응답 템플릿을 `GetItem` 호출에서 사용하면 액세스 권한이 있는 그룹 내 사용자이고(AND) 해당 데이터가 퍼블릭으로 표시되는 경우에 한하여 데이터를 표시할 수 있습니다.

```
#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
```

또한 위의 코드는 OR(||) 로직을 사용하여 퍼블릭 레코드인 경우 또는 레코드에 대한 권한이 있는 사람에게만 읽기를 허용할 수도 있습니다.

```
#if($hasPermission || $context.result.public == 'yes')
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

일반적으로 표준 연산자인 ==, !=, && 및 ||는 권한 부여 확인 수행 시 유용합니다.

실시간 데이터

이 설명서의 앞부분에 설명된 동일한 기술을 사용하여 클라이언트가 구독을 만들 때 세분화된 액세스 제어를 GraphQL 구독에 적용할 수 있습니다. 구독 필드에 해석기를 연결할 수 있으며, 이때 데이터 원본에서 데이터를 쿼리하고 요청이나 응답 매핑 템플릿에서 조건부 로직을 수행할 수 있습니다. 또한 데이터 구조가 GraphQL 구독의 반환 유형과 일치하는 한, 구독의 초기 결과 등 추가 데이터를 클라이언트로 반환할 수도 있습니다.

사용 사례: 사용자가 특정 대화만 구독 가능

GrapHQL 구독과 관련된 실시간 데이터의 일반 사용 사례 중 하나는 메시징 또는 프라이빗 채팅 애플리케이션 빌드입니다. 여러 사용자가 있는 채팅 애플리케이션을 작성하려는 경우 두 사람 이상의 여러

사람 간에 대화가 발생할 수 있습니다. 이러한 대화는 프라이빗이나 퍼블릭 "대화방"으로 그룹화될 수 있습니다. 따라서, 자신에게 액세스 권한이 부여된 대화(한 개 이상의 그룹일 수 있음)만 구독하도록 사용자에게 권한을 부여할 수 있습니다. 데모 목적으로 제공되는 아래 샘플은 한 사용자가 프라이빗 메시지를 다른 사용자에게 전송하는 단순 사용 사례를 보여줍니다. 설정에는 두 개의 Amazon DynamoDB 테이블이 있습니다.

- Messages 테이블: (기본 키)`toUser`, (정렬 키)`id`
- Permissions 테이블: (기본 키)`username`

Messages 테이블은 GraphQL 변형을 통해 전송되는 실제 메시지를 저장합니다. Permissions 테이블은 클라이언트 연결 시 권한 부여를 위해 GraphQL 구독에서 확인됩니다. 아래 예제에서는 다음과 같은 GraphQL 스키마를 사용한다고 가정합니다.

```
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
}
```

createUserPermissions()와 같은 표준 작업 중 일부는 구독 해석기를 보여주기 위해 아래에 포함되지 않지만, DynamoDB 해석기의 표준 구현입니다. 대신에 해석기를 사용하는 구독 권한 부여 흐름을 중점적으로 살펴보겠습니다. 한 사용자에서 다른 사용자로 메시지를 전송하려면 해석기를 sendMessage() 필드에 연결하고 다음과 같은 요청 템플릿과 함께 Messages 테이블 데이터 원본을 선택합니다.

```
{
    "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),
    }
}
```

이 예제에서는 $context.identity.username을 사용합니다. 이렇게 하면 AWS Identity and Access Management 또는 Amazon Cognito 사용자에 대한 사용자 정보가 반환됩니다. 응답 템플릿은

$util.toJson($ctx.result)의 단순 전달입니다. 저장하고 스키마 페이지로 돌아갑니다. 그런 다음 권한 테이블을 데이터 소스로 사용하고 다음 요청 매핑 템플릿을 사용하여 newMessage() 구독용 해석기를 연결합니다.

```
{
    "version": "2018-05-29",
    "operation": "GetItem",
    "key": {
        "username": $util.dynamodb.toDynamoDBJson($ctx.identity.username),
    },
}
```

그러고 나면 다음과 같은 응답 매핑 템플릿과 Permissions 테이블의 데이터를 사용하여 권한 부여 확인을 수행합니다.

```
#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
```

여기서는 세 가지 권한 부여 확인을 수행하려고 합니다. 첫 번째는 결과가 반환되는지 확인합니다. 두 번째는 사용자가 다른 사람에게 해당되는 메시지를 구독하고 있지 않은지 확인합니다. 세 번째는 BOOL로 저장된 isAuthorizedForSubscriptions의 DynamoDB 속성을 확인하여 사용자가 모든 필드를 구독하도록 허용되었는지 확인합니다.

테스트하기 위해 Amazon Cognito 사용자 풀과 사용자 이름 'Nadia'를 사용하여 AWS AppSync 콘솔에 로그인하고 나서 다음 GraphQL 구독을 실행할 수 있습니다.

```
subscription AuthorizedSubscription {
    newMessage(toUser: "Nadia") {
        id
        toUser
        fromUser
        content
    }
```

```
}
```

Permissions 테이블에서 `isAuthorizedForSubscriptions`가 `true`로 설정된 `Nadia`의 키 속성 `username`에 대한 레코드가 있는 경우 성공적인 응답이 표시됩니다. 위의 `username` 쿼리에서 다른 `newMessage()`을 시도하면 오류가 반환됩니다.

AWS WAF을 사용하여 API 보호

AWS WAF는 웹 애플리케이션 및 API를 공격으로부터 보호하는 데 도움이 되는 웹 애플리케이션 방화벽입니다. WAF를 통해 사용자가 정의한 맞춤형 웹 보안 규칙 및 조건에 따라 웹 요청을 허용, 차단 또는 모니터링하는 규칙 집합(이른바 웹 액세스 제어 목록(웹 ACL)이라고 함)을 구성할 수 있습니다. AWS AppSync API를 AWS WAF와 통합하면 API에서 허용하는 HTTP 트래픽을 더 잘 제어하고 가시성을 확보할 수 있습니다. AWS WAF에 대한 자세한 내용은 AWS WAF 개발자 안내서의 AWS WAF 작동 방식을 참조하세요.

AWS WAF를 사용하여 SQL 명령어 주입 및 교차 사이트 스크립팅(XSS) 공격과 같은 일반적인 웹 익스플로잇으로부터 AppSync API를 보호할 수 있습니다. 이는 API 가용성 및 성능에 영향을 미치거나, 보안을 손상시키거나, 리소스를 과도하게 소비할 수 있습니다. 예를 들어 지정된 IP 주소 범위의 요청, CIDR 블록의 요청, 특정 국가 또는 리전에서 시작된 요청, 악성 SQL 코드가 포함된 요청, 악성 스크립트가 포함된 요청을 허용하거나 차단하는 규칙을 만들 수 있습니다.

HTTP 헤더, 메서드, 쿼리 문자열, URI 및 요청 본문(처음 8KB로 제한)에 지정된 문자열이나 정규식 패턴과 일치하는 규칙을 만들 수도 있습니다. 또한 특정 사용자 에이전트, 악성 봇, 콘텐츠 스크래퍼의 공격을 차단하는 규칙을 생성할 수 있습니다. 예를 들어, 비율 기반 규칙을 사용하여 연속적으로 업데이트되는 이후 5분 동안 각 클라이언트 IP에서 허용하는 웹 요청 수를 지정할 수 있습니다.

지원되는 규칙 유형과 추가 AWS WAF 기능에 대해 자세히 알아보려면 AWS WAF 개발자 안내서 및 AWS WAF API 참조를 참조하세요.

> **Important**
>
> AWS WAF는 웹 익스플로잇에 대한 방어벽 제1선입니다. API에서 AWS WAF가 활성화된 경우, AWS WAF 규칙은 다른 액세스 제어 기능(API 키 인증, IAM 정책, OIDC 토큰, Amazon Cognito 사용자 풀)보다 먼저 평가됩니다.

AppSync API를 AWS WAF와 통합

AWS Management Console, AWS CLI, AWS CloudFormation 또는 기타 호환 가능한 클라이언트를 사용하여 AppSync API를 AWS WAF와 통합할 수 있습니다.

AWS AppSync API를 AWS WAF와 통합하려면

1. AWS WAF 웹 ACL을 생성합니다. AWS WAF 콘솔 사용의 자세한 단계는 웹 ACL 생성을 참조하세요.
2. 웹 ACL의 규칙을 정의합니다. 규칙은 웹 ACL을 생성하는 과정에서 정의됩니다. 규칙 구성 방법에 대한 자세한 내용은 AWS WAF 규칙을 참조하세요. AWS AppSync API에 정의할 수 있는 유용한 규칙의 예는 웹 ACL에 대한 규칙 생성을 참조하세요.
3. 웹 ACL을 AWS AppSync API와 연결합니다. AWS WAF 콘솔 또는 AppSync 콘솔에서 이 단계를 수행할 수 있습니다.
   - AWS WAF 콘솔에서 웹 ACL을 AWS AppSync API와 연결하려면 AWS WAF 개발자 안내서의 웹 ACL을 AWS 리소스와 연결 또는 연결 해제하기의 지침을 따르세요.
   - AWS AppSync 콘솔에서 웹 ACL을 AWS AppSync API와 연결하려면
     a. AWS Management Console에 로그인하고 AppSync 콘솔을 엽니다.
     b. 웹 ACL과 연결할 API를 선택합니다.
     c. 탐색 창에서 설정(Settings)을 선택합니다.
     d. 웹 애플리케이션 방화벽 섹션에서 AWS WAF 활성화를 켭니다.
     e. 웹 ACL 드롭다운 목록에서 API와 연결할 웹 ACL의 이름을 선택합니다.
     f. 저장을 선택하여 웹 ACL을 API와 연결합니다.

> **Note**
>
> AWS WAF 콘솔에서 웹 ACL을 생성한 후, 새 웹 ACL을 사용할 수 있을 때까지 몇 분 정도 걸릴 수 있습니다. 웹 애플리케이션 방화벽 메뉴에 새로 만든 웹 ACL이 표시되지 않는 경우, 몇 분 정도 기다린 후 단계를 다시 시도하여 웹 ACL을 API와 연결해 봅니다.

> **Note**
>
> AWS WAF 통합은 실시간 엔드포인트에 대한 `Subscription registration message` 이벤트만 지원합니다. AWS AppSync는 AWS WAF에 의해 차단된 `Subscription registration message`에 대해서는 `start_ack` 메시지 대신 오류 메시지로 응답합니다.

웹 ACL을 AWS AppSync API와 연결한 후에는 AWS WAF API를 사용하여 웹 ACL을 관리하게 됩니다. AWS AppSync API를 다른 웹 ACL에 연결하려는 경우가 아니라면 웹 ACL을 AWS AppSync API에 다시 연결할 필요가 없습니다.

웹 ACL에 대한 규칙 생성

규칙은 웹 요청을 검사하는 방법과 웹 요청이 검사 기준과 일치할 때 수행할 작업을 정의합니다. 규칙은 AWS WAF에 자체적으로 존재하지 않습니다. 규칙 그룹 또는 규칙이 정의된 웹 ACL에서 이름으로 규칙에 액세스할 수 있습니다. 자세한 내용은 AWS WAF 규칙을 참조하세요. 다음 예제는 AppSync API를 보호하는 데 유용한 규칙을 정의하고 연결하는 방법을 보여줍니다.

Example 요청 본문 크기를 제한하는 웹 ACL 규칙

다음은 요청의 본문 크기를 제한하는 규칙의 예입니다. 이는 AWS WAF 콘솔에서 웹 ACL을 생성할 때 규칙 JSON 편집기에 입력됩니다.

```
{
    "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
         }
}
```

앞의 예제 규칙을 사용하여 웹 ACL을 생성한 후에는 이를 AppSync API와 연결해야 합니다. AWS Management Console을 사용하는 대신 다음 명령을 실행하여 AWS CLI에서 이 단계를 수행할 수 있습니다.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

변경 사항이 전파되는 데 몇 분 정도 걸릴 수 있지만, 이 명령을 실행한 후에는 1024바이트보다 큰 본문이 포함된 요청이 AWS AppSync에 의해 거부됩니다.

> **Note**
>
> AWS WAF 콘솔에서 새 웹 ACL을 만든 후, API와 연결할 수 있게 되려면 몇 분 정도 걸릴 수 있습니다. CLI 명령을 실행할 때 `WAFUnavailableEntityException` 오류가 발생하면 몇 분 정도 기다렸다가 명령을 다시 실행해 보세요.

Example 단일 IP 주소에서 요청을 제한하는 웹 ACL 규칙

다음은 AppSync API를 단일 IP 주소에서 요청 100개로 제한하는 규칙의 예입니다. 이는 AWS WAF 콘솔에서 속도 기반 규칙으로 웹 ACL을 생성할 때 규칙 JSON 편집기에 입력됩니다.

```
{
  "Name": "Throttle",
  "Priority": 0,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "Throttle"
```

```
  },
  "Statement": {
    "RateBasedStatement": {
      "Limit": 100,
      "AggregateKeyType": "IP"
    }
  }
}
```

앞의 예제 규칙을 사용하여 웹 ACL을 생성한 후에는 이를 AppSync API와 연결해야 합니다. AWS CLI에서 다음 명령을 실행하여 이 단계를 수행할 수 있습니다.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

Example API에 대한 GraphQL ____schema 내부 검사 쿼리를 방지하는 웹 ACL 규칙

다음은 API에 대한 GraphQL __schema 내부 검사 쿼리를 방지하는 규칙의 예입니다. 문자열 '__schema'를 포함하는 모든 HTTP 본문은 차단됩니다. 이는 AWS WAF 콘솔에서 웹 ACL을 생성할 때 규칙 JSON 편집기에 입력됩니다.

```
{
  "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
```

```
        }
      ]
    }
  }
}
```

앞의 예제 규칙을 사용하여 웹 ACL을 생성한 후에는 이를 AppSync API와 연결해야 합니다. AWS CLI에서 다음 명령을 실행하여 이 단계를 수행할 수 있습니다.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

보안 내부 AWS AppSync

클라우드 AWS 보안이 최우선 과제입니다. AWS 고객은 가장 보안에 민감한 조직의 요구 사항을 충족하도록 구축된 데이터 센터 및 네트워크 아키텍처를 활용할 수 있습니다.

보안은 기업과 기업 간의 공동 책임입니다. AWS 공동 책임 모델은 이 사항을 클라우드의 보안 및 클라우드 내 보안으로 설명합니다.

- 클라우드 보안 - AWS 클라우드에서 AWS 서비스를 실행하는 인프라를 보호하는 역할을 합니다. AWS AWS 또한 안전하게 사용할 수 있는 서비스를 제공합니다. 적용되는 규정 준수 프로그램에 대해 자세히 알아보려면 규정 준수 프로그램별 범위 내AWS 서비스 규정 준수 참조하십시오. AWS AppSync
- 클라우드에서의 보안 - 귀하의 책임은 사용하는 AWS 서비스에 따라 결정됩니다. 또한 귀하는 귀사의 데이터의 민감도, 귀사의 요구 사항, 관련 법률 및 규정을 비롯한 기타 요소에 대해서도 책임이 있습니다.

이 설명서는 공동 책임 모델을 사용할 때 공동 책임 모델을 적용하는 방법을 이해하는 데 도움이 AWS AppSync 됩니다. 다음 항목에서는 보안 및 규정 준수 목표를 AWS AppSync 충족하도록 구성하는 방법을 보여줍니다. 또한 AWS AppSync 리소스를 모니터링하고 보호하는 데 도움이 되는 다른 AWS 서비스를 사용하는 방법도 알아봅니다.

주제

- 데이터 보호: AWS AppSync
- 규정 준수 검증: AWS AppSync
- 인프라 보안: AWS AppSync
- 레질리언스: AWS AppSync
- ID 및 액세스 관리 대상 AWS AppSync
- 를 통한 AWS AppSync API 호출 로깅 AWS CloudTrail
- 에 대한 보안 모범 사례 AWS AppSync

데이터 보호: AWS AppSync

AWS 공동 책임 모델 의 데이터 보호에 적용됩니다 AWS AppSync. 이 모델에 설명된 대로 AWS 는 모든 데이터를 실행하는 글로벌 인프라를 보호하는 역할을 AWS 클라우드합니다. 사용자는 인프라에서

호스팅되는 콘텐츠를 관리해야 합니다. 사용하는 AWS 서비스 의 보안 구성과 관리 작업에 대한 책임도 사용자에게 있습니다. 데이터 프라이버시에 대한 자세한 내용은 데이터 프라이버시 FAQ를 참조하세요. 유럽의 데이터 보호에 대한 자세한 내용은 AWS 보안 블로그의 AWS 공동 책임 모델 및 GDPR 블로그 게시물을 참조하세요.

데이터 보호를 위해 AWS 계정 자격 증명을 보호하고 AWS IAM Identity Center OR AWS Identity and Access Management (IAM) 을 사용하여 개별 사용자를 설정하는 것이 좋습니다. 이렇게 하면 개별 사용자에게 자신의 직무를 충실히 이행하는 데 필요한 권한만 부여됩니다. 또한 다음과 같은 방법으로 데이터를 보호하는 것이 좋습니다.

- 각 계정에 멀티 팩터 인증 설정(MFA)을 사용하세요.
- SSL/TLS를 사용하여 리소스와 통신하세요. AWS TLS 1.2는 필수이며 TLS 1.3를 권장합니다.
- 를 사용하여 API 및 사용자 활동 로깅을 설정합니다. AWS CloudTrail
- 포함된 모든 기본 보안 제어와 함께 AWS 암호화 솔루션을 사용하십시오 AWS 서비스.
- Amazon S3에 저장된 민감한 데이터를 검색하고 보호하는 데 도움이 되는 Amazon Macie와 같은 고급 관리형 보안 서비스를 사용하세요.
- 명령줄 인터페이스 또는 API를 AWS 통해 액세스할 때 FIPS 140-2로 검증된 암호화 모듈이 필요한 경우 FIPS 엔드포인트를 사용하십시오. 사용 가능한 FIPS 엔드포인트에 대한 자세한 내용은 FIPS(Federal Information Processing Standard) 140-2를 참조하세요.

고객의 이메일 주소와 같은 기밀 정보나 중요한 정보는 태그나 이름 필드와 같은 자유 양식 필드에 입력하지 않는 것이 좋습니다. 여기에는 콘솔, API AWS AppSync 또는 AWS 서비스 SDK를 사용하거나 다른 방법으로 작업하는 경우가 포함됩니다. AWS CLI AWS 이름에 사용되는 태그 또는 자유 형식 텍스트 필드에 입력하는 모든 데이터는 청구 또는 진단 로그에 사용될 수 있습니다. 외부 서버에 URL을 제공할 때 해당 서버에 대한 요청을 검증하기 위해 보안 인증 정보를 URL에 포함시켜서는 안 됩니다.

작동 중 암호화

AWS AppSync모든 AWS 서비스와 마찬가지로 AWS 게시된 API 및 SDK를 사용할 때 통신에 TLS1.2 이상을 사용합니다.

Amazon DynamoDB와 같은 다른 AWS 서비스와 AWS AppSync 함께 사용하면 전송 중 암호화가 보장됩니다. 달리 명시되지 않는 한 AWS 모든 서비스는 TLS 1.2 이상을 사용하여 서로 통신합니다. Amazon EC2 CloudFront 또는 를 사용하는 리졸버의 경우 TLS (HTTPS) 가 구성되고 안전한지 확인하는 것은 사용자의 책임입니다. Amazon EC2에서 HTTPS를 구성하는 방법에 대한 정보는 Amazon EC2 사용 설명서의 Amazon Linux 2에서 SSL/TLS 구성을 참조하세요. HTTPS를 구성하는 방법

에 대한 자세한 내용은 CloudFront 사용 설명서의 CloudFrontAmazon의 HTTPS를 참조하십시오. CloudFront

규정 준수 검증: AWS AppSync

제3자 감사자는 여러 규정 AWS 준수 프로그램의 AWS AppSync 일환으로 보안 및 규정 준수를 평가합니다. AWS AppSync SOC, PCI, HIPAA/HIPAA BAA, IRAP, C5, ENS High, OSPAR 및 HITRUST CSF 프로그램을 준수합니다.

일반 정보는 AWS 규정 준수 프로그램AWS 보증 프로그램 규정AWS 참조하십시오.

를 사용하여 AWS Artifact타사 감사 보고서를 다운로드할 수 있습니다. 자세한 내용은 의 보고서 https://docs.aws.amazon.com/artifact/latest/ug/downloading-documents.html 참조하십시오 AWS Artifact.

사용 시 규정 준수 AWS 서비스 책임은 데이터의 민감도, 회사의 규정 준수 목표, 관련 법률 및 규정에 따라 결정됩니다. AWS 규정 준수에 도움이 되는 다음 리소스를 제공합니다.

- 보안 및 규정 준수 퀵 스타트 가이드 - 이 배포 가이드에서는 아키텍처 고려 사항을 설명하고 보안 및 규정 준수에 AWS 중점을 둔 기본 환경을 배포하기 위한 단계를 제공합니다.
- Amazon Web Services의 HIPAA 보안 및 규정 준수를 위한 설계 — 이 백서에서는 기업이 HIPAA 적격 애플리케이션을 만드는 AWS 데 사용할 수 있는 방법을 설명합니다.

  > **Note**
  >
  > 모든 AWS 서비스 사람이 HIPAA 자격을 갖춘 것은 아닙니다. 자세한 내용은 HIPAA 적격 서비스 참조를 참조하십시오.

- AWS 규정 준수 리소스AWS — 이 워크북 및 가이드 모음은 해당 산업 및 지역에 적용될 수 있습니다.
- AWS 고객 규정 준수 가이드 — 규정 준수의 관점에서 공동 책임 모델을 이해하십시오. 이 가이드에서는 보안을 유지하기 위한 모범 사례를 AWS 서비스 요약하고 여러 프레임워크 (미국 표준 기술 연구소 (NIST), 결제 카드 산업 보안 표준 위원회 (PCI), 국제 표준화기구 (ISO) 등) 에서 보안 제어에 대한 지침을 매핑합니다.
- AWS Config 개발자 안내서의 규칙을 사용하여 리소스 평가 — 이 AWS Config 서비스는 리소스 구성이 내부 관행, 업계 지침 및 규정을 얼마나 잘 준수하는지 평가합니다.
- AWS Security Hub— 이를 AWS 서비스 통해 내부 AWS보안 상태를 포괄적으로 파악할 수 있습니다. Security Hub는 보안 제어를 사용하여 AWS 리소스를 평가하고 보안 업계 표준 및 모범 사례에

대한 규정 준수를 확인합니다. 지원되는 서비스 및 제어 목록은 Security Hub 제어 참조를 참조하십시오.

- Amazon GuardDuty — 환경에 의심스럽고 악의적인 활동이 있는지 AWS 계정모니터링하여 워크로드, 컨테이너 및 데이터에 대한 잠재적 위협을 AWS 서비스 탐지합니다. GuardDuty 특정 규정 준수 프레임워크에서 요구하는 침입 탐지 요구 사항을 충족하여 PCI DSS와 같은 다양한 규정 준수 요구 사항을 해결하는 데 도움이 될 수 있습니다.
- AWS Audit Manager— 이를 AWS 서비스 통해 AWS 사용량을 지속적으로 감사하여 위험을 관리하고 규정 및 업계 표준을 준수하는 방법을 단순화할 수 있습니다.

인프라 보안: AWS AppSync

관리형 서비스로서 AWS 글로벌 네트워크 보안으로 AWS AppSync 보호됩니다. AWS 보안 서비스 및 인프라 AWS 보호 방법에 대한 자세한 내용은 AWS 클라우드 보안을 참조하십시오. 인프라 보안 모범 사례를 사용하여 AWS 환경을 설계하려면 Security Pillar AWS Well-Architected Framework의 인프라 보호를 참조하십시오.

AWS 게시된 API 호출을 사용하여 네트워크를 통해 AWS AppSync 액세스합니다. 고객은 다음을 지원해야 합니다.

- 전송 계층 보안(TLS) TLS 1.2는 필수이며 TLS 1.3을 권장합니다.
- DHE(Ephemeral Diffie-Hellman) 또는 ECDHE(Elliptic Curve Ephemeral Diffie-Hellman)와 같은 완전 전송 보안(PFS)이 포함된 암호 제품군 Java 7 이상의 최신 시스템은 대부분 이러한 모드를 지원합니다.

또한 요청은 액세스 키 ID 및 IAM 주체와 관련된 비밀 액세스 키를 사용하여 서명해야 합니다. 또는 AWS Security Token Service(AWS STS)를 사용하여 임시 보안 인증을 생성하여 요청에 서명할 수 있습니다.

레질리언스: AWS AppSync

AWS 글로벌 인프라는 AWS 지역 및 가용 영역을 중심으로 구축됩니다. AWS 지역은 물리적으로 분리되고 격리된 여러 가용 영역을 제공하며, 이러한 가용 영역은 지연 시간이 짧고 처리량이 높으며 중복성이 높은 네트워킹으로 연결됩니다. 가용 영역을 사용하면 중단 없이 영역 간에 자동으로 장애 극복 조치가 이루어지는 애플리케이션 및 데이터베이스를 설계하고 운영할 수 있습니다. 가용 영역은 기존의 단일 또는 다중 데이터 센터 인프라보다 가용성, 내결함성, 확장성이 뛰어납니다.

AWS 지역 및 가용 영역에 대한 자세한 내용은 글로벌 인프라를 참조하십시오AWS .

AWS 글로벌 인프라 외에도 템플릿을 사용하여 대부분의 리소스를 정의할 수 있습니다. AWS CloudFormation 템플릿을 사용하여 AWS AppSync 리소스를 선언하는 예를 보려면 AWS 블로그 및 사용 설명서의 AWS AppSync Pipeline Resolver 실제 사용 사례를 참조하십시오. AWS AppSync AWS CloudFormationAWS CloudFormation

ID 및 액세스 관리 대상 AWS AppSync

AWS Identity and Access Management (IAM) 은 관리자가 리소스에 대한 액세스를 안전하게 제어할 수 AWS 서비스 있도록 AWS 도와줍니다. IAM 관리자는 리소스를 사용할 수 있는 인증 (로그인) 및 권한 부여 (권한 보유) 를 받을 수 있는 사용자를 제어합니다. AWS AppSync IAM은 추가 AWS 서비스 비용 없이 사용할 수 있습니다.

주제

- 고객
- ID를 통한 인증
- 정책을 사용한 액세스 관리
- IAM의 AWS AppSync 작동 방식
- AWS AppSync에 대한 ID 기반 정책
- AWS AppSync ID 및 액세스 문제 해결

고객

사용하는 방식 AWS Identity and Access Management (IAM) 은 수행하는 작업에 따라 다릅니다. AWS AppSync

서비스 사용자 - AWS AppSync 서비스를 사용하여 작업을 수행하는 경우 관리자가 필요한 자격 증명과 권한을 제공합니다. 더 많은 AWS AppSync 기능을 사용하여 작업을 수행함에 따라 추가 권한이 필요할 수 있습니다. 액세스 권한 관리 방식을 이해하면 적절한 권한을 관리자에게 요청할 수 있습니다. 에서 AWS AppSync 기능에 액세스할 수 없는 경우 을 참조하십시오AWS AppSync ID 및 액세스 문제 해결.

서비스 관리자 — 회사에서 AWS AppSync 리소스를 담당하고 있다면 전체 액세스 권한이 있을 것입니다 AWS AppSync. 서비스 사용자가 액세스해야 하는 AWS AppSync 기능과 리소스를 결정하는 것은 여러분의 몫입니다. 그런 다음, IAM 관리자에게 요청을 제출하여 서비스 사용자의 권한을 변경해야 합

니다. 이 페이지의 정보를 검토하여 IAM의 기본 개념을 이해하십시오. 회사에서 IAM을 어떻게 사용할 수 있는지 자세히 AWS AppSync 알아보려면 을 참조하십시오IAM의 AWS AppSync 작동 방식.

IAM 관리자 — IAM 관리자라면 액세스 관리를 위한 정책을 작성하는 방법에 대해 자세히 알고 싶을 것입니다. AWS AppSync IAM에서 사용할 수 있는 AWS AppSync ID 기반 정책의 예를 보려면 을 참조하십시오. AWS AppSync에 대한 ID 기반 정책

ID를 통한 인증

인증은 ID 자격 증명을 AWS 사용하여 로그인하는 방법입니다. IAM 사용자로 인증 (로그인 AWS) 하거나 IAM 역할을 맡아 인증 (로그인) 해야 합니다. AWS 계정 루트 사용자

ID 소스를 통해 제공된 자격 증명을 사용하여 페더레이션 ID로 로그인할 수 있습니다. AWS AWS IAM Identity Center (IAM ID 센터) 사용자, 회사의 싱글 사인온 인증, Google 또는 Facebook 자격 증명이 페더레이션 ID의 예입니다. 연동 자격 증명으로 로그인할 때 관리자가 이전에 IAM 역할을 사용하여 ID 페더레이션을 설정했습니다. 페더레이션을 사용하여 액세스하는 경우 AWS 간접적으로 역할을 맡게 됩니다.

사용자 유형에 따라 AWS Management Console 또는 AWS 액세스 포털에 로그인할 수 있습니다. 로그인에 대한 자세한 내용은 AWS 로그인 사용 설명서의 내 로그인 방법을 참조하십시오. AWS AWS 계정

AWS 프로그래밍 방식으로 액세스하는 경우 자격 증명을 사용하여 요청에 암호화 방식으로 서명할 수 있는 소프트웨어 개발 키트 (SDK) 와 명령줄 인터페이스 (CLI) 를 AWS 제공합니다. AWS 도구를 사용하지 않는 경우 요청에 직접 서명해야 합니다. 권장 방법을 사용하여 직접 요청에 서명하는 방법에 대한 자세한 내용은 IAM 사용 설명서의 AWS API 요청 서명을 참조하십시오.

사용하는 인증 방법에 상관없이 추가 보안 정보를 제공해야 할 수도 있습니다. 예를 들어, AWS 계정의 보안을 강화하기 위해 다단계 인증 (MFA) 을 사용할 것을 권장합니다. 자세한 내용은 AWS IAM Identity Center 사용 설명서의 다중 인증 및 IAM 사용 설명서의 AWS에서 다중 인증(MFA) 사용을 참조하세요.

AWS 계정 루트 사용자

계정을 AWS 계정만들 때는 먼저 계정의 모든 AWS 서비스 리소스에 대한 완전한 액세스 권한을 가진 하나의 로그인 ID로 시작합니다. 이 ID를 AWS 계정 루트 사용자라고 하며, 계정을 만들 때 사용한 이메일 주소와 비밀번호로 로그인하여 액세스할 수 있습니다. 일상적인 태스크에 루트 사용자를 사용하지 않을 것을 강력히 권장합니다. 루트 사용자 보안 인증 정보를 보호하고 루트 사용자만 수행할 수 있는 태스크를 수행하는 데 사용하세요. 루트 사용자로 로그인해야 하는 태스크의 전체 목록은 IAM 사용자 안내서의 루트 사용자 보안 인증이 필요한 태스크를 참조하세요.

연동 자격 증명

가장 좋은 방법은 관리자 액세스가 필요한 사용자를 비롯한 수동 AWS 서비스 사용자가 ID 공급자와의 페더레이션을 사용하여 임시 자격 증명을 사용하여 액세스하도록 하는 것입니다.

페더레이션 ID는 기업 사용자 디렉토리, 웹 ID 공급자, Identity Center 디렉터리의 사용자 또는 ID 소스를 통해 제공된 자격 증명을 사용하여 액세스하는 AWS 서비스 모든 사용자를 말합니다. AWS Directory Service페더레이션 ID에 AWS 계정액세스하면 이들이 역할을 맡고 역할은 임시 자격 증명을 제공합니다.

중앙 집중식 액세스 관리를 위해 AWS IAM Identity Center(을)를 사용하는 것이 좋습니다. IAM Identity Center에서 사용자 및 그룹을 생성하거나 자체 ID 소스의 사용자 및 그룹 집합에 연결하고 동기화하여 모든 사용자 및 애플리케이션에서 사용할 수 있습니다. AWS 계정 IAM Identity Center에 대한 자세한 내용은AWS IAM Identity Center 사용 설명서에서 IAM Identity Center란 무엇입니까?를 참조하세요.

IAM 사용자 및 그룹

IAM 사용자는 단일 사용자 또는 애플리케이션에 대한 특정 권한을 AWS 계정 가진 사용자 내 자격 증명입니다. 가능하면 암호 및 액세스 키와 같은 장기 자격 증명이 있는 IAM 사용자를 생성하는 대신 임시 자격 증명을 사용하는 것이 좋습니다. 하지만 IAM 사용자의 장기 자격 증명이 필요한 특정 사용 사례가 있는 경우 액세스 키를 교체하는 것이 좋습니다. 자세한 내용은 IAM 사용 설명서의 장기 보안 인증이 필요한 사용 사례의 경우 정기적으로 액세스 키 교체를 참조하세요.

IAM 그룹은 IAM 사용자 컬렉션을 지정하는 자격 증명입니다. 사용자는 그룹으로 로그인할 수 없습니다. 그룹을 사용하여 여러 사용자의 권한을 한 번에 지정할 수 있습니다. 그룹을 사용하면 대규모 사용자 집합의 권한을 더 쉽게 관리할 수 있습니다. 예를 들어, IAMAdmins라는 그룹이 있고 이 그룹에 IAM 리소스를 관리할 권한을 부여할 수 있습니다.

사용자는 역할과 다릅니다. 사용자는 한 사람 또는 애플리케이션과 고유하게 연결되지만, 역할은 해당 역할이 필요한 사람이라면 누구나 수임할 수 있습니다. 사용자는 영구적인 장기 보안 인증을 가지고 있지만, 역할은 임시 보안 인증만 제공합니다. 자세한 정보는 IAM 사용 설명서의 IAM 사용자를 만들어야 하는 경우(역할이 아님)를 참조하세요.

IAM 역할

IAM 역할은 특정 권한을 가진 사용자 AWS 계정 내의 자격 증명입니다. IAM 사용자와 유사하지만, 특정 개인과 연결되지 않습니다. 역할을 AWS Management Console 전환하여 에서 일시적으로 IAM 역할을 맡을 수 있습니다. AWS CLI 또는 AWS API 작업을 호출하거나 사용자 지정 URL을 사용하여 역할을 수임할 수 있습니다. 역할 사용 방법에 대한 자세한 정보는 IAM 사용 설명서의 IAM 역할 사용을 참조하세요.

임시 보안 인증이 있는 IAM 역할은 다음과 같은 상황에서 유용합니다.

- 페더레이션 사용자 액세스 - 연동 자격 증명에 권한을 부여하려면 역할을 생성하고 해당 역할의 권한을 정의합니다. 연동 자격 증명이 인증되면 역할이 연결되고 역할에 정의된 권한이 부여됩니다. 페더레이션 역할에 대한 자세한 내용은 IAM 사용 설명서의 타사 자격 증명 공급자의 역할 만들기를 참조하세요. IAM Identity Center를 사용하는 경우 권한 세트를 구성합니다. 인증 후 아이덴티티가 액세스할 수 있는 항목을 제어하기 위해 IAM Identity Center는 권한 세트를 IAM의 역할과 연관 짓습니다. 권한 세트에 대한 자세한 내용은 AWS IAM Identity Center 사용 설명서의 권한 세트를 참조하세요.
- 임시 IAM 사용자 권한 - IAM 사용자 또는 역할은 IAM 역할을 수임하여 특정 태스크에 대한 다양한 권한을 임시로 받을 수 있습니다.
- 크로스 계정 액세스 - IAM 역할을 사용하여 다른 계정의 사용자(신뢰할 수 있는 보안 주체)가 내 계정의 리소스에 액세스하도록 허용할 수 있습니다. 역할은 계정 간 액세스를 부여하는 기본적인 방법입니다. 그러나 일부 AWS 서비스경우에는 역할을 프록시로 사용하는 대신 정책을 리소스에 직접 연결할 수 있습니다. 크로스 계정 액세스를 위한 역할과 리소스 기반 정책의 차이점을 알아보려면 IAM 사용 설명서의 IAM 역할과 리소스 기반 정책의 차이를 참조하세요.
- 서비스 간 액세스 — 일부는 다른 AWS 서비스서비스의 기능을 AWS 서비스 사용합니다. 예컨대, 어떤 서비스에서 호출을 수행하면 일반적으로 해당 서비스는 Amazon EC2에서 애플리케이션을 실행하거나 Amazon S3에 객체를 저장합니다. 서비스는 호출하는 보안 주체의 권한을 사용하거나, 서비스 역할을 사용하거나, 또는 서비스 연결 역할을 사용하여 이 작업을 수행할 수 있습니다.
  - 순방향 액세스 세션 (FAS) — IAM 사용자 또는 역할을 사용하여 작업을 수행하는 경우 보안 AWS 주체로 간주됩니다. 일부 서비스를 사용하는 경우 다른 서비스에서 다른 작업을 시작하는 작업을 수행할 수 있습니다. FAS는 전화를 거는 주체의 권한을 다운스트림 AWS 서비스서비스에 AWS 서비스 요청하기 위한 요청과 결합하여 사용합니다. FAS 요청은 다른 서비스 AWS 서비스 또는 리소스와의 상호 작용이 필요한 요청을 서비스가 수신한 경우에만 이루어집니다. 이 경우 두 작업을 모두 수행할 수 있는 권한이 있어야 합니다. FAS 요청 시 정책 세부 정보는 전달 액세스 세션을 참조하세요.
  - 서비스 역할 - 서비스 역할은 서비스가 사용자를 대신하여 태스크를 수행하기 위해 맡는 IAM 역할입니다. IAM 관리자는 IAM 내에서 서비스 역할을 생성, 수정 및 삭제할 수 있습니다. 자세한 정보는 IAM 사용 설명서의 AWS 서비스에 대한 권한을 위임할 역할 생성을 참조하세요.
  - 서비스 연결 역할 — 서비스 연결 역할은 에 연결된 서비스 역할의 한 유형입니다. AWS 서비스서비스는 사용자를 대신하여 작업을 수행하기 위해 역할을 수임할 수 있습니다. 서비스 연결 역할은 사용자에게 AWS 계정 표시되며 해당 서비스가 소유합니다. IAM 관리자는 서비스 링크 역할의 권한을 볼 수 있지만 편집은 할 수 없습니다.
- Amazon EC2에서 실행되는 애플리케이션 — IAM 역할을 사용하여 EC2 인스턴스에서 실행되고 API 요청을 AWS CLI 하는 애플리케이션의 임시 자격 증명을 관리할 수 있습니다. AWS 이는 EC2 인스

턴스 내에 액세스 키를 저장할 때 권장되는 방법입니다. EC2 인스턴스에 AWS 역할을 할당하고 모든 애플리케이션에서 사용할 수 있게 하려면 인스턴스에 연결된 인스턴스 프로필을 생성합니다. 인스턴스 프로파일에는 역할이 포함되어 있으며 EC2 인스턴스에서 실행되는 프로그램이 임시 보안 인증을 얻을 수 있습니다. 자세한 정보는 IAM 사용 설명서의 IAM 역할을 사용하여 Amazon EC2 인스턴스에서 실행되는 애플리케이션에 권한 부여를 참조하세요.

IAM 역할을 사용할지 또는 IAM 사용자를 사용할지를 알아보려면 IAM 사용 설명서의 IAM 역할(사용자 대신)을 생성하는 경우를 참조하세요.

정책을 사용한 액세스 관리

정책을 생성하고 이를 AWS ID 또는 리소스에 AWS 연결하여 액세스를 제어할 수 있습니다. 정책은 ID 또는 리소스와 연결될 때 AWS 해당 권한을 정의하는 객체입니다. AWS 주도자 (사용자, 루트 사용자 또는 역할 세션) 가 요청할 때 이러한 정책을 평가합니다. 정책에서 권한은 요청이 허용되거나 거부되는 지를 결정합니다. 대부분의 정책은 JSON 문서로 AWS 저장됩니다. JSON 정책 문서의 구조와 콘텐츠에 대한 자세한 정보는 IAM 사용 설명서의 JSON 정책 개요를 참조하세요.

관리자는 AWS JSON 정책을 사용하여 누가 무엇에 액세스할 수 있는지 지정할 수 있습니다. 즉, 어떤 보안 주체가 어떤 리소스와 어떤 조건에서 작업을 수행할 수 있는지를 지정할 수 있습니다.

기본적으로, 사용자와 역할에는 어떠한 권한도 없습니다. 사용자에게 사용자가 필요한 리소스에서 작업을 수행할 권한을 부여하려면 IAM 관리자가 IAM 정책을 생성하면 됩니다. 그런 다음 관리자가 IAM 정책을 역할에 추가하고, 사용자가 역할을 수임할 수 있습니다.

IAM 정책은 작업을 수행하기 위해 사용하는 방법과 상관없이 작업에 대한 권한을 정의합니다. 예를 들어, `iam:GetRole`태스크를 허용하는 정책이 있다고 가정합니다. 해당 정책을 사용하는 사용자는 AWS Management Console, AWS CLI, 또는 AWS API에서 역할 정보를 가져올 수 있습니다.

ID 기반 정책

ID 기반 정책은 IAM 사용자, 사용자 그룹 또는 역할과 같은 자격 증명에 연결할 수 있는 JSON 권한 정책 문서입니다. 이러한 정책은 사용자와 역할이 어떤 리소스와 어떤 조건에서 어떤 작업을 수행할 수 있는 지를 제어합니다. 자격 증명 기반 정책을 생성하는 방법을 알아보려면 IAM 사용 설명서의 IAM 정책 생성을 참조하세요.

자격 증명 기반 정책은 인라인 정책 또는 관리형 정책으로 한층 더 분류할 수 있습니다. 인라인 정책은 단일 사용자, 그룹 또는 역할에 직접 포함됩니다. 관리형 정책은 내 여러 사용자, 그룹 및 역할에 연결할 수 있는 독립형 정책입니다. AWS 계정관리형 정책에는 AWS 관리형 정책과 고객 관리형 정책이 포함

됩니다. 관리형 정책 또는 인라인 정책을 선택하는 방법을 알아보려면 IAM 사용 설명서의 관리형 정책과 인라인 정책의 선택을 참조하세요.

리소스 기반 정책

리소스 기반 정책은 리소스에 연결하는 JSON 정책 설명서입니다. 리소스 기반 정책의 예는 IAM 역할 신뢰 정책과 Amazon S3 버킷 정책입니다. 리소스 기반 정책을 지원하는 서비스에서 서비스 관리자는 이러한 정책을 사용하여 특정 리소스에 대한 액세스를 통제할 수 있습니다. 정책이 연결된 리소스의 경우 정책은 지정된 보안 주체가 해당 리소스와 어떤 조건에서 어떤 작업을 수행할 수 있는 지를 정의합니다. 리소스 기반 정책에서 보안 주체를 지정해야 합니다. 보안 주체에는 계정, 사용자, 역할, 연동 사용자 등이 포함될 수 있습니다. AWS 서비스

리소스 기반 정책은 해당 서비스에 있는 인라인 정책입니다. IAM의 AWS 관리형 정책은 리소스 기반 정책에 사용할 수 없습니다.

액세스 제어 목록(ACLs)

액세스 제어 목록(ACL)은 어떤 보안 주체(계정 멤버, 사용자 또는 역할)가 리소스에 액세스할 수 있는 권한을 가지고 있는지를 제어합니다. ACLs는 JSON 정책 문서 형식을 사용하지 않지만 리소스 기반 정책과 유사합니다.

ACL을 지원하는 서비스의 예로는 아마존 S3와 아마존 VPC가 있습니다. AWS WAF ACL에 대해 자세히 알아보려면 Amazon Simple Storage Service 개발자 안내서의 액세스 제어 목록(ACL) 개요를 참조하세요.

기타 정책 타입

AWS 일반적이지 않은 추가 정책 유형을 지원합니다. 이러한 정책 타입은 더 일반적인 정책 타입에 따라 사용자에게 부여되는 최대 권한을 설정할 수 있습니다.

- 권한 경계 – 권한 경계는 보안 인증 기반 정책에 따라 IAM 엔터티(IAM 사용자 또는 역할)에 부여할 수 있는 최대 권한을 설정하는 고급 기능입니다. 개체에 대한 권한 경계를 설정할 수 있습니다. 그 결과로 얻는 권한은 엔터티의 자격 증명 기반 정책과 그 권한 경계의 교집합입니다. `Principal` 필드에서 사용자나 역할을 보안 주체로 지정하는 리소스 기반 정책은 권한 경계를 통해 제한되지 않습니다. 이러한 정책 중 하나에 포함된 명시적 거부는 허용을 재정의합니다. 권한 경계에 대한 자세한 정보는 IAM 사용 설명서의 IAM 엔터티에 대한 권한 경계를 참조하세요.
- 서비스 제어 정책 (SCP) - SCP는 조직 또는 조직 단위 (OU) 에 대한 최대 권한을 지정하는 JSON 정책입니다. AWS Organizations AWS Organizations 사업체가 소유한 여러 AWS 계정 개를 그룹화하고 중앙에서 관리하는 서비스입니다. 조직에서 모든 기능을 활성화할 경우 서비스 제어 정책 (SCP)을 임의의 또는 모든 계정에 적용할 수 있습니다. SCP는 구성원 계정의 엔티티 (각 엔티티 포

함) 에 대한 권한을 제한합니다. AWS 계정 루트 사용자조직 및 SCP에 대한 자세한 정보는 AWS Organizations 사용 설명서의 SCP 작동 방식을 참조하세요.

- 세션 정책 – 세션 정책은 역할 또는 페더레이션 사용자에 대해 임시 세션을 프로그래밍 방식으로 생성할 때 파라미터로 전달하는 고급 정책입니다. 결과적으로 얻는 세션의 권한은 사용자 또는 역할 자격 증명 기반 정책의 교차 및 세션 정책입니다. 또한 권한을 리소스 기반 정책에서 가져올 수도 있습니다. 이러한 정책 중 하나에 포함된 명시적 거부는 허용을 재정의합니다. 자세한 정보는 IAM 사용 설명서의 세션 정책을 참조하세요.

여러 정책 타입

여러 정책 타입이 요청에 적용되는 경우 결과 권한은 이해하기가 더 복잡합니다. 여러 정책 유형이 관련된 경우 요청을 허용할지 여부를 AWS 결정하는 방법을 알아보려면 IAM 사용 설명서의 정책 평가 로직을 참조하십시오.

IAM의 AWS AppSync 작동 방식

IAM을 사용하여 액세스를 AWS AppSync 관리하기 전에 어떤 IAM 기능과 함께 사용할 수 있는지 알아보세요. AWS AppSync

함께 사용할 수 있는 IAM 기능 AWS AppSync

IAM 특성	AWS AppSync 지원
ID 기반 정책	예
리소스 기반 정책	아니요
정책 작업	예
정책 리소스	예
정책 조건 키	아니요
ACL	아니요
ABAC(정책 내 태그)	부분
임시 보안 인증	예

IAM 특성	AWS AppSync 지원
전달 액세스 세션(FAS)	부분
서비스 역할	아니요
서비스 링크 역할	부분

AWS AppSync 및 기타 AWS 서비스가 대부분의 IAM 기능과 어떻게 작동하는지 자세히 알아보려면 IAM 사용 설명서의 IAM과 함께 작동하는AWS 서비스를 참조하십시오.

ID 기반 정책은 다음과 같습니다. AWS AppSync

ID 기반 정책 지원	예

자격 증명 기반 정책은 IAM 사용자, 사용자 그룹 또는 역할과 같은 자격 증명에 연결할 수 있는 JSON 권한 정책 문서입니다. 이러한 정책은 사용자와 역할이 어떤 리소스와 어떤 조건에서 어떤 작업을 수행할 수 있는 지를 제어합니다. 자격 증명 기반 정책을 생성하는 방법을 알아보려면 IAM 사용 설명서의 IAM 정책 생성을 참조하세요.

IAM 자격 증명 기반 정책을 사용하면 허용되거나 거부되는 작업와 리소스뿐 아니라 작업이 허용되거나 거부되는 조건을 지정할 수 있습니다. 자격 증명 기반 정책에서는 보안 주체가 연결된 사용자 또는 역할에 적용되므로 보안 주체를 지정할 수 없습니다. JSON 정책에서 사용하는 모든 요소에 대해 알아보려면 IAM 사용 설명서의 IAM JSON 정책 요소 참조를 참조하세요.

다음에 대한 ID 기반 정책 예제 AWS AppSync

AWS AppSync ID 기반 정책의 예를 보려면 을 참조하십시오. AWS AppSync에 대한 ID 기반 정책

내 리소스 기반 정책 AWS AppSync

리소스 기반 정책 지원	아니요

리소스 기반 정책은 리소스에 연결하는 JSON 정책 문서입니다. 리소스 기반 정책의 예는 IAM 역할 신뢰 정책과 Amazon S3 버킷 정책입니다. 리소스 기반 정책을 지원하는 서비스에서 서비스 관리자는 이

러한 정책을 사용하여 특정 리소스에 대한 액세스를 통제할 수 있습니다. 정책이 연결된 리소스의 경우 정책은 지정된 보안 주체가 해당 리소스와 어떤 조건에서 어떤 작업을 수행할 수 있는 지를 정의합니다. 리소스 기반 정책에서 보안 주체를 지정해야 합니다. 보안 주체에는 계정, 사용자, 역할, 연동 사용자 등이 포함될 수 있습니다. AWS 서비스

계정 간 액세스를 활성화하려는 경우 전체 계정이나 다른 계정의 IAM 엔터티를 리소스 기반 정책의 보안 주체로 지정할 수 있습니다. 리소스 기반 정책에 크로스 계정 보안 주체를 추가하는 것은 트러스트 관계 설정의 절반밖에 되지 않는다는 것을 유념하세요. 보안 주체와 리소스가 다른 AWS 계정경우 신뢰할 수 있는 계정의 IAM 관리자는 보안 주체 개체 (사용자 또는 역할) 에게 리소스에 액세스할 수 있는 권한도 부여해야 합니다. 개체에 자격 증명 기반 정책을 연결하여 권한을 부여합니다. 하지만 리소스 기반 정책이 동일 계정의 보안 주체에 액세스를 부여하는 경우 추가 자격 증명 기반 정책이 필요하지 않습니다. 자세한 정보는 IAM 사용 설명서의 IAM 역할과 리소스 기반 정책의 차이를 참조하세요.

에 대한 정책 조치 AWS AppSync

정책 작업 지원	예

관리자는 AWS JSON 정책을 사용하여 누가 무엇에 액세스할 수 있는지 지정할 수 있습니다. 즉, 어떤 보안 주체가 어떤 리소스와 어떤 조건에서 작업을 수행할 수 있는지를 지정할 수 있습니다.

JSON 정책의 `Action`요소는 정책에서 액세스를 허용하거나 거부하는 데 사용할 수 있는 태스크를 설명합니다. 정책 작업은 일반적으로 관련 AWS API 작업과 이름이 같습니다. 일치하는 API 작업이 없는 권한 전용 작업 같은 몇 가지 예외도 있습니다. 정책에서 여러 작업이 필요한 몇 가지 작업도 있습니다. 이러한 추가 작업을 일컬어 종속 작업이라고 합니다.

연결된 작업을 수행할 수 있는 권한을 부여하기 위한 정책에 작업을 포함하십시오.

AWS AppSync 작업 목록을 보려면 서비스 권한 부여 AWS AppSync 참조에 정의된 작업을 참조하십시오.

정책 조치는 조치 앞에 다음 접두사를 AWS AppSync 사용합니다.

```
appsync
```

단일 문에서 여러 작업을 지정하려면 다음과 같이 쉼표로 구분합니다.

```
"Action": [
      "appsync:action1",
```

```
      "appsync:action2"
        ]
```

AWS AppSync ID 기반 정책의 예를 보려면 을 참조하십시오. AWS AppSync에 대한 ID 기반 정책

에 대한 정책 리소스 AWS AppSync

정책 리소스 지원	예

관리자는 AWS JSON 정책을 사용하여 누가 무엇에 액세스할 수 있는지 지정할 수 있습니다. 즉, 어떤 보안 주체가 어떤 리소스와 어떤 조건에서 작업을 수행할 수 있는지 지정할 수 있습니다.

`Resource` JSON 정책 요소는 작업이 적용되는 하나 이상의 개체를 지정합니다. 문장에는 `Resource`또는 `NotResource`요소가 반드시 추가되어야 합니다. 모범 사례에 따라 Amazon 리소스 이름(ARN)을 사용하여 리소스를 지정합니다. 리소스 수준 권한이라고 하는 특정 리소스 타입을 지원하는 작업에 대해 이 작업을 수행할 수 있습니다.

작업 나열과 같이 리소스 수준 권한을 지원하지 않는 작업의 경우, 와일드카드(*)를 사용하여 해당 문이 모든 리소스에 적용됨을 나타냅니다.

```
"Resource": "*"
```

AWS AppSync 리소스 유형 및 해당 ARN 목록을 보려면 서비스 권한 부여 AWS AppSync 참조에 정의된 리소스를 참조하십시오. 각 리소스의 ARN을 지정할 수 있는 작업에 대해 알아보려면 작업 정의 기준을 참조하십시오. AWS AppSync

AWS AppSync ID 기반 정책의 예를 보려면 을 참조하십시오. AWS AppSync에 대한 ID 기반 정책

에 대한 정책 조건 키 AWS AppSync

서비스별 정책 조건 키 지원	아니요

관리자는 AWS JSON 정책을 사용하여 누가 무엇에 액세스할 수 있는지 지정할 수 있습니다. 즉, 어떤 보안 주체가 어떤 리소스와 어떤 조건에서 작업을 수행할 수 있는지 지정할 수 있습니다.

Condition 요소(또는 Condition 블록)를 사용하면 정책이 발효되는 조건을 지정할 수 있습니다. Condition 요소는 옵션입니다. 같거나 작음과 같은 조건 연산자를 사용하여 정책의 조건을 요청의 값과 일치시키는 조건식을 생성할 수 있습니다.

한 문에서 여러 Condition요소를 지정하거나 단일 Condition요소에서 여러 키를 지정하는 경우 AWS 는 논리적 AND태스크를 사용하여 평가합니다. 단일 조건 키에 여러 값을 지정하는 경우 는 논리적 OR 연산을 사용하여 조건을 AWS 평가합니다. 명문의 권한을 부여하기 전에 모든 조건을 충족해야 합니다.

조건을 지정할 때 자리 표시자 변수를 사용할 수도 있습니다. 예를 들어, IAM 사용자에게 IAM 사용자 이름으로 태그가 지정된 경우에만 리소스에 액세스할 수 있는 권한을 부여할 수 있습니다. 자세한 정보는 IAM 사용 설명서의 IAM 정책 요소: 변수 및 태그를 참조하세요.

AWS 글로벌 조건 키 및 서비스별 조건 키를 지원합니다. 모든 AWS 글로벌 조건 키를 보려면 IAM 사용 AWS 설명서의 글로벌 조건 컨텍스트 키를 참조하십시오.

AWS AppSync 조건 키 목록을 보려면 서비스 권한 부여 참조의 조건 키를 참조하십시오 AWS AppSync. 조건 키를 사용할 수 있는 작업 및 리소스에 대해 알아보려면 작업 정의 기준을 참조하십시오 AWS AppSync.

AWS AppSync ID 기반 정책의 예를 보려면 을 참조하십시오. AWS AppSync에 대한 ID 기반 정책

AWS AppSync의 ACL(액세스 제어 목록)

ACL 지원	아니요

액세스 제어 목록(ACLs)은 어떤 보안 주체(계정 멤버, 사용자 또는 역할)가 리소스에 액세스할 수 있는 권한을 가지고 있는 지를 제어합니다. ACLs는 JSON 정책 문서 형식을 사용하지 않지만 리소스 기반 정책과 유사합니다.

ABAC (속성 기반 액세스 제어) 를 통한 AWS AppSync

ABAC(정책 내 태그) 지원	부분

속성 기반 액세스 제어(ABAC)는 속성을 기반으로 권한을 정의하는 권한 부여 전략입니다. AWS에서는 이러한 속성을 태그라고 합니다. IAM 개체 (사용자 또는 역할) 및 여러 AWS 리소스에 태그를 첨부할 수 있습니다. ABAC의 첫 번째 단계로 개체 및 리소스에 태그를 지정합니다. 그런 다음 보안 주체의 태그가 액세스하려는 리소스의 태그와 일치할 때 작업을 허용하도록 ABAC 정책을 설계합니다.

ABAC는 빠르게 성장하는 환경에서 유용하며 정책 관리가 번거로운 상황에 도움이 됩니다.

태그를 기반으로 액세스를 제어하려면 `aws:ResourceTag/key-name`, `aws:RequestTag/key-name` 또는 `aws:TagKeys` 조건 키를 사용하여 정책의 조건 요소에 태그 정보를 제공합니다.

서비스가 모든 리소스 타입에 대해 세 가지 조건 키를 모두 지원하는 경우, 값은 서비스에 대해 예입니다. 서비스가 일부 리소스 타입에 대해서만 세 가지 조건 키를 모두 지원하는 경우, 값은 부분적입니다.

ABAC에 대한 자세한 정보는 IAM 사용 설명서의 ABAC란 무엇인가요?를 참조하세요. ABAC 설정 단계가 포함된 자습서를 보려면 IAM 사용 설명서의 속성 기반 액세스 제어(ABAC) 사용을 참조하세요.

다음과 같은 임시 자격 증명 사용 AWS AppSync

임시 보안 인증 지원	예

임시 자격 증명을 사용하여 로그인하면 일부 자격 증명이 AWS 서비스 작동하지 않습니다. 임시 자격 증명을 사용하는 방법을 AWS 서비스 비롯한 추가 정보는 IAM 사용 설명서의 IAM과AWS 서비스 연동되는 내용을 참조하십시오.

사용자 이름과 암호를 제외한 다른 방법을 AWS Management Console 사용하여 로그인하면 임시 자격 증명을 사용하는 것입니다. 예를 들어 회사의 SSO (Single Sign-On) 링크를 AWS 사용하여 액세스하는 경우 이 프로세스에서 자동으로 임시 자격 증명을 생성합니다. 또한 콘솔에 사용자로 로그인한 다음 역할을 전환할 때 임시 보안 인증을 자동으로 생성합니다. 역할 전환에 대한 자세한 정보는 IAM 사용 설명서의 역할로 전환(콘솔)을 참조하세요.

또는 API를 사용하여 임시 자격 증명을 수동으로 생성할 수 있습니다 AWS CLI . AWS 그런 다음 해당 임시 자격 증명을 사용하여 액세스할 수 AWS있습니다. AWS 장기 액세스 키를 사용하는 대신 임시 자격 증명을 동적으로 생성할 것을 권장합니다. 자세한 정보는 IAM의 임시 보안 인증 섹션을 참조하세요.

전달 액세스 세션 대상 AWS AppSync

전달 액세스 세션(FAS) 지원	부분

IAM 사용자 또는 역할을 사용하여 작업을 수행하는 AWS경우 사용자는 보안 주체로 간주됩니다. 일부 서비스를 사용하는 경우 다른 서비스에서 다른 작업을 시작하는 작업을 수행할 수 있습니다. FAS는 전화를 거는 주체의 권한을 다운스트림 서비스에 AWS 서비스 요청하라는 요청과 결합하여 사용합니다. AWS 서비스 FAS 요청은 다른 서비스 AWS 서비스 또는 리소스와의 상호 작용이 필요한 요청을 서

비스가 수신한 경우에만 이루어집니다. 이 경우 두 작업을 모두 수행할 수 있는 권한이 있어야 합니다. FAS 요청 시 정책 세부 정보는 전달 액세스 세션을 참조하세요.

AWS AppSync의 서비스 역할

서비스 역할 지원	아니요

서비스 역할은 서비스가 사용자를 대신하여 작업을 수행하기 위해 수임하는 IAM role(IAM 역할)입니다. IAM 관리자는 IAM 내에서 서비스 역할을 생성, 수정 및 삭제할 수 있습니다. 자세한 정보는 IAM 사용 설명서의 AWS 서비스에 대한 권한을 위임할 역할 생성을 참조하세요.

> **Warning**
>
> 서비스 역할의 권한을 변경하면 AWS AppSync 기능이 중단될 수 있습니다. 서비스 역할을 편집하기 위한 지침이 AWS AppSync 제공되는 경우에만 서비스 역할을 편집하십시오.

서비스 연결 역할은 다음과 같습니다. AWS AppSync

서비스 링크 역할 지원	부분

서비스 연결 역할은 에 연결된 서비스 역할 유형입니다. AWS 서비스서비스는 사용자를 대신하여 작업을 수행하기 위해 역할을 수임할 수 있습니다. 서비스 연결 역할은 사용자에게 AWS 계정 표시되며 해당 서비스가 소유합니다. IAM 관리자는 서비스 링크 역할의 권한을 볼 수 있지만 편집은 할 수 없습니다.

서비스 연결 역할 생성 또는 관리에 대한 자세한 내용은 IAM 사용 설명서의 IAM으로 작업하는AWS 서비스 섹션을 참조하십시오. 서비스 연결 역할 열에서 Yes(이)가 포함된 서비스를 테이블에서 찾습니다. 해당 서비스에 대한 서비스 연결 역할 설명서를 보려면 Yes(네) 링크를 선택합니다.

AWS AppSync에 대한 ID 기반 정책

기본적으로 사용자와 역할에는 리소스를 만들거나 수정할 AWS AppSync 권한이 없습니다. 또한 AWS Management Console, AWS Command Line Interface (AWS CLI) 또는 AWS API를 사용하여 작업을 수행할 수 없습니다. 사용자에게 사용자가 필요한 리소스에서 작업을 수행할 권한을 부여하려면 IAM 관리자가 IAM 정책을 생성하면 됩니다. 그런 다음 관리자가 IAM 정책을 역할에 추가하고, 사용자가 역할을 맡을 수 있습니다.

이러한 예제 JSON 정책 문서를 사용하여 IAM ID 기반 정책을 생성하는 방법을 알아보려면 IAM 사용 설명서의 IAM 정책 생성을 참조하세요.

각 리소스 유형의 ARN 형식을 비롯하여 에서 정의한 AWS AppSync 작업 및 리소스 유형에 대한 자세한 내용은 서비스 권한 부여 참조의 작업, 리소스 및 조건 키를 참조하십시오. AWS AppSync

IAM 자격 증명 기반 정책을 생성하고 구성하기 위한 모범 사례를 알아보려면 the section called “IAM 정책 모범 사례” 섹션을 참조하세요.

에 대한 IAM 자격 증명 기반 정책 목록은 을 참조하십시오. AWS AppSync AWS 관리형 정책 대상 AWS AppSync

주제

- AWS AppSync 콘솔 사용
- 사용자가 자신의 고유한 권한을 볼 수 있도록 허용
- 하나의 Amazon S3 버킷에 액세스
- 태그 AWS AppSync 기반 위젯 보기
- AWS 관리형 정책 대상 AWS AppSync

AWS AppSync 콘솔 사용

AWS AppSync 콘솔에 액세스하려면 최소 권한 집합이 있어야 합니다. 이러한 권한을 통해 내 AWS AppSync 리소스의 세부 정보를 나열하고 볼 수 있어야 AWS 계정합니다. 최소 필수 권한보다 더 제한적인 자격 증명 기반 정책을 만들면 콘솔이 해당 정책에 연결된 엔터티(사용자 또는 역할)에 대해 의도대로 작동하지 않습니다.

AWS CLI 또는 AWS API만 호출하는 사용자에게 최소 콘솔 권한을 허용할 필요는 없습니다. 그 대신, 수행하려는 API 작업과 일치하는 작업에만 액세스할 수 있도록 합니다.

IAM 사용자 및 역할이 AWS AppSync 콘솔을 계속 사용할 수 있도록 하려면 엔티티에 AWS AppSync `ConsoleAccess` 또는 `ReadOnly` AWS 관리형 정책도 연결하십시오. 자세한 내용은 IAM 사용 설명서의 사용자에게 권한 추가를 참조하십시오.

사용자가 자신의 고유한 권한을 볼 수 있도록 허용

이 예시는 IAM 사용자가 자신의 사용자 자격 증명에 연결된 인라인 및 관리형 정책을 볼 수 있도록 허용하는 정책을 생성하는 방법을 보여줍니다. 이 정책에는 콘솔에서 또는 API를 사용하여 프로그래밍 방식으로 이 작업을 완료할 수 있는 AWS CLI 권한이 포함됩니다. 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": "*"
        }
    ]
}
```

하나의 Amazon S3 버킷에 액세스

이 예시에서는 AWS 계정의 IAM 사용자에게 Amazon S3 버킷 중 하나에 대한 액세스 권한을 부여하려고 합니다. `examplebucket` 또한 사용자가 객체를 추가, 업데이트 및 삭제하도록 허용하려고 합니다.

이 정책에서는 `s3:PutObject`, `s3:GetObject` 및 `s3:DeleteObject` 권한을 사용자에게 부여할 뿐만 아니라 `s3:ListAllMyBuckets`, `s3:GetBucketLocation` 및 `s3:ListBucket` 권한도 부여합니다. 이러한 권한은 콘솔에 필요한 추가 권한입니다. 또한 콘솔에서 객체를 복사, 자르기 및 붙여넣기를 할 수 있으려면 `s3:PutObjectAcl` 및 `s3:GetObjectAcl` 작업이 필요합니다. 사용자에게 권

한을 부여하고 콘솔을 사용하여 권한을 테스트하는 예제 연습은 예제 연습: 사용자 정책을 사용하여 버킷에 대한 액세스 제어를 참조하세요.

```
{
   "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/*"
      }
   ]
}
```

태그 AWS AppSync ## ## 보기

ID 기반 정책의 조건을 사용하여 태그를 기반으로 AWS AppSync 리소스에 대한 액세스를 제어할 수 있습니다. 이 예제에서는 ## 보기를 허용하는 정책을 생성하는 방법을 보여 줍니다. 하지만 ## 태그

Owner가 해당 사용자의 사용자 이름 값을 가지고 있는 경우에만 권한이 부여됩니다. 이 정책은 콘솔에서 이 태스크를 완료하는 데 필요한 권한도 부여합니다.

```
{
    "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}"}
            }
        }
    ]
}
```

이 정책을 계정의 IAM 사용자에게 연결할 수 있습니다. 라는 사용자가 위젯을 richard-roe 보려고 하면 AWS AppSync ### 또는 태그를 Owner=richard-roe 지정해야 합니다. owner=richard-roe 그렇지 않으면 액세스가 거부됩니다. 조건 키 이름은 대소문자를 구분하지 않기 때문에 조건 태그 키 Owner는 Owner 및 owner 모두와 일치합니다. 자세한 내용은 IAM 사용 설명서의 IAM JSON 정책 요소: 조건을 참조하세요.

AWS 관리형 정책 대상 AWS AppSync

사용자, 그룹 및 역할에 권한을 추가하려면 정책을 직접 작성하는 것보다 AWS 관리형 정책을 사용하는 것이 더 쉽습니다. 팀에 필요한 권한만 제공하는 IAM 고객 관리형 정책을 생성하기 위해서는 시간과 전문 지식이 필요합니다. 빠르게 시작하려면 AWS 관리형 정책을 사용할 수 있습니다. 이 정책은 일반적인 사용 사례를 다루며 사용자의 AWS 계정에서 사용할 수 있습니다. AWS 관리형 정책에 대한 자세한 내용은 IAM 사용 설명서의AWS 관리형 정책을 참조하십시오.

AWS 서비스는 AWS 관리형 정책을 유지 관리하고 업데이트합니다. AWS 관리형 정책에서는 권한을 변경할 수 없습니다. 서비스가 새 기능을 지원하기 위해 AWS 관리형 정책에 권한을 추가하는 경우가 있습니다. 이 타입의 업데이트는 정책이 연결된 모든 보안 인증(사용자, 그룹 및 역할)에 적용됩니다. 서비스는 새 기능이 출시되거나 새 작업을 사용할 수 있게 되면 AWS 관리형 정책을 업데이트할 가능성이 높습니다. 서비스는 AWS 관리형 정책에서 권한을 제거하지 않으므로 정책 업데이트로 인해 기존 권한이 손상되지 않습니다.

또한 여러 서비스에 걸친 작업 기능에 대한 관리형 정책을 AWS 지원합니다. 예를 들어 ReadOnlyAccess AWS 관리형 정책은 모든 AWS 서비스와 리소스에 대한 읽기 전용 액세스를 제공합니다. 서비스가 새 기능을 시작하면 새 작업 및 리소스에 대한 읽기 전용 권한이 AWS 추가됩니다. 직무 정책의 목록과 설명은 IAM 사용 설명서의 직무에 관한AWS 관리형 정책을 참조하세요.

AWS 관리형 정책: AWSAppSyncInvokeFullAccess

AWSAppSyncInvokeFullAccess AWS 관리형 정책을 사용하여 관리자가 콘솔을 통해 또는 독립적으로 AWS AppSync 서비스에 액세스할 수 있도록 허용할 수 있습니다.

AWSAppSyncInvokeFullAccess 정책을 IAM 보안 인증에 연결할 수 있습니다.

권한 세부 정보

이 정책에는 다음 권한이 포함되어 있습니다.

- AWS AppSync— 의 모든 리소스에 대한 전체 관리 액세스를 허용합니다. AWS AppSync

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "appsync:GraphQL",
                "appsync:GetGraphqlApi",
                "appsync:ListGraphqlApis",
                "appsync:ListApiKeys"
            ],
            "Resource": "*"
        }
```

```
    ]
}
```

AWS 관리형 정책: AWSAppSyncSchemaAuthor

`AWSAppSyncSchemaAuthor` AWS 관리형 정책을 사용하여 IAM 사용자가 액세스하여 GraphQL 스키마를 생성, 업데이트 및 쿼리할 수 있도록 허용할 수 있습니다. 사용자가 이러한 권한으로 할 수 있는 작업에 대한 자세한 내용은 GraphQL API 설계 섹션을 참조하세요.

`AWSAppSyncSchemaAuthor` 정책을 IAM 보안 인증에 연결할 수 있습니다.

권한 세부 정보

이 정책에는 다음 권한이 포함되어 있습니다.

- `AWS AppSync` - 다음 작업을 허용합니다.
  - GraphQL 스키마 생성
  - GraphQL 유형, 해석기 및 함수의 생성, 수정, 삭제 허용
  - 요청 및 응답 템플릿 로직 평가
  - 런타임 및 컨텍스트로 코드 평가
  - GraphQL API에 GraphQL 쿼리 전송
  - GraphQL 데이터 검색

```
{
    "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 관리형 정책: AWSAppSyncPushToCloudWatchLogs

AWS AppSync CloudWatchAmazon을 사용하여 GraphQL 요청 문제를 해결하고 최적화하는 데 사용할 수 있는 로그를 생성하여 애플리케이션의 성능을 모니터링합니다. 자세한 정보는 모니터링 및 로깅을 참조하세요.

AWSAppSyncPushToCloudWatchLogs AWS 관리형 정책을 사용하면 IAM 사용자 계정으로 로그를 AWS AppSync 푸시할 수 있습니다. CloudWatch

AWSAppSyncPushToCloudWatchLogs 정책을 IAM 보안 인증에 연결할 수 있습니다.

권한 세부 정보

이 정책에는 다음 권한이 포함되어 있습니다.

- CloudWatch Logs— 지정된 이름으로 로그 그룹과 스트림을 생성할 수 있습니다. AWS AppSync AWS AppSync로그 이벤트를 지정된 로그 스트림으로 푸시합니다.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS 관리형 정책: AWSAppSyncAdministrator

AWSAppSyncAdministrator AWS 관리형 정책을 사용하여 관리자가 AWS 콘솔을 AWS AppSync 제외한 모든 항목에 액세스할 수 있도록 허용하세요.

AWSAppSyncAdministrator를 IAM 엔터티에 연결할 수 있습니다. AWS AppSync 또한 이 정책을 서비스 역할에 연결하여 사용자를 대신하여 작업을 수행할 수 있도록 합니다.

권한 세부 정보

이 정책에는 다음 권한이 포함되어 있습니다.

- AWS AppSync— 의 모든 리소스에 대한 전체 관리 액세스를 허용합니다. AWS AppSync
- IAM - 다음 작업을 허용합니다.
  - 사용자를 대신하여 다른 서비스의 리소스를 AWS AppSync 분석할 수 있는 서비스 연결 역할 생성
  - 서비스 연결 역할 삭제

- 서비스 연결 역할을 다른 AWS 서비스에 전달하여 나중에 역할을 맡고 사용자 대신 작업을 수행할 수 있도록 합니다.

```
{
    "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 관리형 정책: AWSAppSyncServiceRolePolicy

AWSAppSyncServiceRolePolicy AWS 관리형 정책을 사용하여 사용하거나 관리하는 AWS 서비스 및 리소스에 대한 액세스를 허용할 AWS AppSync 수 있습니다.

AWSAppSyncServiceRolePolicy를 IAM 엔터티에 연결할 수 없습니다. 이 정책은 사용자를 AWS AppSync 대신하여 작업을 수행할 수 있는 서비스 연결 역할에 연결됩니다. 자세한 정보는 서비스 연결 역할은 다음과 같습니다. AWS AppSync 을 참조하세요.

권한 세부 정보

이 정책에는 다음 권한이 포함되어 있습니다.

- X-Ray— AWS AppSync 애플리케이션 내에서 이루어진 요청에 대한 데이터를 수집하는 AWS X-Ray 데 사용합니다. 자세한 정보는 AWS X-Ray를 사용하여 추적을 참조하세요.

  이 정책을 통해 다음 작업을 수행할 수 있습니다.
  - 샘플링 규칙 및 결과 검색
  - 추적 데이터를 X-Ray 대몬(daemon)에 전송

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:PutTraceSegments",
                "xray:PutTelemetryRecords",
                "xray:GetSamplingTargets",
                "xray:GetSamplingRules",
                "xray:GetSamplingStatisticSummaries"
```

```
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

AWS 관리형 정책으로AWS AppSync 업데이트

이 서비스가 이러한 변경 사항을 추적하기 시작한 AWS AppSync 이후의 AWS 관리형 정책 업데이트에 대한 세부 정보를 확인하십시오. 이 페이지의 변경 사항에 대한 자동 알림을 받으려면 AWS AppSync 문서 기록 페이지에서 RSS 피드를 구독하십시오.

변경 사항	설명	날짜
AWSAppSyncSchemaAuthor - 기존 정책에 대한 업데이트	사용자가 런타임 및 컨텍스트로 코드를 평가할 수 있도록 `EvaluateCode` 정책 작업을 추가했습니다.	2023년 2월 7일
AWSAppSyncSchemaAuthor - 기존 정책에 대한 업데이트	API에 대한 나열, 가져오기, 생성, 업데이트 및 삭제 기능을 허용하는 정책 작업을 추가했습니다. 사용자가 요청 및 응답 해석기 매핑 템플릿 로직을 평가할 수 있도록 `EvaluateMappingTemplate` 정책 작업을 추가했습니다. 리소스 태깅을 허용하는 정책 작업을 추가했습니다.	2022년 8월 25일
AWS AppSync 변경 내용 추적 시작	AWS AppSync AWS 관리형 정책의 변경 사항 추적을 시작했습니다.	2022년 8월 25일

AWS AppSync ID 및 액세스 문제 해결

다음 정보를 사용하면 및 IAM을 사용할 때 발생할 수 있는 일반적인 문제를 AWS AppSync 진단하고 해결하는 데 도움이 됩니다.

저는 다음과 같은 작업을 수행할 권한이 없습니다. AWS AppSync

작업을 수행할 권한이 없다는 AWS Management Console 메시지가 표시되면 관리자에게 도움을 요청해야 합니다. 관리자는 사용자 이름과 비밀번호를 제공한 사람입니다.

다음 예제 오류는 IAM 사용자 `mateojackson`이 콘솔을 사용하여 가상 *`my-example-widget`* 리소스에 대한 세부 정보를 보려고 하지만 가상 `appsync:`*`GetWidget`* 권한이 없을 때 발생합니다.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 appsync:GetWidget on resource: my-example-widget
```

이 경우 Mateo는 *`my-example-widget`* 작업을 사용하여 `appsync:`*`GetWidget`* 리소스에 액세스하도록 허용하는 정책을 업데이트하라고 관리자에게 요청합니다.

저는 IAM을 수행할 권한이 없습니다. PassRole

작업을 수행할 권한이 없다는 오류가 발생하는 경우 역할을 넘길 수 있도록 정책을 업데이트해야 합니다. `iam:PassRole` AWS AppSync

일부 AWS 서비스 서비스에서는 새 서비스 역할 또는 서비스 연결 역할을 만드는 대신 기존 역할을 해당 서비스에 전달할 수 있습니다. 이렇게 하려면 사용자가 서비스에 역할을 전달할 수 있는 권한을 가지고 있어야 합니다.

다음 예제 오류는 이라는 IAM 사용자가 콘솔을 사용하여 작업을 `marymajor` 수행하려고 할 때 발생합니다. AWS AppSync 하지만 작업을 수행하려면 서비스 역할이 부여한 권한이 서비스에 있어야 합니다. Mary는 서비스에 역할을 전달할 수 있는 권한을 가지고 있지 않습니다.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

이 경우 Mary가 `iam:PassRole`작업을 수행할 수 있도록 Mary의 정책을 업데이트해야 합니다.

도움이 필요하면 AWS 관리자에게 문의하세요. 관리자는 로그인 보안 인증 정보를 제공한 사람입니다.

액세스 키를 보아야 합니다.

IAM 사용자 액세스 키를 생성한 후에는 언제든지 액세스 키 ID를 볼 수 있습니다. 하지만 보안 액세스 키는 다시 볼 수 없습니다. 보안 액세스 키를 잃어버린 경우 새로운 액세스 키 페어를 생성해야 합니다.

액세스 키는 액세스 키 ID(예: `AKIAIOSFODNN7EXAMPLE`)와 보안 액세스 키(예: `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`)의 두 가지 부분으로 구성됩니다. 사용자 이름 및 암호와 같이 액세스 키 ID와 보안 액세스 키를 함께 사용하여 요청을 인증해야 합니다. 사용자 이름과 암호를 관리하는 것처럼 안전하게 액세스 키를 관리합니다.

> **Important**
>
> 정식 사용자 ID를 찾는 데 도움이 되더라도 액세스 키를 타사에 제공하지 마시기 바랍니다. 이렇게 하면 다른 사람에게 내 계정에 대한 영구 액세스 권한을 부여할 수 있습니다 AWS 계정.

액세스 키 페어를 생성할 때는 액세스 키 ID와 보안 액세스 키를 안전한 위치에 저장하라는 메시지가 나타납니다. 보안 액세스 키는 생성할 때만 사용할 수 있습니다. 하지만 보안 액세스 키를 잃어버린 경우 새로운 액세스 키를 IAM 사용자에게 추가해야 합니다. 최대 두 개의 액세스 키를 가질 수 있습니다. 이미 두 개가 있는 경우 새로 생성하려면 먼저 키 페어 하나를 삭제해야 합니다. 지침을 보려면 IAM 사용 설명서의 액세스 키 관리 단원을 참조하십시오.

저는 관리자이며 다른 사람들이 액세스할 수 있도록 허용하고 싶습니다. AWS AppSync

다른 사람이 액세스할 수 있도록 하려면 액세스가 AWS AppSync 필요한 개인 또는 애플리케이션을 위한 IAM 엔티티 (사용자 또는 역할) 를 생성해야 합니다. 다른 사용자들은 해당 엔터티에 대한 보안 인증을 사용해 AWS에 액세스합니다. 그런 다음 올바른 권한을 부여하는 정책을 엔티티에 연결해야 합니다. AWS AppSync

바로 시작하려면 IAM 사용 설명서의 첫 번째 IAM 위임 사용자 및 그룹 생성을 참조하십시오.

내 AWS 계정 외부의 사용자가 내 AWS AppSync 리소스에 액세스할 수 있도록 허용하고 싶습니다.

다른 계정의 사용자 또는 조직 외부의 사람이 리소스에 액세스할 때 사용할 수 있는 역할을 생성할 수 있습니다. 역할을 수임할 신뢰할 수 있는 사람을 지정할 수 있습니다. 리소스 기반 정책 또는 액세스 제어 목록(ACL)을 지원하는 서비스의 경우 이러한 정책을 사용하여 다른 사람에게 리소스에 대한 액세스 권한을 부여할 수 있습니다.

자세히 알아보려면 다음을 참조하세요.

- 이러한 기능의 AWS AppSync 지원 여부를 알아보려면 을 참조하십시오IAM의 AWS AppSync 작동 방식.
- 소유한 리소스에 대한 액세스 권한을 AWS 계정 부여하는 방법을 알아보려면 IAM 사용 설명서에서 자신이 소유한 다른 AWS 계정 IAM 사용자에게 액세스 권한 제공을 참조하십시오.
- 제3자에게 리소스에 대한 액세스 권한을 제공하는 방법을 알아보려면 IAM 사용 설명서의 타사 AWS 계정AWS 계정 소유에 대한 액세스 제공을 참조하십시오.
- ID 페더레이션을 통해 액세스 권한을 제공하는 방법을 알아보려면 IAM 사용 설명서의 외부에서 인증된 사용자에게 액세스 권한 제공(ID 페더레이션)을 참조하세요.
- 크로스 계정 액세스를 위한 역할과 리소스 기반 정책 사용의 차이점을 알아보려면 IAM 사용 설명서의 IAM 역할과 리소스 기반 정책의 차이를 참조하세요.

를 통한 AWS AppSync API 호출 로깅 AWS CloudTrail

AWS AppSync 사용자 AWS CloudTrail, 역할 또는 서비스가 수행한 작업의 기록을 제공하는 AWS 서비스와 통합되어 AWS AppSync 있습니다. CloudTrail API 호출을 AWS AppSync 이벤트로 캡처합니다. 캡처된 호출에는 AWS AppSync 콘솔에서의 호출 및 AWS AppSync API 작업에 대한 코드 호출이 포함됩니다. 트레일을 생성하면 에 대한 이벤트를 포함하여 Amazon S3 버킷으로 CloudTrail 이벤트를 지속적으로 전송할 수 AWS AppSync 있습니다. 트레일을 구성하지 않아도 CloudTrail 콘솔의 이벤트 기록에서 가장 최근 이벤트를 계속 볼 수 있습니다. 에서 수집한 CloudTrail 정보를 사용하여 요청을 받은 사람 AWS AppSync, 요청한 IP 주소, 요청한 사람, 요청 시기 및 추가 세부 정보를 확인할 수 있습니다.

자세한 CloudTrail 내용은 AWS CloudTrail 사용 설명서를 참조하십시오.

AWS AppSync 에 대한 정보 CloudTrail

CloudTrail 계정을 만들 때 AWS 계정에서 활성화됩니다. 에서 AWS AppSync 활동이 발생하면 해당 활동이 CloudTrail 이벤트 기록의 다른 AWS 서비스 이벤트와 함께 이벤트에 기록됩니다. AWS 계정에서 최근 이벤트를 보고, 검색하고, 다운로드할 수 있습니다. 자세한 내용은 이벤트 기록으로 CloudTrail 이벤트 보기를 참조하십시오.

에 대한 이벤트를 포함하여 AWS 계정에서 진행 중인 이벤트의 기록을 보려면 AWS AppSync 트레일을 생성하세요. 트레일을 사용하면 CloudTrail Amazon S3 버킷으로 로그 파일을 전송할 수 있습니다. 기본적으로 콘솔에서 트레일을 생성하면 트레일이 모든 AWS 지역에 적용됩니다. 트레일은 AWS 파티션에 있는 모든 지역의 이벤트를 기록하고 지정한 Amazon S3 버킷으로 로그 파일을 전송합니다. 또한 CloudTrail 로그에서 수집된 이벤트 데이터를 추가로 분석하고 이에 따라 조치를 취하도록 다른 AWS 서비스를 구성할 수 있습니다. 자세한 내용은 다음을 참조하십시오.

- 추적 생성 개요
- CloudTrail 지원되는 서비스 및 통합
- 에 대한 Amazon SNS 알림 구성 CloudTrail
- 여러 지역에서 CloudTrail 로그 파일 수신 및 여러 계정으로부터 CloudTrail 로그 파일 수신

AWS AppSync AWS AppSync API를 통한 통화 로깅을 지원합니다. 현재 API에 대한 호출과 리졸버에 대한 호출은 로그인되지 않습니다. AWS AppSync CloudTrail

모든 이벤트 및 로그 항목에는 요청을 생성한 사용자에 대한 정보가 들어 있습니다. 신원 정보를 이용하면 다음을 쉽게 알아볼 수 있습니다.

- 요청이 루트 또는 AWS Identity and Access Management (IAM) 사용자 자격 증명으로 이루어졌는지 여부.
- 역할 또는 연동 사용자를 위한 임시 보안 인증으로 요청을 생성하였는지.
- 다른 AWS 서비스에서 요청했는지 여부.

자세한 내용은 CloudTrail UserIdentity 요소를 참조하십시오.

로그 파일 항목 이해 AWS AppSync

트레일은 지정한 Amazon S3 버킷에 이벤트를 로그 파일로 전송할 수 있는 구성입니다. CloudTrail 로그 파일에는 하나 이상의 로그 항목이 포함되어 있습니다. 이벤트는 모든 소스의 단일 요청을 나타내며 요청된 작업, 작업 날짜 및 시간, 요청 매개 변수 등에 대한 정보를 포함합니다. CloudTrail 로그 파일은 공개 API 호출의 정렬된 스택 트레이스가 아니므로 특정 순서로 표시되지 않습니다.

다음 예제는 AWS AppSync 콘솔을 통해 `GetGraphqlApi` 수행된 작업을 보여주는 CloudTrail 로그 항목을 보여줍니다.

```
{
    "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"
}
```


다음 예제는 를 통해 CreateApikey 수행된 작업을 보여주는 CloudTrail 로그 항목을 보여줍니다. 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"
}
```

에 대한 보안 모범 사례 AWS AppSync

AWS AppSync 보안은 단순히 몇 가지 레버를 켜거나 로깅을 설정하는 것 이상입니다. 다음 섹션에서는 서비스 사용 방식에 따라 달라지는 보안 모범 사례에 대해 설명합니다.

인증 방법 이해

AWS AppSync GraphQL API에 대해 사용자를 인증하는 다양한 방법을 제공합니다. 각 방법에는 보안, 감사 가능성 및 사용성 측면에서 상충되는 부분이 있습니다.

다음과 같은 인증 방법을 공통으로 사용할 수 있습니다.

- Amazon Cognito 사용자 풀을 사용하면 GraphQL API가 사용자 특성을 세밀한 액세스 제어 및 필터링에 사용할 수 있습니다.
- API 토큰은 수명이 제한되어 있으며 지속적 통합 시스템 및 외부 API와의 통합과 같은 자동화 시스템에 적합합니다.
- AWS Identity and Access Management (IAM) 은 에서 관리하는 내부 애플리케이션에 적합합니다. AWS 계정
- OpenID Connect를 사용하면 OpenID Connect 프로토콜을 통해 액세스를 제어하고 페더레이션할 수 있습니다.

의 인증 및 권한 부여에 AWS AppSync대한 자세한 내용은 을 참조하십시오인증 및 권한 부여.

HTTP 해석기에 TLS 사용

HTTP 해석기를 사용할 경우 가능하면 TLS 보안(HTTPS) 연결을 사용해야 합니다. AWS AppSync 신뢰할 수 있는 TLS 인증서의 전체 목록은 을 참조하십시오. HTTPS 엔드포인트에 대해 AWS AppSync가 인정하는 인증 기관(CA)

가능한 한 권한이 가장 적은 역할 사용

DynamoDB 해석기와 같은 해석기를 사용할 때는 Amazon DynamoDB 테이블 등의 리소스에 대한 가장 제한적인 보기를 제공하는 역할을 사용하세요.

IAM 정책 모범 사례

ID 기반 정책은 누군가가 계정에서 AWS AppSync 리소스를 생성, 액세스 또는 삭제할 수 있는지 여부를 결정합니다. 이 작업으로 인해 AWS 계정에 비용이 발생할 수 있습니다. 자격 증명 기반 정책을 생성하거나 편집할 때는 다음 지침과 권장 사항을 따르십시오.

- AWS 관리형 정책으로 시작하여 최소 권한 권한으로 이동 — 사용자와 워크로드에 권한을 부여하려면 여러 일반적인 사용 사례에 권한을 부여하는 AWS 관리형 정책을 사용하세요. 해당 내용은 에서 사용할 수 있습니다. AWS 계정사용 사례에 맞는 AWS 고객 관리형 정책을 정의하여 권한을 더 줄이는 것이 좋습니다. 자세한 정보는 IAM 사용 설명서의 AWS 관리형 정책 또는 AWS 직무에 대한 관리형 정책을 참조하세요.
- 최소 권한 적용 – IAM 정책을 사용하여 권한을 설정하는 경우 태스크를 수행하는 데 필요한 권한만 부여합니다. 이렇게 하려면 최소 권한으로 알려진 특정 조건에서 특정 리소스에 대해 수행할 수 있는 작업을 정의합니다. IAM을 사용하여 권한을 적용하는 방법에 대한 자세한 정보는 IAM 사용 설명서에 있는 IAM의 정책 및 권한을 참조하세요.

- IAM 정책의 조건을 사용하여 액세스 추가 제한 – 정책에 조건을 추가하여 작업 및 리소스에 대한 액세스를 제한할 수 있습니다. 예를 들어 SSL을 사용하여 모든 요청을 전송해야 한다고 지정하는 정책 조건을 작성할 수 있습니다. 예를 AWS 서비스들어 특정 작업을 통해 서비스 작업을 사용하는 경우 조건을 사용하여 서비스 작업에 대한 액세스 권한을 부여할 수도 AWS CloudFormation있습니다. 자세한 정보는 IAM 사용 설명서의 IAM JSON 정책 요소: 조건을 참조하세요.
- IAM Access Analyzer를 통해 IAM 정책을 검증하여 안전하고 기능적인 권한 보장 – IAM Access Analyzer에서는 IAM 정책 언어(JSON)와 모범 사례가 정책에서 준수되도록 신규 및 기존 정책을 검증합니다. IAM Access Analyzer는 100개 이상의 정책 확인 항목과 실행 가능한 추천을 제공하여 안전하고 기능적인 정책을 작성하도록 돕습니다. 자세한 정보는 IAM 사용 설명서의 IAM Access Analyzer 정책 검증을 참조하세요.
- 멀티 팩터 인증 (MFA) 필요 - IAM 사용자 또는 루트 사용자가 필요한 시나리오가 있는 경우 추가 보안을 위해 AWS 계정 MFA를 활성화하십시오. API 작업을 직접 호출할 때 MFA가 필요하면 정책에 MFA 조건을 추가합니다. 자세한 정보는 IAM 사용 설명서의 MFA 보호 API 액세스 구성을 참조하세요.

IAM의 모범 사례에 대한 자세한 내용은 IAM 사용 설명서의 IAM의 보안 모범 사례를 참조하세요.

해석기 참조(JavaScript)

다음 섹션에서는 APPSYNC_JS 런타임 및 JavaScript 해석기를 설명합니다.

주제

- JavaScript 해석기 개요
- 해석기 컨텍스트 객체 참조
- 해석기 및 함수에 대한 JavaScript 런타임 기능
- DynamoDB용 JavaScript 해석기 함수 참조
- OpenSearch용 JavaScript 해석기 함수 참조
- JavaScript Lambda용 리졸버 함수 참조
- JavaScript 데이터 소스에 대한 EventBridge 리졸버 함수 참조
- None 데이터 소스용 JavaScript 해석기 함수 참조
- JavaScript HTTP에 대한 리졸버 함수 참조
- JavaScript Amazon RDS용 리졸버 함수 레퍼런스

JavaScript 해석기 개요

AWS AppSync에서는 데이터 소스에 대한 작업을 수행하여 GraphQL 요청에 응답할 수 있습니다. 쿼리, 뮤테이션 또는 구독을 실행하려는 각 GraphQL 필드에 대해 해석기를 첨부해야 합니다.

해석기는 GraphQL과 데이터 소스를 연결하는 커넥터입니다. 들어오는 GraphQL 요청을 백엔드 데이터 소스에 대한 지침으로 변환하는 방법과 해당 데이터 소스를 GraphQL 응답으로 변환하는 방법을 AWS AppSync에 알려줍니다. AWS AppSync를 사용하면 JavaScript를 사용하여 해석기를 작성하고 이를 AWS AppSync(APPSYNC_JS) 환경에서 실행할 수 있습니다.

AWS AppSync를 사용하면 파이프라인에서 여러 AWS AppSync 기능으로 구성된 단위 해석기 또는 파이프라인 해석기를 작성할 수 있습니다.

지원되는 런타임 기능

AWS AppSync JavaScript 런타임은 JavaScript 라이브러리, 유틸리티 및 기능의 하위 집합을 제공합니다. APPSYNC_JS 런타임에서 지원되는 전체 기능 목록은 해석기 및 함수에 대한 JavaScript 런타임 기능을 참조하세요.

유닛 해석기

유닛 해석기는 데이터 소스에 대해 실행되는 요청 및 응답 핸들러를 정의하는 코드로 구성됩니다. 요청 핸들러는 컨텍스트 객체를 인수로 사용하고 데이터 소스를 호출하는 데 사용된 요청 페이로드를 반환합니다. 응답 핸들러는 실행된 요청의 결과와 함께 데이터 소스로부터 페이로드를 다시 받습니다. 응답 핸들러는 페이로드를 GraphQL 응답으로 변환하여 GraphQL 필드를 해결합니다. 아래 예제에서 해석기는 DynamoDB 데이터 소스에서 항목을 검색합니다.

```
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;
```

JavaScript 파이프라인 해석기의 구조

파이프라인 해석기는 요청 핸들러 및 응답 핸들러와 함수 목록을 정의하는 코드로 구성됩니다. 각 함수에는 데이터 소스에 대해 실행되는 요청 및 응답 핸들러가 있습니다. 파이프라인 해석기는 함수 목록에 실행을 위임하기 때문에 자신은 어떠한 데이터 소스에도 연결되지 않습니다. 유닛 해석기와 함수는 데이터 원본에 대해 작업을 실행하는 기본 요소입니다.

파이프라인 해석기 요청 핸들러

이 파이프라인 해석기의 요청 핸들러(이전 단계)를 사용하면 정의된 함수를 실행하기 전에 몇 가지 준비 로직을 수행할 수 있습니다.

함수 목록

파이프라인 해석기가 순서대로 실행하는 함수 목록입니다. 파이프라인 해석기 요청 핸들러 평가 결과는 첫 번째 함수에 `ctx.prev.result`로 사용할 수 있습니다. 각 함수 평가 결과는 다음 함수에 `ctx.prev.result`로 사용할 수 있습니다.

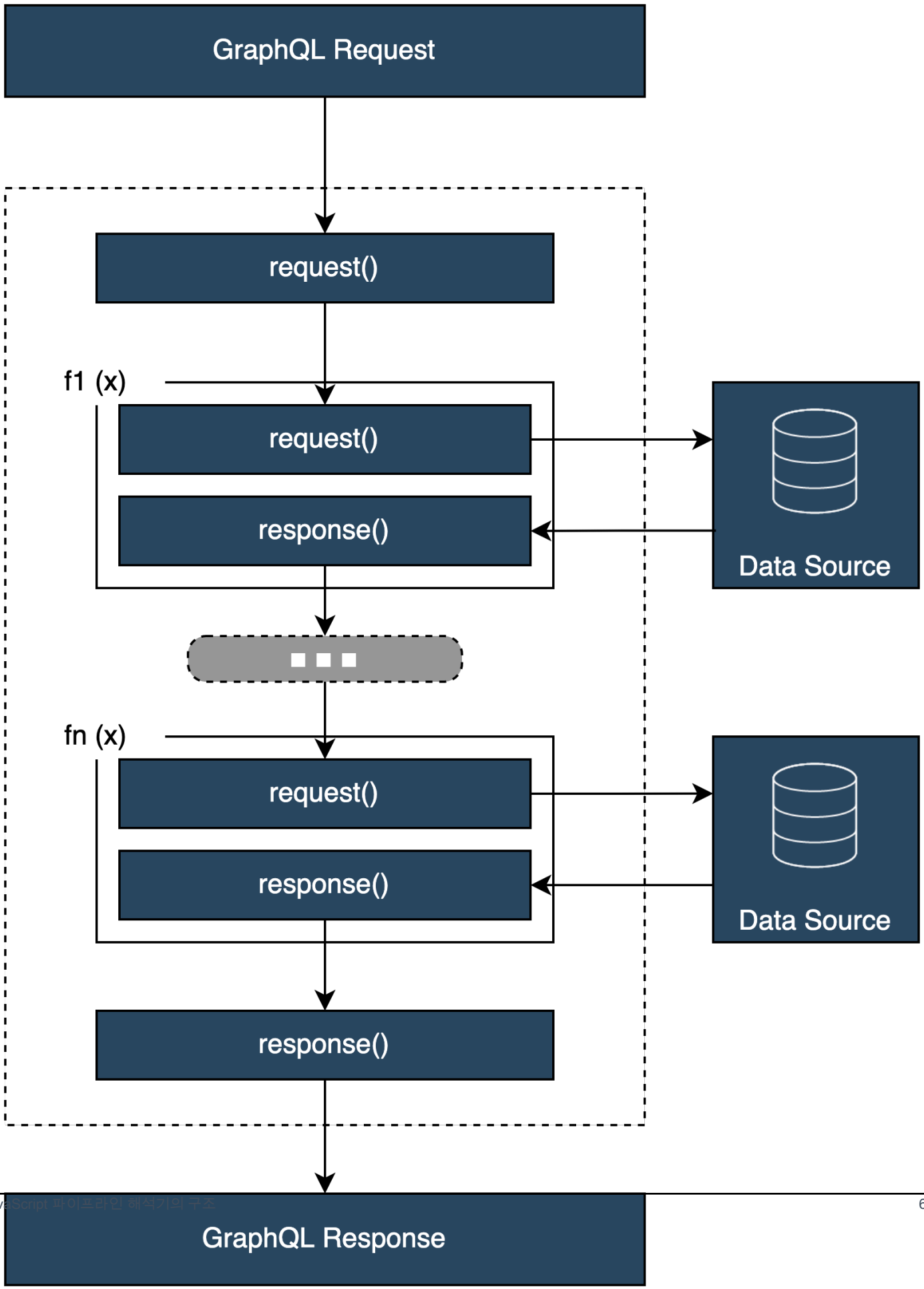
파이프라인 해석기 응답 핸들러

이 파이프라인 해석기의 응답 핸들러를 사용하면 마지막 함수의 출력에서 예상 GraphQL 필드 유형으로 몇 가지 최종 로직을 수행할 수 있습니다. 함수 목록의 마지막 함수 출력은 파이프라인 해석기 응답 핸들러에서 `ctx.prev.result` 또는 `ctx.result`로 사용할 수 있습니다.

실행 흐름

함수 2개로 구성된 파이프라인 해석기의 경우, 아래 목록은 해석기 호출 시 실행 흐름을 나타냅니다.

1. 파이프라인 해석기 요청 핸들러
2. 함수 1: 함수 요청 핸들러
3. 함수 1: 데이터 원본 호출
4. 함수 1: 함수 응답 핸들러
5. 함수 2: 함수 요청 핸들러
6. 함수 2: 데이터 원본 호출
7. 함수 2: 함수 응답 핸들러
8. 파이프라인 해석기 응답 핸들러


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

유용한 APPSYNC_JS 런타임 내장 유틸리티

다음 유틸리티는 파이프라인 해석기로 작업할 때 유용할 수 있습니다.

ctx.stash

stash는 각 해석기 및 함수 요청 및 응답 핸들러 내에서 사용할 수 있는 객체입니다. 단일 해석기가 실행되는 동안에는 동일한 stash 인스턴스가 사용됩니다. 즉, stash를 사용하여 요청 및 응답 핸들러와 파이프라인 해석기의 함수 간에 임의의 데이터를 전달할 수 있습니다. 정규 JavaScript 객체처럼 stash를 테스트할 수 있습니다.

ctx.prev.result

`ctx.prev.result`는 파이프라인에서 실행된 이전 작업의 결과를 나타냅니다. 이전 작업이 파이프라인 해석기 요청 처리기였던 경우, `ctx.prev.result`는 체인의 첫 번째 함수에서 사용할 수 있게 됩니다. 이전 작업이 첫 번째 함수였으면 `ctx.prev.result`는 첫 번째 함수의 결과를 나타내고, 파이프라인의 두 번째 함수에 사용할 수 있습니다. 이전 작업이 마지막 함수였으면 `ctx.prev.result`는 마지막 함수의 결과를 나타내고, 파이프라인 해석기 응답 핸들러에 사용할 수 있습니다.

util.error

`util.error` 유틸리티는 필드 오류를 발생시키는 데 유용합니다. 함수 요청 또는 응답 핸들러 내에서 `util.error`를 사용하면 즉시 필드 오류가 발생해 후속 함수가 실행되지 않도록 방지합니다. 자세한 내용과 기타 `util.error` 서명을 보려면 해석기 및 함수에 대한 JavaScript 런타임 기능을 방문하세요.

util.appendError

`util.appendError`는 `util.error()`와 유사하지만 핸들러 평가를 중단하지 않는다는 큰 차이점이 있습니다. 대신 필드에 오류가 있다는 신호를 보내지만 핸들러가 계속해서 평가되도록 하고 이어서 데이터를 반환합니다. 함수 내에서 `util.appendError`를 사용하면 파이프라인의 실행 흐름이 중단되지 않습니다. 자세한 내용과 기타 `util.error` 서명을 보려면 해석기 및 함수에 대한 JavaScript 런타임 기능을 방문하세요.

runtime.earlyReturn

이 `runtime.earlyReturn` 함수를 사용하면 모든 요청 함수에서 조기에 반환할 수 있습니다. `runtime.earlyReturn`을 해석기 요청 핸들러 내에서 사용하면 해석기에서 반환됩니다. AWS AppSync 함수 요청 핸들러에서 호출하면 함수에서 반환되어 파이프라인의 다음 함수 또는 해석기 응답 핸들러로 계속 실행됩니다.

파이프라인 해석기 작성

파이프라인 해석기에는 파이프라인의 함수 실행을 둘러싼 요청 및 응답 핸들러도 있습니다. 요청 핸들러는 첫 번째 함수 요청 전에 실행되고, 응답 핸들러는 마지막 함수 응답 후에 실행됩니다. 해석기 요청 핸들러는 파이프라인의 함수에서 사용할 데이터를 설정할 수 있습니다. 해석기 응답 핸들러는 GraphQL 필드 출력 유형에 매핑되는 데이터를 반환하는 역할을 합니다. 아래 예에서 해석기 요청 처리기는 `allowedGroups`를 정의합니다. 반환된 데이터는 이러한 그룹 중 하나에 속해야 합니다. 이 값은 해석기의 함수에서 데이터를 요청하는 데 사용할 수 있습니다. 해석기의 응답 핸들러는 최종 검사를 수행하고 결과를 필터링하여 허용된 그룹에 속한 항목만 반환되도록 합니다.

```
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 AppSync 함수 작성

AWS AppSync 함수를 사용하면 스키마의 여러 해석기에서 재사용할 수 있는 공통 로직을 작성할 수 있습니다. 예를 들어 Amazon DynamoDB 데이터 소스에서 항목을 쿼리하는 역할을 하는 `QUERY_ITEMS`라는 AWS AppSync 함수가 하나 있을 수 있습니다. 항목을 쿼리하려는 해석기의 경우

해석기의 파이프라인에 함수를 추가하고 사용할 쿼리 색인을 제공하면 됩니다. 로직을 다시 구현할 필요는 없습니다.

코드 작성

다음 GraphQL 쿼리를 사용하여 Amazon DynamoDB 데이터 소스에서 Post 유형을 반환하는 getPost(id:ID!)라는 필드에 파이프라인 해석기를 연결한다고 가정해 보겠습니다.

```
getPost(id:1){
    id
    title
    content
}
```

먼저 아래 코드를 사용하여 간단한 해석기를 Query.getPost에 연결합니다. 다음은 간단한 해석기 코드의 예입니다. 요청 핸들러에는 로직이 정의되어 있지 않으며, 응답 핸들러는 단순히 마지막 함수의 결과를 반환합니다.

```
/**
 * 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
}
```

다음으로, 데이터 소스에서 사후 항목을 검색하는 함수 GET_ITEM을 정의합니다.

```
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
}
```

요청 중에 오류가 발생하면 함수의 응답 핸들러가 GraphQL 응답에서 호출 클라이언트에 반환될 오류를 추가합니다. GET_ITEM 함수를 해석기 함수 목록에 추가합니다. 쿼리를 실행하면 GET_ITEM 함수의 요청 핸들러는 AWS AppSync의 DynamoDB 모듈에서 제공하는 유틸리티를 사용하여 id를 키로 사용하는 DynamoDBGetItem 요청을 생성합니다. ddb.get({ key: { id } })는 적절한 GetItem 작업을 생성합니다.

```
{
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "1" }
    }
}
```

AWS AppSync는 요청을 사용하여 Amazon DynamoDB에서 데이터를 가져옵니다. 데이터가 반환되면 GET_ITEM 함수의 응답 핸들러에서 처리되며, 이 핸들러는 오류를 확인한 다음 결과를 반환합니다.

```
{
  "result" : {
    "id": 1,
    "title": "hello world",
    "content": "<long story>"
  }
}
```

마지막으로, 해석기의 응답 핸들러는 결과를 직접 반환합니다.

오류 작업

요청 중에 함수에서 오류가 발생하면 `ctx.error`의 함수 응답 핸들러에서 오류를 확인할 수 있습니다. `util.appendError` 유틸리티를 사용하여 GraphQL 응답에 오류를 추가할 수 있습니다. stash를 사용하여 파이프라인의 다른 함수에서 오류를 확인하도록 할 수 있습니다. 아래 예를 참조하세요.

```
/**
 * 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;
}
```

유틸리티

AWS AppSync는 `APPSYNC_JS` 런타임을 사용하여 해석기 개발을 지원하는 두 가지 라이브러리를 제공합니다.

- `@aws-appsync/eslint-plugin` - 개발 중에 문제를 빠르게 발견하고 수정합니다.
- `@aws-appsync/utils` - 코드 편집기에서 유형 검증 및 자동 완성 기능을 제공합니다.

eslint 플러그인 구성

ESLint는 코드를 정적으로 분석하여 문제를 빠르게 찾아내는 도구입니다. 지속적인 통합 파이프라인의 일부로 ESLint를 실행할 수 있습니다. `@aws-appsync/eslint-plugin`은 `APPSYNC_JS` 런타임을 활용할 때 코드에서 잘못된 구문을 찾아내는 ESLint 플러그인입니다. 플러그인을 사용하면 변경 내용을 클라우드로 푸시하지 않고도 개발 중에 코드에 대한 피드백을 빠르게 받을 수 있습니다.

`@aws-appsync/eslint-plugin`은 개발 중에 사용할 수 있는 두 가지 규칙 세트를 제공합니다.

"plugin:@aws-appsync/base"는 프로젝트에서 활용할 수 있는 다음 기본 규칙 세트를 구성합니다.

규칙	설명
no-async	비동기식 프로세스 및 약속은 지원되지 않습니다.
no-await	비동기식 프로세스 및 약속은 지원되지 않습니다.
no-classes	클래스는 지원되지 않습니다.
no-for	`for`는 지원되지 않습니다(`for-in` 및 `for-of`는 지원되는 경우 제외).
no-continue	`continue`는 지원되지 않습니다.
no-generators	생성기는 지원되지 않습니다.
no-yield	`yield`는 지원되지 않습니다.
no-labels	레이블은 지원되지 않습니다.
no-this	`this` 키워드는 지원되지 않습니다.
no-try	Try/catch 구조는 지원되지 않습니다.
no-while	while 루프는 지원되지 않습니다.
no-disallowed-unary-operators	`++`, `--` 및 ~ 단항 연산자는 허용되지 않습니다.
no-disallowed-binary-operators	`instanceof` 연산자는 허용되지 않습니다.

규칙	설명
no-promise	비동기식 프로세스 및 약속은 지원되지 않습니다.

"plugin:@aws-appsync/recommended"는 몇 가지 추가 규칙을 제공하지만 프로젝트에 TypeScript 구성을 추가해야 합니다.

규칙	설명
no-recursion	재귀 함수 호출은 허용되지 않습니다.
no-disallowed-methods	일부 메서드는 허용되지 않습니다. 지원되는 내장 함수 전체 세트는 참조를 참조하세요.
no-function-passing	함수를 함수 인수로 함수에 전달하는 것은 허용되지 않습니다.
no-function-reassign	함수는 재할당할 수 없습니다.
no-function-return	함수는 함수의 반환값이 될 수 없습니다.

프로젝트에 플러그인을 추가하려면 ESLint 시작하기의 설치 및 사용 단계를 따르세요. 그런 다음 프로젝트 패키지 관리자(예: npm, Yarn 또는 pnpm)를 사용하여 프로젝트에 플러그인을 설치합니다.

```
$ npm install @aws-appsync/eslint-plugin
```

`.eslintrc.{js,yml,json}` 파일에서 `extends` 속성에 "plugin:@aws-appsync/base" 또는 "plugin:@aws-appsync/recommended"를 추가합니다. 아래 코드 조각은 JavaScript의 기본 샘플 `.eslintrc` 구성입니다.

```
{
  "extends": ["plugin:@aws-appsync/base"]
}
```

"plugin:@aws-appsync/recommended" 규칙 세트를 사용하려면 필수 종속성을 설치하세요.

```
$ npm install -D @typescript-eslint/parser
```

그런 다음 `.eslintrc.js` 파일을 생성하려면:

```
{
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaVersion": 2018,
    "project": "./tsconfig.json"
  },
  "extends": ["plugin:@aws-appsync/recommended"]
}
```

번들링, TypeScript, 소스 맵

라이브러리 활용 및 코드 번들링

해석기 및 함수 코드에서는 APPSYNC_JS 요구 사항을 준수하는 한 사용자 지정 라이브러리와 외부 라이브러리를 모두 활용할 수 있습니다. 이렇게 하면 애플리케이션에서 기존 코드를 재사용할 수 있습니다. 여러 파일로 정의된 라이브러리를 사용하려면 esbuild와 같은 번들링 도구를 사용하여 코드를 단일 파일로 결합한 다음 이를 AWS AppSync 해석기 또는 함수에 저장해야 합니다.

코드를 번들링할 때 다음 사항에 유의하세요.

- APPSYNC_JS는 ECMAScript 모듈(ESM)만 지원합니다.
- `@aws-appsync/*` 모듈은 APPSYNC_JS에 통합되므로 코드와 함께 번들로 제공해서는 안 됩니다.
- APPSYNC_JS 런타임 환경은 코드가 브라우저 환경에서 실행되지 않는다는 점에서 NodeJS와 유사합니다.
- 선택적 소스 맵을 포함할 수 있습니다. 그러나 소스 콘텐츠는 포함되지 마세요.

  소스 맵에 대한 자세한 내용은 소스 맵 사용을 참조하세요.

예를 들어 `src/appsync/getPost.resolver.js`에 있는 해석기 코드를 번들로 묶으려면 다음 esbuild CLI 명령을 사용하면 됩니다.

```
$ 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
```

코드 작성 및 TypeScript로 작업하기

TypeScript는 Microsoft에서 개발한 프로그래밍 언어로, TypeScript 타이핑 시스템과 함께 JavaScript의 모든 기능을 제공합니다. TypeScript를 사용하여 type-safe 코드를 작성하고 코드를 AWS AppSync에 저장하기 전에 빌드 시 오류와 버그를 잡을 수 있습니다. `@aws-appsync/utils` 패키지가 완전히 입력되었습니다.

`APPSYNC_JS` 런타임은 TypeScript를 직접 지원하지 않습니다. 코드를 AWS AppSync에 저장하기 전에 먼저 TypeScript 코드를 `APPSYNC_JS` 런타임이 지원하는 JavaScript 코드로 변환해야 합니다. TypeScript를 사용하여 로컬 통합 개발 환경(IDE)에서 코드를 작성할 수 있지만 AWS AppSync 콘솔에서는 TypeScript 코드를 생성할 수 없습니다.

시작하려면 프로젝트에 TypeScript가 설치되어 있는지 확인하세요. 그런 다음 TSConfig를 사용하여 `APPSYNC_JS` 런타임에서 작동하도록 TypeScript 트랜스컴파일 설정을 구성합니다. 사용할 수 있는 기본 `tsconfig.json` 파일의 예는 다음과 같습니다.

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "esnext",
    "module": "esnext",
   "noEmit": true,
   "moduleResolution": "node",
  }
}
```

그런 다음 esbuild와 같은 번들링 도구를 사용하여 코드를 컴파일하고 번들링할 수 있습니다. 예를 들어 `src/appsync`에 AWS AppSync 코드가 있는 프로젝트의 경우 다음 명령을 사용하여 코드를 컴파일하고 번들링할 수 있습니다.

```
$ esbuild --bundle \
--sourcemap=inline \
--sources-content=false \
--target=esnext \
--platform=node \
```

```
--format=esm \
--external:@aws-appsync/utils \
--outdir=out/appsync \
 src/appsync/**/*.ts
```

Amplify codegen 사용

Amplify CLI를 사용하여 스키마의 유형을 생성할 수 있습니다. `schema.graphql` 파일이 있는 디렉터리에서 다음 명령을 실행하고 프롬프트를 검토하여 codegen을 구성합니다.

```
$  npx @aws-amplify/cli codegen add
```

특정 상황(예: 스키마 업데이트 시)에서 codegen을 재생성하려면 다음 명령을 실행합니다.

```
$ npx @aws-amplify/cli codegen
```

그런 다음 생성된 유형을 해석기 코드에 사용할 수 있습니다. 예를 들어 다음과 같은 스키마가 있습니다.

```
type Todo {
  id: ID!
  title: String!
  description: String
}

type Mutation {
  createTodo(title: String!, description: String): Todo
}

type Query {
  listTodos: Todo
}
```

다음 예제 AWS AppSync 함수에서 생성된 형식을 사용할 수 있습니다.

```
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
}
```

TypeScript에서 제네릭 사용

제네릭을 제공된 여러 유형과 함께 사용할 수 있습니다. 예를 들어 아래 코드 조각은 Todo 유형입니다.

```
export type Todo = {
  __typename: "Todo",
  id: string,
  title: string,
  description?: string | null,
};
```

Todo를 사용하는 구독용 해석기를 작성할 수 있습니다. IDE에서 유형 정의 및 자동 완성 힌트는 `toSubscriptionFilter` 변환 유틸리티를 올바르게 사용하는 방법을 안내합니다.

```
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
}
```

번들 린팅

`esbuild-plugin-eslint` 플러그인을 가져와서 번들을 자동으로 린트할 수 있습니다. 그런 다음 eslint 기능을 활성화하는 `plugins` 값을 제공하여 활성화할 수 있습니다. 아래는 `build.mjs`라는 파일에서 esbuild JavaScript API를 사용하는 코드 조각입니다.

```
/* 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 })],
})
```

소스 맵 사용

JavaScript 코드로 인라인 소스 맵(sourcemap)을 제공할 수 있습니다. 소스 맵은 JavaScript 또는 TypeScript 코드를 번들로 제공하고 로그 및 런타임 JavaScript 오류 메시지에서 입력 소스 파일에 대한 참조를 확인하려는 경우에 유용합니다.

sourcemap은 코드 끝에 표시되어야 합니다. 이 코드는 다음 형식을 따르는 단일 주석 행으로 정의됩니다.

```
//# sourceMappingURL=data:application/json; base64,<base64 encoded string>
```

다음은 그 예입니다:

```
//# sourceMappingURL=data:application/
json; base64,ewogICJ2ZXJzaW9uIjogMywKICAic291cmNlcyI6IFsibGliLmpzIiwgImNvZGUuanMiXSwKICAibWFwcGl
```

esbuild를 사용하여 소스 맵을 생성할 수 있습니다. 아래 예제는 코드를 빌드하고 번들링할 때 esbuild JavaScript API를 사용하여 인라인 소스 맵을 포함하는 방법을 보여줍니다.

```
/* 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 })],
})
```

특히 sourcemap 및 sourcesContent 옵션은 각 빌드가 끝날 때마다 소스 맵을 줄지어 추가하되 소스 콘텐츠는 포함하지 않도록 지정합니다. 규칙에 따라 소스 콘텐츠는 sourcemap에 포함하지 않는 것이 좋습니다. sources-content를 false로 설정하여 esbuild에서 이를 비활성화할 수 있습니다.

소스 맵의 작동 방식을 설명하려면 해석기 코드가 도우미 라이브러리의 도우미 함수를 참조하는 다음 예제를 검토하세요. 코드에는 해석기 코드와 도우미 라이브러리의 로그 명령문이 포함되어 있습니다.

./src/default.resolver.ts(사용자의 해석기)

```
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(도우미 파일)

```
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..')
}
```

해석기 파일을 빌드하고 번들링하면 해석기 코드에 인라인 소스 맵이 포함됩니다. 해석기가 실행되면 CloudWatch 로그에 다음과 같은 항목이 나타납니다.

```
INFO - ../src/default.resolver.ts:5:2: "start >"
INFO - ../src/helper.ts:3:2: "[logger]" "<hello world>" "<42>" "<true>"
INFO - ../src/default.resolver.ts:7:2: "< end"
{"logType":"BeforeRequestFunctionEvaluation","path":["logstuff"],"fieldName":"logstuff","resolverArn":"arn:aws:
INFO - ../src/helper.ts:8:2: "i just say hello.."
{"logType":"AfterResponseFunctionEvaluation","path":["logstuff"],"fieldName":"logstuff","resolverArn":"arn:aws:
```

CloudWatch 로그의 항목을 살펴보면 두 파일의 기능이 함께 번들로 묶여 동시에 실행되고 있음을 알 수 있습니다. 각 파일의 원본 파일 이름도 로그에 명확하게 반영됩니다.

테스트

코드를 해석기 또는 함수에 저장하기 전에 `EvaluateCode` API 명령을 사용하여 모의 데이터로 해석기 및 함수 핸들러를 원격으로 테스트할 수 있습니다. 명령어를 시작하려면 정책에 `appsync:evaluatecode` 권한을 추가했는지 확인하세요. 예:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateCode",
            "Resource": "arn:aws:appsync:<region>:<account>:*"
        }
    ]
}
```

AWSCLI 또는 AWS SDK를 사용하여 명령을 활용할 수 있습니다. 예를 들어 CLI를 사용하여 코드를 테스트하려면 파일을 가리키고, 컨텍스트를 제공하고, 평가할 핸들러를 지정하기만 하면 됩니다.

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

응답에는 핸들러가 반환한 페이로드가 포함된 evaluationResult가 포함되어 있습니다. 또한 평가 중에 핸들러가 생성한 로그 목록이 들어 있는 logs 객체도 포함되어 있습니다. 이렇게 하면 코드 실행을 쉽게 디버깅하고 평가에 대한 정보를 확인하여 문제를 해결하는 데 도움이 됩니다. 예:

```
{
    "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\""
    ]
}
```

평가 결과를 JSON으로 구문 분석하여 다음과 같은 결과를 얻을 수 있습니다.

```
{
  "operation": "PutItem",
  "key": {
    "id": {
      "S": "record-id"
    }
  },
  "attributeValues": {
    "owner": {
      "S": "John doe"
    },
    "expectedVersion": {
      "N": 2
    },
    "authorId": {
      "S": "Sammy Davis"
    }
  }
}
```

SDK를 사용하면 테스트 제품군의 테스트를 쉽게 통합하여 코드 동작을 검증할 수 있습니다. 이 예제에서는 Jest 테스팅 프레임워크를 사용하지만 어떤 테스트 제품군이든 사용할 수 있습니다. 다음 코드 조각은 가상의 검증 실행을 보여줍니다. 평가 응답은 유효한 JSON일 것으로 예상하므로 `JSON.parse`를 사용하여 문자열 응답에서 JSON을 검색합니다.

```
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)
})
```

이 결과는 다음과 같아야 합니다.

```
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
```

VTL에서 JavaScript로 마이그레이션

AWS AppSync를 사용하면 VTL 또는 JavaScript를 사용하여 해석기 및 함수에 대한 비즈니스 로직을 작성할 수 있습니다. 두 언어를 모두 사용하여 AWS AppSync 서비스에 데이터 소스와 상호 작용하는 방법을 지시하는 로직을 작성합니다. VTL을 사용하면 유효한 JSON으로 인코딩된 문자열로 평가되어야 하는 매핑 템플릿을 작성할 수 있습니다. JavaScript를 사용하면 객체를 반환하는 요청 핸들러 및 응답 핸들러를 작성할 수 있습니다. JSON으로 인코딩된 문자열은 반환하지 않습니다.

예를 들어, 다음 VTL 매핑 템플릿을 사용하여 Amazon DynamoDB 항목을 가져옵니다.

```
{
    "operation": "GetItem",
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
    }
}
```

유틸리티 `$util.dynamodb.toDynamoDBJson`은 JSON으로 인코딩된 문자열을 반환합니다. `$ctx.args.id`가 <id>로 설정된 경우 템플릿은 유효한 JSON 인코딩 문자열로 평가됩니다.

```
{
    "operation": "GetItem",
    "key": {
        "id": {"S": "<id>"},
    }
}
```

JavaScript로 작업할 때 코드 내에서 원시 JSON 인코딩 문자열을 인쇄할 필요가 없으며 `toDynamoDBJson`과 같은 유틸리티를 사용할 필요가 없습니다. 위의 매핑 템플릿과 동일한 예제는 다음과 같습니다.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: {id: util.dynamodb.toDynamoDB(ctx.args.id)}
  };
}
```

대안은 값 객체를 처리하는 데 권장되는 접근 방식인 `util.dynamodb.toMapValues`를 사용하는 것입니다.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}
```

이 결과는 다음과 같이 평가됩니다.

```
{
  "operation": "GetItem",
  "key": {
    "id": {
      "S": "<id>"
    }
  }
}
```

> **Note**
>
> DynamoDB 모듈을 DynamoDB 데이터 소스와 함께 사용하는 것이 좋습니다.
>
> ```
> import * as ddb from '@aws-appsync/utils/dynamodb'
>
> export function request(ctx) {
>  ddb.get({ key: { id: ctx.args.id } })
> }
> ```

또 다른 예로, 다음 매핑 템플릿을 사용하여 Amazon DynamoDB 데이터 소스에 항목을 입력합니다.

```
{
    "operation" : "PutItem",
    "key" : {
        "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

평가 시 이 매핑 템플릿 문자열은 유효한 JSON 인코딩 문자열을 생성해야 합니다. JavaScript를 사용하는 경우 코드는 요청 객체를 직접 반환합니다.

```
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),
  };
}
```

이는 다음과 같이 평가됩니다.

```
{
  "operation": "PutItem",
  "key": {
    "id": { "S": "2bff3f05-ff8c-4ed8-92b4-767e29fc4e63" }
  },
  "attributeValues": {
    "firstname": { "S": "Shaggy" },
    "age": { "N": 4 }
  }
}
```

Note

DynamoDB 모듈을 DynamoDB 데이터 소스와 함께 사용하는 것이 좋습니다.

```
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 })
}
```

직접 데이터 소스 액세스와 Lambda 데이터 소스를 통한 프록시 중 선택

AWS AppSync 및 APPSYNC_JS 런타임을 사용하면 AWS AppSync 함수를 사용하여 데이터 소스에 액세스함으로써 사용자 지정 비즈니스 로직을 구현하는 코드를 직접 작성할 수 있습니다. 따라서 추가 계산 서비스 또는 인프라를 배포하지 않고도 Amazon DynamoDB, Aurora Serverless, OpenSearch Service, HTTP API 및 기타 AWS 서비스와 같은 데이터 소스와 쉽게 직접 상호 작용할 수 있습니다. 또한 AWS AppSync를 사용하면 Lambda 데이터 소스를 구성하여 AWS Lambda 함수와 쉽게 상호 작용할 수 있습니다. Lambda 데이터 소스를 사용하면 AWS Lambda의 전체 기능을 사용하여 복잡한 비즈니스 로직을 실행하여 GraphQL 요청을 해결할 수 있습니다. 대부분의 경우, 필요한 모든 기능은 대상 데이터 소스에 직접 연결된 AWS AppSync 함수가 제공합니다. APPSYNC_JS 런타임에서 지원하지 않

는 복잡한 비즈니스 로직을 구현해야 하는 상황에서는 Lambda 데이터 소스를 프록시로 사용하여 대상 데이터 소스와 상호 작용할 수 있습니다.

	직접 데이터 소스 통합	프록시로서의 Lambda 데이터 소스
사용 사례	AWS AppSync functions interact directly with API data sources.	AWS AppSync functions call Lambdas that interact with API data sources.
Runtime	*APPSYNC_JS* (JavaScript)	지원되는 Lambda 런타임
Maximum size of code	AWS AppSync 함수당 32,000자	Lambda당 50MB(직접 업로드용 압축 파일)
External modules	제한적 - APPSYNC_JS는 기능만 지원	예
Call any AWS service	예 - AWS AppSync HTTP 데이터 소스 사용	예 - AWS SDK 사용
Access to the request header	예	예
Network access	아니요	예
File system access	아니요	예
Logging and metrics	예	예
Build and test entirely within AppSync	예	아니요
Cold start	아니요	아니요 - 프로비저닝된 동시성
Auto-scaling	예 - AWS AppSync에 의해 투명하게	예 - Lambda에서 구성한 대로
Pricing	추가 요금 없음	Lambda 사용에 대한 요금 부과

대상 데이터 소스와 직접 통합되는 AWS AppSync 기능은 다음과 같은 사용 사례에 이상적입니다.

- Amazon DynamoDB, Aurora Serverless 및 OpenSearch Service와의 상호 작용
- HTTP API와의 상호 작용 및 수신 헤더 전달
- HTTP 데이터 소스를 사용하여 AWS 서비스와 상호 작용(제공된 데이터 소스 역할을 사용하여 요청에 자동으로 서명하는 AWS AppSync 사용)
- 데이터 소스에 액세스하기 전에 액세스 제어 구현
- 요청을 이행하기 전에 검색된 데이터에 대한 필터링 구현
- 해석기 파이프라인에서 AWS AppSync 함수를 순차적으로 실행하여 간단한 오케스트레이션 구현
- 쿼리와 뮤테이션의 캐싱 및 구독 연결 제어

Lambda 데이터 소스를 프록시로 사용하는 AWS AppSync 기능은 다음과 같은 사용 사례에 이상적입니다.

- JavaScript 또는 Velocity Template Language(VTL) 이외의 언어 사용
- CPU 또는 메모리를 조정 및 제어하여 성능 최적화
- 타사 라이브러리를 가져오거나 APPSYNC_JS에서 지원되지 않는 기능을 요구
- 여러 네트워크 요청 및/또는 쿼리 수행을 위한 파일 시스템 액세스 권한 얻기
- 일괄 처리 구성을 사용한 요청 일괄 처리.

해석기 컨텍스트 객체 참조

AWS AppSync는 요청 및 응답 핸들러 작업을 위한 변수 및 함수 세트를 정의합니다. 이렇게 하면 GraphQL을 사용하는 데이터에 대한 논리적 조작이 더 쉬워집니다. 이 문서에서는 이러한 함수를 설명하고 예를 제공합니다.

`context`에 액세스

요청 및 응답 핸들러의 `context` 인수는 해석기 호출에 대한 모든 컨텍스트 정보를 포함하는 객체입니다. 이 변수의 구조는 다음과 같습니다.

```
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**
>
> context 객체를 ctx라고 부르는 경우가 종종 있습니다.

context 객체의 각 필드는 다음과 같이 정의됩니다.

context 필드

arguments

이 필드에 대한 모든 GraphQL 인수를 포함하는 맵입니다.

identity

호출자에 대한 정보를 포함하는 객체입니다. 이 필드의 구조에 대한 자세한 내용은 자격 증명을 참조하십시오.

source

상위 필드의 해결 방법을 포함하는 맵입니다.

stash

stash는 각 해석기 및 함수 핸들러 내에서 사용할 수 있는 객체입니다. 단일 해석기가 실행되는 동안에는 동일한 stash 객체가 사용됩니다. 즉, stash를 사용하여 요청 및 응답 핸들러와 파이프라인 해석기의 함수 간에 임의의 데이터를 전달할 수 있습니다.

> **Note**
>
> 전체 stash를 삭제하거나 바꿀 수는 없지만 stash의 속성을 추가, 업데이트, 삭제 및 읽을 수는 있습니다.

아래 코드 예제 중 하나를 수정하여 stash에 항목을 추가할 수 있습니다.

```
//Example 1
ctx.stash.newItem = { key: "something" }

//Example 2
Object.assign(ctx.stash, {key1: value1, key2: value})
```

아래 코드를 수정하여 stash에서 항목을 제거할 수 있습니다.

```
delete ctx.stash.key
```

result

해석기의 결과를 포함하고 있는 컨테이너입니다. 이 필드는 응답 핸들러만 사용할 수 있습니다.

예를 들어, 다음 쿼리의 author 필드를 해석하는 경우:

```
query {
    getPost(id: 1234) {
        postId
        title
        content
        author {
            id
            name
        }
    }
}
```

그러면 응답 핸들러를 평가할 때 전체 context 변수를 사용할 수 있습니다.

```
{
  "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

파이프라인 해석기에서 실행된 이전 작업의 결과입니다.

이전 작업이 파이프라인 해석기의 요청 처리기였던 경우 `ctx.prev.result`는 해당 평가 결과를 나타내며 파이프라인의 첫 번째 함수에서 사용할 수 있게 됩니다.

이전 작업이 첫 번째 함수인 경우 `ctx.prev.result`는 첫 번째 함수 응답 핸들러의 평가 결과를 나타내며 파이프라인의 두 번째 함수에서 사용할 수 있게 됩니다.

이전 작업이 마지막 함수인 경우 `ctx.prev.result`는 마지막 함수의 평가 결과를 나타내며 파이프라인 해석기의 응답 핸들러에서 사용할 수 있게 됩니다.

info

GraphQL 요청에 대한 정보가 포함된 객체입니다. 이 필드의 구조는 정보를 참조하십시오.

Identity

`identity` 섹션에는 호출자에 대한 정보가 포함되어 있습니다. 이 섹션의 모양은 AWS AppSync API의 권한 부여 유형에 따라 달라집니다.

AWS AppSync 보안 옵션에 대한 자세한 내용은 권한 부여 및 인증을 참조하세요.

API_KEY 권한 부여

`identity` 필드는 채워져 있지 않습니다.

AWS_LAMBDA 권한 부여

`identity`의 형식은 다음과 같습니다.

```
type AppSyncIdentityLambda = {
  resolverContext: any;
};
```

identity에는 요청을 승인하는 Lambda 함수가 반환한 동일한 resolverContext 콘텐츠가 포함된 resolverContext 키가 포함되어 있습니다.

AWS_IAM 권한 부여

identity의 형식은 다음과 같습니다.

```
type AppSyncIdentityIAM = {
  accountId: string;
  cognitoIdentityPoolId: string;
  cognitoIdentityId: string;
  sourceIp: string[];
  username: string;
  userArn: string;
  cognitoIdentityAuthType: string;
  cognitoIdentityAuthProvider: string;
};
```

AMAZON_COGNITO_USER_POOLS 권한 부여

identity의 형식은 다음과 같습니다.

```
type AppSyncIdentityCognito = {
  sourceIp: string[];
  username: string;
  groups: string[] | null;
  sub: string;
  issuer: string;
  claims: any;
  defaultAuthStrategy: string;
};
```

각 필드는 다음과 같이 정의됩니다.

accountId

호출자의 AWS 계정 ID입니다.

claims

사용자의 클레임입니다.

cognitoIdentityAuthType

자격 증명 유형을 기반으로 인증되거나 인증되지 않습니다.

cognitoIdentityAuthProvider

요청에 서명하는 데 사용되는 보안 인증 정보를 얻는 데 사용되는 외부 자격 증명 공급자 정보를 쉼표로 구분한 목록입니다.

cognitoIdentityId

호출자의 Amazon Cognito 자격 증명 ID입니다.

cognitoIdentityPoolId

호출자와 연결된 Amazon Cognito 자격 증명 풀 ID입니다.

defaultAuthStrategy

이 호출자의 기본 권한 부여 전략(`ALLOW` 또는 `DENY`)입니다.

issuer

토큰 발행자입니다.

sourceIp

AWS AppSync를 수신하는 호출자의 소스 IP 주소입니다. 요청에 `x-forwarded-for` 헤더가 포함되어 있지 않으면 소스 IP 값에 TCP 연결의 단일 IP 주소만 포함됩니다. 요청에 `x-forwarded-for` 헤더가 포함되어 있으면 소스 IP는 TCP 연결의 IP 주소 이외에 `x-forwarded-for` 헤더의 IP 주소 목록입니다.

sub

인증된 사용자의 UUID입니다.

user

IAM 사용자입니다.

userArn

IAM 사용자의 Amazon 리소스 이름(ARN)입니다.

username

인증된 사용자의 사용자 이름입니다. AMAZON_COGNITO_USER_POOLS 권한 부여 시 username의 값은 cognito:username 속성의 값입니다. AWS_IAM 권한 부여 시 username의 값은 AWS 사용자 보안 주체의 값입니다. Amazon Cognito 자격 증명 풀에서 판매된 자격 증명으로 IAM 인증을 사용하는 경우 cognitoIdentityId를 사용하는 것이 좋습니다.

요청 헤더에 액세스

AWS AppSync에서는 클라이언트에서 사용자 지정 헤더를 전달하고, GraphQL 해석기에서 ctx.request.headers를 사용하여 이러한 헤더에 액세스하는 것을 지원합니다. 그런 다음 데이터 소스에 데이터 삽입 또는 권한 부여 검사 등과 같은 작업에 헤더 값을 사용할 수 있습니다. 다음 예에 표시된 것처럼 명령줄에서 API 키와 함께 $curl을 사용하여 단일 또는 다중 요청 헤더를 사용할 수 있습니다.

단일 헤더의 예

다음과 같이 값이 nadia인 custom의 헤더를 설정한다고 가정해 보겠습니다.

```
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
```

그러면 ctx.request.headers.custom을 사용하여 이 헤더에 액세스할 수 있습니다. 예를 들어, DynamoDB에 대한 다음 코드에 있을 수 있습니다.

```
"custom": util.dynamodb.toDynamoDB(ctx.request.headers.custom)
```

다중 헤더의 예

또한 단일 요청에서 여러 헤더를 전달하고 해석기 핸들러에서 이러한 헤더에 액세스할 수 있습니다. 예를 들어, custom 헤더가 다음 두 값으로 설정되어 있는 경우:

```
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
```

이러한 두 값에는 배열(예: ctx.request.headers.custom[1])로 액세스할 수 있습니다.

> **Note**
>
> AWS AppSync는 `ctx.request.headers`의 쿠키 헤더를 노출하지 않습니다.

사용자 지정 도메인 이름 요청 액세스

AWS AppSync는 GraphQL 및 API의 실시간 엔드포인트에 액세스하는 데 사용할 수 있는 사용자 지정 도메인 구성을 지원합니다. 사용자 지정 도메인 이름으로 요청하는 경우 `ctx.request.domainName`을 사용하여 도메인 이름을 가져올 수 있습니다.

기본 GraphQL 엔드포인트 도메인 이름을 사용하는 경우 값은 `null`입니다.

Info

이 `info` 섹션에는 GraphQL 요청에 대한 정보가 포함되어 있습니다. 이 섹션은 다음과 같은 형식으로 되어 있습니다.

```
type Info = {
  fieldName: string;
  parentTypeName: string;
  variables: any;
  selectionSetList: string[];
  selectionSetGraphQL: string;
};
```

각 필드는 다음과 같이 정의됩니다.

`fieldName`

현재 확인 중인 필드의 이름입니다.

`parentTypeName`

현재 확인 중인 필드에 대한 상위 유형의 이름입니다.

`variables`

GraphQL 요청에 전달된 모든 변수를 포함하는 맵입니다.

`selectionSetList`

GraphQL 선택 세트에 있는 필드의 목록 표현입니다. 별칭이 있는 필드는 필드 이름이 아닌 별칭 이름으로만 참조됩니다. 다음 예제는 이 구조를 자세히 보여 줍니다.

`selectionSetGraphQL`

GraphQL 스키마 정의 언어(SDL)로 형식이 지정된 선택 세트의 문자열 표현입니다. 조각이 선택 세트에 병합되지는 않지만 다음 예제와 같이 인라인 조각은 유지됩니다.

> **Note**
>
> `JSON.stringify`는 문자열 직렬화에 `selectionSetGraphQL` 및 `selectionSetList`를 포함하지 않습니다. 이러한 속성은 사용자가 직접 참조해야 합니다.

예를 들어, 다음 쿼리의 `getPost` 필드를 해석하는 경우:

```
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
  }
}
```

핸들러를 처리할 때 사용할 수 있는 전체 ctx.info 변수는 다음과 같습니다.

```
{
  "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}"
}
```

selectionSetList는 현재 유형에 속하는 필드만 노출합니다. 현재 유형이 인터페이스 또는 집합인 경우 인터페이스에 속하는 선택된 필드만 노출됩니다. 예를 들어 다음과 같은 스키마:

```
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
}
```

그리고 다음과 같은 쿼리:

```
query {
    node(id: "post1") {
        id
        ... on Post {
            title
        }

        ... on Blog {
            title
        }
    }
}
```

`Query.node` 필드 해상도에서 `ctx.info.selectionSetList`를 호출하면 `id`만 노출됩니다.

```
"selectionSetList": [
    "id"
]
```

해석기 및 함수에 대한 JavaScript 런타임 기능

`APPSYNC_JS` 런타임 환경은 ECMAScript (ES) 버전 6.0과 유사한 기능을 제공합니다. 일부 기능을 지원하며 ES 사양의 일부가 아닌 몇 가지 추가 메서드(유틸리티)를 제공합니다. 다음 항목에는 지원되는 모든 언어 기능이 나열되어 있습니다.

> **Note**
>
> 현재 이 참조는 런타임 버전 1.0.0에만 적용됩니다.

주제

- 지원되는 런타임 기능
- 기본 제공 유틸리티
- 기본 제공 모듈
- 런타임 유틸리티
- util.time의 시간 도우미
- util.dynamodb의 DynamoDB 도우미
- util.http의 HTTP 헬퍼
- util.transform의 변환 도우미
- util.str의 문자열 도우미
- 확장 프로그램
- utils.xml의 XML 도우미

지원되는 런타임 기능

아래 섹션에서는 APPSYNC_JS 런타임의 지원되는 기능 세트를 설명합니다.

핵심 기능

다음 핵심 기능이 지원됩니다.

유형

다음 유형이 지원됩니다.

- 번호
- 문자열
- 부울
- 객체
- 배열
- 함수

연산자

다음을 포함한 연산자가 지원됩니다.

- 표준 수학 연산자(+, -, /, %, * 등)
- nullish 병합 연산자(??)
- 선택적 체인(?.)
- bitwise 연산자
- void 및 typeof 연산자

다음 연산자는 지원되지 않습니다.

- 단항 연산자(++, -- 및 ~)
- in 연산자

> **Note**
>
> Object.hasOwn 연산자를 사용하여 지정된 속성이 지정된 객체에 있는지 확인합니다.

명령문

다음 명령문이 지원됩니다.

- const
- let
- var
- break
- else
- for-in
- for-of
- if
- return
- switch
- 스프레드 구문

다음은 지원하지 않습니다.

- `catch`
- `continue`
- `do-while`
- `finally`
- `for(initialization; condition; afterthought)`

> **Note**
>
> 지원되는 예외는 `for-in` 및 `for-of` 표현식입니다.

- `throw`
- `try`
- `while`
- 레이블이 지정된 명령문

리터럴

다음 ES 6 템플릿 리터럴이 지원됩니다.

- 여러 줄 문자열
- 표현식 보간
- 매핑 템플릿

함수

다음 함수 구문이 지원됩니다.

- 함수 선언이 지원됩니다.
- ES 6 화살표 함수가 지원됩니다.
- ES 6 rest 파라미터 구문이 지원됩니다.

엄격 모드

함수는 기본적으로 엄격 모드에서 작동하므로 함수 코드에 `use_strict` 명령문을 추가할 필요가 없습니다. 이것은 변경할 수 없습니다.

기본 객체

다음의 ES 기본 객체와 해당 기능이 지원됩니다.

객체

다음 객체가 지원됩니다.

- `Object.assign()`
- `Object.entries()`
- `Object.hasOwn()`
- `Object.keys()`
- `Object.values()`
- `delete`

문자열

다음 문자열이 지원됩니다.

- `String.prototype.length()`
- `String.prototype.charAt()`
- `String.prototype.concat()`
- `String.prototype.endsWith()`
- `String.prototype.indexOf()`
- `String.prototype.lastIndexOf()`
- `String.raw()`
- `String.prototype.replace()`

> **Note**
>
> 정규 표현식은 지원되지 않습니다.

- `String.prototype.replaceAll()`

> **Note**
>
> 정규 표현식은 지원되지 않습니다.

- `String.prototype.slice()`
- `String.prototype.split()`
- `String.prototype.startsWith()`
- `String.prototype.toLowerCase()`
- `String.prototype.toUpperCase()`
- `String.prototype.trim()`
- `String.prototype.trimEnd()`
- `String.prototype.trimStart()`

숫자

다음 번호가 지원됩니다.

- `Number.isFinite`
- `Number.isNaN`

기본 제공 객체 및 함수

다음 객체 및 함수가 지원됩니다.

수학 연산

다음 수학 연산 함수가 지원됩니다.

- `Math.random()`
- `Math.min()`
- `Math.max()`
- `Math.round()`
- `Math.floor()`
- `Math.ceil()`

배열

다음 배열 메서드가 지원됩니다.

- `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()`는 인수를 지원하지 않습니다.

- `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`

콘솔

콘솔 객체를 디버깅에 사용할 수 있습니다. 라이브 쿼리 실행 중에 콘솔 로그/오류 명령문이 Amazon CloudWatch Logs로 전송됩니다(로깅이 활성화된 경우). `evaluateCode`를 사용하여 코드를 평가하는 동안 명령 응답으로 로그 명령문이 반환됩니다.

- `console.error()`
- `console.log()`

JSON

다음 JSON 메서드가 지원됩니다.

- `JSON.parse()`

> **Note**
> 구문 분석된 문자열이 유효한 JSON이 아닌 경우 빈 문자열을 반환합니다.

- `JSON.stringify()`

함수

- `apply`, `bind` 및 `call` 메서드는 지원되지 않습니다.
- 함수 생성자는 지원되지 않습니다.
- 함수를 인수로 전달하는 것은 지원되지 않습니다.
- 재귀 함수 호출은 지원되지 않습니다.

약속

비동기식 프로세스 및 약속은 지원되지 않습니다.

> **Note**
>
> AWS AppSync의 APPSYNC_JS 런타임 내에서는 네트워크 및 파일 시스템 액세스가 지원되지 않습니다. AWS AppSync는 AWS AppSync 해석기 또는 AWS AppSync 기능의 요청에 따라 모든 I/O 작업을 처리합니다.

Globals

다음과 같은 글로벌 제약이 지원됩니다.

- `NaN`
- `Infinity`
- `undefined`
- `util`
- `extensions`
- `runtime`

오류 유형

`throw`를 사용한 오류 발생은 지원되지 않습니다. `util.error()` 함수를 사용하여 오류를 반환할 수 있습니다. `util.appendError` 함수를 사용하여 GraphQL 응답에 오류를 포함할 수 있습니다.

자세한 내용은 오류 유틸리티를 참조하세요.

기본 제공 유틸리티

`util` 변수에는 데이터 작업에 도움이 되는 일반 유틸리티 메서드가 포함되어 있습니다. 달리 지정하지 않는 한, 모든 유틸리티는 UTF-8 문자 집합을 사용합니다.

인코딩 유틸리티

인코딩 유틸리티 목록

`util.urlEncode(String)`

입력 문자열을 `application/x-www-form-urlencoded` 인코딩 문자열로 반환합니다.

`util.urlDecode(String)`

application/x-www-form-urlencoded 인코딩 문자열을 인코딩되지 않은 형식으로 다시 디코딩합니다.

`util.base64Encode(string) : string`

입력을 base64 인코딩 문자열로 인코딩합니다.

`util.base64Decode(string) : string`

데이터를 base64 인코딩 문자열에서 디코딩합니다.

ID 생성 유틸리티

ID 생성 유틸리티

`util.autoId()`

임의로 생성된 128비트 UUID를 반환합니다.

`util.autoUlid()`

무작위로 생성된 128비트 ULID(Universally Unique Lexicographically Sortable Identifier)를 반환합니다.

`util.autoKsuid()`

길이가 27인 문자열로 인코딩된 무작위로 생성된 128비트 KSUID(K-Sortable Unique Identifier) base62를 반환합니다.

오류 유틸리티

오류 유틸리티 목록

`util.error(String, String?, Object?, Object?)`

사용자 지정 오류를 발생시킵니다. 이 필드는 요청 또는 응답 매핑 템플릿에서 템플릿이 요청 또는 호출 결과와 관련된 오류를 감지하는 경우 사용할 수 있습니다. 또한 `errorType`, `data` 및 `errorInfo` 필드를 지정할 수 있습니다. GraphQL 응답에서 `data` 값은 `error` 내 해당 `errors` 블록에 추가됩니다.

> **Note**
>
> 참고: data는 쿼리 선택 세트를 기반으로 필터링됩니다. GraphQL 응답에서 errorInfo 값은 error 내 해당 errors 블록에 추가됩니다.
> errorInfo는 쿼리 선택 세트를 기반으로 필터링되지 않습니다.

`util.appendError(String, String?, Object?, Object?)`

사용자 지정 오류를 추가합니다. 이 필드는 요청 또는 응답 매핑 템플릿에서 템플릿이 요청 또는 호출 결과와 관련된 오류를 감지하는 경우 사용할 수 있습니다. 또한 errorType, data 및 errorInfo 필드를 지정할 수 있습니다. `util.error(String, String?, Object?, Object?)`와 달리 템플릿 평가가 중단되지 않기 때문에 데이터를 호출자에게 반환할 수 있습니다. GraphQL 응답에서 data 값은 error 내 해당 errors 블록에 추가됩니다.

> **Note**
>
> 참고: data는 쿼리 선택 세트를 기반으로 필터링됩니다. GraphQL 응답에서 errorInfo 값은 error 내 해당 errors 블록에 추가됩니다.
> errorInfo는 쿼리 선택 세트를 기반으로 필터링되지 않습니다.

유형 및 패턴 매칭 유틸리티

유형 및 패턴 매칭 유틸리티 목록

`util.matches(String, String) : Boolean`

첫 번째 인수의 지정된 패턴이 두 번째 인수에서 제공되는 데이터와 일치하는 경우 true를 반환합니다. 패턴은 `util.matches("a*b", "aaaaab")` 등과 같은 정규식이어야 합니다. 이 기능은 Pattern을 기반으로 하며, 여기서 자세한 설명을 참조할 수 있습니다.

`util.authType()`

요청에 사용되는 다중 인증 유형을 설명하는 문자열을 반환하고 'IAM 권한 부여', '사용자 풀 권한 부여', 'Open ID Connect 권한 부여' 또는 'API 키 인증'을 반환합니다.

반환 값 동작 유틸리티

반환 값 동작 유틸리티 목록

`util.escapeJavaScript(String)`

입력 문자열을 JavaScript 의 이스케이프된 문자열로 반환합니다.

해석기 권한 부여 유틸리티

해석기 권한 부여 유틸리티 목록

`util.unauthorized()`

해석 중인 필드에 대해 `Unauthorized`를 발생시킵니다. 요청 또는 응답 매핑 템플릿에서 이를 사용하여 호출자가 필드를 확인하도록 허용할지 여부를 결정합니다.

기본 제공 모듈

모듈은 `APPSYNC_JS` 런타임의 일부이며 JavaScript 해석기 및 함수를 작성하는 데 도움이 되는 유틸리티를 제공합니다.

DynamoDB 모듈 함수

DynamoDB 모듈 함수는 DynamoDB 데이터 소스와 상호 작용할 때 향상된 경험을 제공합니다. 함수를 사용하고 형식 매핑을 추가하지 않고도 DynamoDB 데이터 소스에 요청을 보낼 수 있습니다.

모듈은 `@aws-appsync/utils/dynamodb`를 사용하여 가져옵니다.

```
// Modules are imported using @aws-appsync/utils/dynamodb
import * as ddb from '@aws-appsync/utils/dynamodb';
```

함수

함수 목록

`get<T>(payload: GetInput): DynamoDBGetItemRequest`

> **Tip**
>
> GetInput에 대한 자세한 내용은 the section called “입력”를 참조하세요.

`DynamoDBGetItemRequest` 객체를 생성하여 DynamoDB에 GetItem 요청을 보냅니다.

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

export function request(ctx) {
 return get({ key: { id: ctx.args.id } });
}
```

`put<T>(payload): DynamoDBPutItemRequest`

`DynamoDBPutItemRequest` 객체를 생성하여 DynamoDB에 PutItem 요청을 보냅니다.

```
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`

`DynamoDBDeleteItemRequest` 객체를 생성하여 DynamoDB에 DeleteItem 요청을 보냅니다.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 return ddb.remove({ key: { id: ctx.args.id } });
}
```

`scan<T>(payload): DynamoDBScanRequest`

`DynamoDBScanRequest`를 생성하여 DynamoDB에 Scan 요청을 보냅니다.

```
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`

DynamoDBSyncRequest 객체를 생성하여 Sync 요청을 보냅니다. 요청은 마지막 쿼리(델타 업데이트) 이후 변경된 데이터만 수신합니다. 버전이 지정된 DynamoDB 데이터 소스에만 요청할 수 있습니다.

```
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`

DynamoDBUpdateItemRequest 객체를 생성하여 DynamoDB에 UpdateItem 요청을 보냅니다.

운영

작업 도우미를 사용하면 업데이트 중에 일부 데이터의 특정 조치를 취할 수 있습니다. 시작하려면 `@aws-appsync/utils/dynamodb`에서 `operations`를 가져옵니다.

```
// Modules are imported using operations
import {operations} from '@aws-appsync/utils/dynamodb';
```

작업 목록

`add<T>(payload)`

DynamoDB를 업데이트할 때 새 속성 항목을 추가하는 도우미 함수입니다.

예

ID 값을 사용하여 기존 DynamoDB 항목에 주소(도로명, 구/군/시, 우편번호)를 추가하려면:

```
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)`

DynamoDB의 기존 목록에 페이로드를 추가하는 도우미 함수입니다.

예

업데이트 중에 새로 추가된 친구 ID(`newFriendIds`)를 기존 친구 목록(`friendsIds`)에 추가하려면:

```
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?)`

DynamoDB를 업데이트할 때 항목의 기존 속성 값을 감소시키는 도우미 함수입니다.

예

친구 카운터(`friendsCount`)를 10씩 줄이려면:

```
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?)`

DynamoDB를 업데이트할 때 항목의 기존 속성 값을 증가시키는 도우미 함수입니다.

예

친구 카운터(`friendsCount`)를 10씩 늘리려면:

```
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)`

DynamoDB의 기존 목록 앞에 추가되는 도우미 함수입니다.

예

업데이트 중에 새로 추가된 친구 ID(`newFriendIds`)를 기존 친구 목록(`friendsIds`)에 추가하려면:

```
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)`

DynamoDB에서 항목을 업데이트할 때 기존 속성을 대체하는 도우미 함수입니다. 이는 페이로드의 키뿐만 아니라 속성의 전체 객체 또는 하위 객체를 업데이트하려는 경우에 유용합니다.

예

객체의 주소(도로명, 구/군/시, 우편번호)를 바꾸려면 다음을 수행하세요.

```
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)`

목록의 항목을 대체하는 도우미 함수입니다.

예

업데이트(newFriendIds) 범위에서, 이 예에서는 updateListItem을 사용하여 목록(friendsIds)에 있는 두 번째 항목(인덱스: 1, 새 ID: 102)과 세 번째 항목(인덱스: 2, 새 ID: 112)의 ID 값을 업데이트했습니다.

```
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 });
```

```
}
```

입력

입력 목록

Type GetInput<T>

```
GetInput<T>: {
    consistentRead?: boolean;
    key: DynamoDBKey<T>;
}
```

유형 선언

- consistentRead?: boolean(선택 사항)

  DynamoDB를 사용하여 강력히 일관된 읽기를 수행할지 여부를 지정하는 선택적 부울입니다.

- key: DynamoDBKey<T>(필수)

  DynamoDB에서 항목의 키를 지정하는 필수 파라미터입니다. DynamoDB 항목은 단일 해시 키 또는 해시 및 정렬 키를 가질 수 있습니다.

Type PutInput<T>

```
PutInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T> | null;
    customPartitionKey?: string;
    item: Partial<T>;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
}
```

유형 선언

- _version?: number (선택 사항)
- condition?: DynamoDBFilterObject<T> | null(선택 사항)

  DynamoDB 테이블에 객체를 넣을 때 작업이 수행되기 전에 이미 DynamoDB에 있는 객체의 상태를 기반으로 요청의 성공 여부를 제어하는 조건 표현식을 선택적으로 지정할 수 있습니다.

- customPartitionKey?: string(선택 사항)

활성화되면 이 문자열 값은 버전 관리가 활성화되었을 때 델타 동기화 테이블에서 사용되는 ds_sk 및 ds_pk 레코드의 형식을 수정합니다. 활성화되면 populateIndexFields 항목 처리도 활성화됩니다.

- item: Partial<T>(필수)

  DynamoDB에 배치할 항목의 나머지 속성입니다.

- key: DynamoDBKey<T>(필수)

  입력이 수행되는 DynamoDB에서 항목의 키를 지정하는 필수 파라미터입니다. DynamoDB 항목은 단일 해시 키 또는 해시 및 정렬 키를 가질 수 있습니다.

- populateIndexFields?: boolean(선택 사항)

  customPartitionKey와 함께 활성화되면 델타 동기화 테이블, 특히 gsi_ds_pk 및 gsi_ds_sk 열의 각 레코드에 대해 새 항목을 생성하는 부울 값입니다. 자세한 내용은 AWS AppSync 개발자 안내서의 충돌 감지 및 동기화를 참조하세요.

Type QueryInput<T>

```
QueryInput<T>: ScanInput<T> & {
    query: DynamoDBKeyCondition<Required<T>>;
}
```

유형 선언

- query: DynamoDBKeyCondition<Required<T>>(필수)

  쿼리할 항목을 설명하는 키 조건을 지정합니다. 지정된 인덱스의 경우 파티션 키의 조건은 동등이어야 하고 정렬 키는 비교 또는 beginsWith(문자열인 경우)여야 합니다. 파티션 및 정렬 키에는 숫자 및 문자열 유형만 지원됩니다.

  예

  아래 User 유형을 선택하세요.

```
type User = {
  id: string;
  name: string;
  age: number;
  isVerified: boolean;
  friendsIds: string[]
```

```
}
```

쿼리에는 id, name 및 age 필드만 포함될 수 있습니다.

```
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;
}
```

유형 선언

- _version?: number (선택 사항)
- condition?: DynamoDBFilterObject<T>(선택 사항)

  DynamoDB에서 객체를 제거할 때 작업이 수행되기 전에 이미 DynamoDB에 있는 객체의 상태를 기반으로 요청의 성공 여부를 제어하는 조건 표현식을 선택적으로 지정할 수 있습니다.

  예

  다음 예는 문서의 소유자가 요청한 사용자와 일치하는 경우에만 작업이 성공하도록 허용하는 조건이 포함된 DeleteItem 표현식입니다.

```
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(선택 사항)

  활성화되면 customPartitionKey 값은 버전 관리가 활성화되었을 때 델타 동기화 테이블에서 사용되는 ds_sk 및 ds_pk 레코드의 형식을 수정합니다. 활성화되면 populateIndexFields 항목 처리도 활성화됩니다.

- key: DynamoDBKey<T>(필수)

  제거되는 DynamoDB에서 항목의 키를 지정하는 필수 파라미터입니다. DynamoDB 항목은 단일 해시 키 또는 해시 및 정렬 키를 가질 수 있습니다.

  예

  User에 사용자 id의 해시 키만 있는 경우 키는 다음과 같습니다.

  ```
  type User = {
   id: number
   name: string
   age: number
   isVerified: boolean
  }
  const key: DynamoDBKey<User> = {
   id: 1,
  }
  ```

  테이블 사용자에게 해시 키(id)와 정렬 키(name)가 있는 경우 키는 다음과 같습니다.

  ```
  type User = {
   id: number
   name: string
   age: number
   isVerified: boolean
   friendsIds: string[]
  }
  ```

```
const key: DynamoDBKey<User> = {
 id: 1,
 name: 'XXXXXXXXXX',
}
```

- `populateIndexFields?: boolean`(선택 사항)

  `customPartitionKey`와 함께 활성화되면 델타 동기화 테이블, 특히 `gsi_ds_pk` 및 `gsi_ds_sk` 열의 각 레코드에 대해 새 항목을 생성하는 부울 값입니다.

`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;
}
```

유형 선언

- `consistentRead?: boolean | null`(선택 사항)

  DynamoDB를 쿼리할 때 일관된 읽기를 나타내는 선택적 부울입니다. 기본 값은 `false`입니다.

- `filter?: DynamoDBFilterObject<T> | null`(선택 사항)

  테이블에서 결과를 검색한 후 결과에 적용할 선택적 필터입니다.

- `index?: string | null`(선택 사항)

  스캔할 인덱스의 선택적 이름입니다.

- `limit?: number | null`(선택 사항)

  반환할 선택적 최대 결과 수입니다.

- `nextToken?: string | null`(선택 사항)

  이전 쿼리를 지속하는 선택적 페이지 매김 토큰입니다. 이 토큰은 이전 쿼리에서 얻습니다.

- `scanIndexForward?: boolean | null`(선택 사항)

쿼리가 오름차순 또는 내림차순으로 수행되는지 여부를 나타내는 선택적 부울입니다. 기본적으로 이 값은 true로 설정됩니다.

- segment?: number (선택 사항)
- select?: DynamoDBSelectAttributes(선택 사항)

  DynamoDB에서 반환할 속성입니다. 기본적으로 AWS AppSync DynamoDB 해석기는 인덱스로 프로젝션된 속성만 반환합니다. 지원되는 값은 다음과 같습니다.

  - ALL_ATTRIBUTES

    지정한 테이블 또는 인덱스에서 항목 속성을 모두 반환합니다. 로컬 보조 인덱스를 쿼리하는 경우 DynamoDB는 인덱스의 일치하는 각 항목에 대해 상위 테이블의 전체 항목을 가져옵니다. 인덱스가 모든 항목 속성을 프로젝션하도록 구성된 경우, 모든 데이터를 로컬의 보조 인덱스에서 얻을 수 있기 때문에 가져올 필요가 없습니다.

  - ALL_PROJECTED_ATTRIBUTES

    인덱스로 프로젝션된 모든 속성을 반환합니다. 모든 속성을 프로젝션하도록 인덱스가 구성된 경우 이 반환 값은 ALL_ATTRIBUTES를 지정하는 것과 동일합니다.

  - SPECIFIC_ATTRIBUTES

    ProjectionExpression에 나열된 속성만 반환합니다. 이 반환 값은 AttributesToGet에 대한 값을 지정하지 않고 ProjectionExpression을 지정하는 것과 같습니다.

- totalSegments?: number(선택 사항)

Type DynamoDBSyncInput<T>

```
DynamoDBSyncInput<T>: {
    basePartitionKey?: string;
    deltaIndexName?: string;
    filter?: DynamoDBFilterObject<T> | null;
    lastSync?: number;
    limit?: number | null;
    nextToken?: string | null;
}
```

유형 선언

- basePartitionKey?: string(선택 사항)

동기화 작업을 수행할 때 사용되는 기본 테이블의 파티션 키입니다. 이 필드를 사용하면 테이블에서 사용자 지정 파티션 키를 활용할 때 동기화 작업을 수행할 수 있습니다.

- `deltaIndexName?: string`(선택 사항)

  동기화 작업에 사용되는 인덱스입니다. 이 인덱스는 테이블에서 사용자 지정 파티션 키를 사용할 때 전체 델타 저장소 테이블에서 동기화 작업을 활성화하는 데 필요합니다. 동기화 작업은 GSI(`gsi_ds_pk` 및 `gsi_ds_sk`에서 생성)에서 수행됩니다.

- `filter?: DynamoDBFilterObject<T> | null`(선택 사항)

  테이블에서 결과를 검색한 후 결과에 적용할 선택적 필터입니다.

- `lastSync?: number`(선택 사항)

  마지막으로 성공한 동기화 작업이 시작되었던 시간(epoch 밀리초)입니다. 지정된 경우, `lastSync` 이후에 변경된 항목만 반환됩니다. 이 필드는 초기 동기화 작업에서 모든 페이지를 가져온 이후에만 채워져야 합니다. 생략하면 기본 테이블의 결과가 반환됩니다. 그렇지 않으면 델타 테이블의 결과가 반환됩니다.

- `limit?: number | null`(선택 사항)

  한 번에 평가올 수 있는 선택적인 최대 항목 수입니다. 생략된 경우, 기본 제한이 `100`개 항목으로 설정됩니다. 이 필드의 최대값은 `1000`개 항목입니다.

- `nextToken?: string | null`(선택 사항)

`Type DynamoDBUpdateInput<T>`

```
DynamoDBUpdateInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T>;
    customPartitionKey?: string;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
    update: DynamoDBUpdateObject<T>;
}
```

유형 선언

- `_version?: number` (선택 사항)
- `condition?: DynamoDBFilterObject<T>`(선택 사항)

DynamoDB에서 객체를 업데이트할 때 작업이 수행되기 전에 이미 DynamoDB에 있는 객체의 상태를 기반으로 요청의 성공 여부를 제어하는 조건 표현식을 선택적으로 지정할 수 있습니다.

- `customPartitionKey?: string`(선택 사항)

  활성화되면 `customPartitionKey` 값은 버전 관리가 활성화되었을 때 델타 동기화 테이블에서 사용되는 `ds_sk` 및 `ds_pk` 레코드의 형식을 수정합니다. 활성화되면 `populateIndexFields` 항목 처리도 활성화됩니다.

- `key: DynamoDBKey<T>`(필수)

  업데이트되는 DynamoDB에서 항목의 키를 지정하는 필수 파라미터입니다. DynamoDB 항목은 단일 해시 키 또는 해시 및 정렬 키를 가질 수 있습니다.

- `populateIndexFields?: boolean`(선택 사항)

  `customPartitionKey`와 함께 활성화되면 델타 동기화 테이블, 특히 `gsi_ds_pk` 및 `gsi_ds_sk` 열의 각 레코드에 대해 새 항목을 생성하는 부울 값입니다.

- `update: DynamoDBUpdateObject<T>`

  업데이트될 속성과 해당 속성에 대한 새 값을 지정하는 객체입니다. 업데이트 객체는 `add`, `remove`, `replace`, `increment`, `decrement`, `append`, `prepend`, `updateListItem`와 함께 사용할 수 있습니다.

Amazon RDS 모듈 함수

Amazon RDS 모듈 함수는 Amazon RDS 데이터 API로 구성된 데이터베이스와 상호 작용할 때 향상된 경험을 제공합니다. 모듈은 `@aws-appsync/utils/rds`를 사용하여 가져옵니다.

```
import * as rds from '@aws-appsync/utils/rds';
```

함수를 개별적으로 가져올 수도 있습니다. 예를 들어 아래 가져오기는 `sql`을 사용합니다.

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

함수

AWS AppSync RDS 모듈의 유틸리티 도우미를 사용하여 데이터베이스와 상호 작용할 수 있습니다.

Select

`select` 유틸리티는 관계형 데이터베이스를 쿼리하기 위한 SELECT 문을 생성합니다.

기본 사용

기본 형식에서 쿼리하려는 테이블을 지정할 수 있습니다.

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // "SELECT * FROM "persons"
    return createPgStatement(select({table: 'persons'}));
}
```

테이블 식별자에 스키마를 지정할 수도 있습니다.

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // SELECT * FROM "private"."persons"
    return createPgStatement(select({table: 'private.persons'}));
}
```

열 지정

`columns` 속성을 사용하여 열을 지정할 수 있습니다. 값으로 설정되지 않은 경우 기본값은 `*`입니다.

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name']
    }));
}
```

열 테이블도 지정할 수 있습니다.

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "persons"."name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'persons.name']
    }));
}
```

제한 및 오프셋

쿼리에 `limit` 및 `offset`을 적용할 수 있습니다.

```
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
    }));
}
```

순서 기준

`orderBy` 속성을 기준으로 결과를 정렬할 수 있습니다. 열을 지정하는 객체 배열과 선택적 `dir` 속성을 제공하십시오.

```
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'}]
    }));
}
```

필터

특수 조건 객체를 사용하여 필터를 구축할 수 있습니다.

```
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'}}
    }));
}
```

필터를 결합할 수도 있습니다.

```
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}}
    }));
}
```

OR 문을 작성할 수 있습니다.

```
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 } }
        ]}
    }));
}
```

not으로 조건을 무효화할 수도 있습니다.

```
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 } }
        ]}
    }));
}
```

다음 작업을 사용하여 값을 비교할 수도 있습니다.

연산자	Description	가능한 값 유형
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

이 `insert` 유틸리티는 `INSERT` 작업과 함께 데이터베이스에 단일 행 항목을 삽입하는 간단한 방법을 제공합니다.

단일 항목 삽입

항목을 삽입하려면 테이블을 지정한 다음 값 객체를 전달하십시오. 객체 키는 테이블 열에 매핑됩니다. 열 이름은 자동으로 이스케이프 처리되고 변수 맵을 사용하여 값이 데이터베이스로 전송됩니다.

```
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)
}
```

MySQL 사용 사례

`select`가 뒤따르는 `insert`를 결합하여 삽입된 행을 검색할 수 있습니다.

```
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)
}
```

Postgres 사용 사례

Postgres를 사용하면 삽입한 행에서 데이터를 가져오는 데 [returning]을 사용할 수 있습니다. * 또는 열 이름 배열을 받아들입니다.

```
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)
```

```
}
```

업데이트

update 유틸리티를 사용하여 기존 행을 업데이트할 수 있습니다. 조건 객체를 사용하여 조건을 충족하는 모든 행의 지정된 열에 변경 내용을 적용할 수 있습니다. 예를 들어 이러한 변형을 만들 수 있는 스키마가 있다고 가정해 보겠습니다. 해당 2000 연도부터 인식(known_since)하게 된 경우에만 Person의name을 3의 id 값으로 업데이트하려고 합니다.

```
mutation Update {
    updatePerson(
        input: {id: 3, name: "Jon"},
        condition: {known_since: {ge: "2000"}}
    ) {
    id
    name
  }
}
```

업데이트 해석기는 다음과 같이 생겼습니다.

```
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)
```

```
}
```

조건에 검사를 추가하여 3와 동등한 프라이머리 키 id를 가진 행만 업데이트되도록 할 수 있습니다. 마찬가지로 Postgres inserts의 경우 수정된 데이터를 반환하는 데 returning을 사용할 수 있습니다.

Remove

remove 유틸리티를 사용하여 기존 행을 제거할 수 있습니다. 조건을 충족하는 모든 행에서 조건 객체를 사용할 수 있습니다. 참고로 delete는 JavaScript에서 예약된 키워드입니다. 대신 remove를 사용해야 합니다.

```
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)
}
```

캐스팅

올바른 객체 유형을 더 구체적으로 지정하여 문에 사용하려는 경우가 있을 수 있습니다. 제공된 유형 힌트를 사용하여 파라미터 유형을 지정할 수 있습니다. AWS AppSync는 데이터 API와 동일한 유형 힌트를 지원합니다. AWS AppSync rds 모듈의 typeHint 함수를 사용하여 파라미터를 캐스팅할 수 있습니다.

다음 예시에서는 JSON 객체로 캐스팅된 값으로 배열을 보낼 수 있습니다. -> 연산자를 사용하여 JSON 배열의 index 2에서 요소를 검색합니다.

```
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
}
```

캐스팅은 DATE, TIME 및 TIMESTAMP 등을 처리하고 비교할 때도 유용합니다.

```
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)
}
```

다음에서는 현재 날짜와 시간을 보낼 수 있는 방법을 보여줍니다.

```
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)}`)
}
```

사용 가능한 유형 힌트

- `typeHint.DATE` – 해당 파라미터는 DATE 유형의 객체로 데이터베이스에 전송됩니다. 승인된 형식은 `YYYY-MM-DD`입니다.
- `typeHint.DECIMAL` – 해당 파라미터는 DECIMAL 유형의 객체로 데이터베이스에 전송됩니다.
- `typeHint.JSON` – 해당 파라미터는 JSON 유형의 객체로 데이터베이스에 전송됩니다.
- `typeHint.TIME` – 해당 문자열 파라미터 값은 TIME 유형의 객체로 데이터베이스에 전송됩니다. 승인된 형식은 `HH:MM:SS[.FFF]`입니다.

- `typeHint.TIMESTAMP` – 해당 문자열 파라미터 값은 `TIMESTAMP` 유형의 객체로 데이터베이스에 전송됩니다. 승인된 형식은 `YYYY-MM-DD HH:MM:SS[.FFF]`입니다.
- `typeHint.UUID` – 해당 문자열 파라미터 값은 `UUID` 유형의 객체로 데이터베이스에 전송됩니다.

런타임 유틸리티

`runtime` 라이브러리는 해석기 및 함수의 런타임 속성을 제어하거나 수정하는 유틸리티를 제공합니다.

런타임 유틸리티 목록

`runtime.earlyReturn(obj?: unknown): never`

이 함수를 간접 호출하면 현재 컨텍스트에 따라 현재 AWS AppSync 함수 또는 해석기(유닛 또는 파이프라인 해석기)의 실행이 중지됩니다. 지정된 객체가 결과로 반환됩니다.

- AWS AppSync 함수 요청 핸들러에서 호출되면 데이터 소스와 응답 핸들러를 건너뛰고 다음 함수 요청 핸들러(또는 이것이 마지막 AWS AppSync 함수인 경우 파이프라인 해석기 응답 핸들러)가 호출됩니다.
- AWS AppSync 파이프라인 해석기 요청 핸들러에서 호출되면 파이프라인 실행을 건너뛰고 파이프라인 해석기 응답 핸들러가 즉시 호출됩니다.

예

```
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
}
```

util.time의 시간 도우미

`util.time` 변수에는 타임스탬프를 생성하고, 날짜/시간 형식 간에 변환하고, 날짜/시간 문자열을 구문 분석하는 데 도움이 되는 날짜/시간 메서드가 포함되어 있습니다. 날짜/시간 형식 구문은 DateTimeFormatter를 기반으로 하며, 여기서 자세한 설명을 참조할 수 있습니다. 아래에서는 몇 가지 예와 사용 가능한 메서드 목록 및 설명을 제공합니다.

시간 유틸리티

시간 유틸리티 목록

`util.time.nowISO8601()`

UTC의 문자열 표현을 ISO8601 형식으로 반환합니다.

`util.time.nowEpochSeconds()`

1970-01-01T00:00:00Z의 epoch부터 지금까지의 시간을 초로 반환합니다.

`util.time.nowEpochMilliSeconds()`

1970-01-01T00:00:00Z의 epoch부터 지금까지의 시간을 밀리초로 반환합니다.

`util.time.nowFormatted(String)`

문자열 입력 유형의 지정된 형식을 사용하여 현재 타임스탬프의 문자열을 UTC로 반환합니다.

`util.time.nowFormatted(String, String)`

문자열 입력 유형의 지정된 형식 및 시간대를 사용하여 시간대의 현재 타임스탬프 문자열을 반환합니다.

`util.time.parseFormattedToEpochMilliSeconds(String, String)`

형식과 함께 문자열로 전달된 타임스탬프를 구문 분석하고 epoch 이후 타임스탬프를 밀리초로 반환합니다.

`util.time.parseFormattedToEpochMilliSeconds(String, String, String)`

형식 및 시간대와 함께 문자열로 전달된 타임스탬프를 구문 분석하고 epoch 이후 타임스탬프를 밀리초로 반환합니다.

`util.time.parseISO8601ToEpochMilliSeconds(String)`

문자열로 전달된 ISO8601 타임스탬프를 구문 분석하고 epoch 이후 타임스탬프를 밀리초로 반환합니다.

`util.time.epochMilliSecondsToSeconds(long)`

epoch 밀리초 타임스탬프를 epoch 초 타임스탬프로 변환합니다.

`util.time.epochMilliSecondsToISO8601(long)`

epoch 밀리초 타임스탬프를 ISO8601 타임스탬프로 변환합니다.

`util.time.epochMilliSecondsToFormatted(long, String)`

long으로 전달된 epoch 밀리초 타임스탬프를 UTC의 제공된 형식에 따라 형식이 지정된 타임스탬프로 변환합니다.

`util.time.epochMilliSecondsToFormatted(long, String, String)`

long으로 전달된 epoch 밀리초 타임스탬프를 제공된 시간대의 제공된 형식에 따라 형식이 지정된 타임스탬프로 변환합니다.

util.dynamodb의 DynamoDB 도우미

`util.dynamodb`에는 Amazon DynamoDB에 데이터 쓰기 및 읽기를 더 용이하게 하는 헬퍼 메서드가 포함되어 있습니다(예: 자동 유형 매핑 및 형식 지정).

toDynamoDB

toDynamoDB 유틸리티 목록

`util.dynamodb.toDynamoDB(Object)`

입력 객체를 적절한 DynamoDB 표현으로 변환하는 DynamoDB용 일반 객체 변환 도구입니다. 이 도구는 몇 가지 형식을 표현하는 방법에 대해 독자적인 방식을 가지고 있습니다. 예를 들어 집합("SS", "NS", "BS")보다는 목록("L")을 사용합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

문자열 예제

```
Input:      util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

숫자 예제

```
Input:      util.dynamodb.toDynamoDB(12345)
```

```
Output:     { "N" : 12345 }
```

부울 예제

```
Input:      util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

록 예제

```
Input:      util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

맵 예제

```
Input:      util.dynamodb.toDynamoDB({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
                           { "S" : "boop" }
                       ]
                   }
               }
           }
```

toString 유틸리티

toString 유틸리티 목록

`util.dynamodb.toString(String)`

입력 문자열을 DynamoDB 문자열 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

`util.dynamodb.toStringSet(List<String>)`

문자열이 포함된 목록을 DynamoDB 문자열 집합 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

toNumber 유틸리티

toNumber 유틸리티 목록

`util.dynamodb.toNumber(Number)`

숫자를 DynamoDB 숫자 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

`util.dynamodb.toNumberSet(List<Number>)`

숫자 목록을 DynamoDB 숫자 집합 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

toBinary 유틸리티

toBinary 유틸리티 목록

`util.dynamodb.toBinary(String)`

base64 문자열로 인코딩된 이진 데이터를 DynamoDB 이진 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

`util.dynamodb.toBinarySet(List<String>)`

base64 문자열로 인코딩된 이진 데이터 목록을 DynamoDB 이진수 집합 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

toBoolean 유틸리티

toBoolean 유틸리티 목록

`util.dynamodb.toBoolean(Boolean)`

부울을 적절한 DynamoDB 부울 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

toNull 유틸리티

toNull 유틸리티 목록

`util.dynamodb.toNull()`

null을 DynamoDB null 형식으로 반환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      util.dynamodb.toNull()
Output:     { "NULL" : null }
```

toList 유틸리티

toList 유틸리티 목록

`util.dynamodb.toList(List)`

객체 목록을 DynamoDB 목록 형식으로 변환합니다. 목록의 각 항목 역시 적절한 DynamoDB 형식으로 변환됩니다. 이 도구는 몇 가지 중첩 객체를 표현하는 방법에 대해 독자적인 방식을 가지고 있습니다. 예를 들어 집합("SS", "NS", "BS")보다는 목록("L")을 사용합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

toMap 유틸리티

toMap 유틸리티 목록

`util.dynamodb.toMap(Map)`

맵을 DynamoDB 맵 형식으로 변환합니다. 맵의 각 값 역시 적절한 DynamoDB 형식으로 변환됩니다. 이 도구는 몇 가지 중첩 객체를 표현하는 방법에 대해 독자적인 방식을 가지고 있습니다. 예를 들어 집합("SS", "NS", "BS")보다는 목록("L")을 사용합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
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)`

각 값이 적절한 DynamoDB 형식으로 변환된 탭의 사본을 생성합니다. 이 도구는 몇 가지 중첩 객체를 표현하는 방법에 대해 독자적인 방식을 가지고 있습니다. 예를 들어 집합("SS", "NS", "BS")보다는 목록("L")을 사용합니다.

```
Input:      util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "foo"  : { "S" : "bar" },
               "baz"  : { "N" : 1234 },
               "beep" : {
                   "L" : [
                       { "S" : "boop" }
                   ]
               }
           }
```

> **Note**
>
> 이는 전체 속성 값 자체가 아니라 DynamoDB 속성 값의 내용만 반환하기 때문에 `util.dynamodb.toMap(Map)`과 약간 다릅니다. 예를 들어 다음 문은 정확하게 동일합니다.
>
> ```
> util.dynamodb.toMapValues(<map>)
> util.dynamodb.toMap(<map>)("M")
> ```

S3Object 유틸리티

S3Object 유틸리티 목록

`util.dynamodb.toS3Object(String key, String bucket, String region)`

키, 버킷 및 리전을 DynamoDB S3 객체 표현으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
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)`

키, 버킷, 리전 및 선택적 버전을 DynamoDB S3 객체 표현으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      util.dynamodb.toS3Object("foo", "bar", "baz", "beep")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region
\" : \"baz\", \"version\" = \"beep\" } }" }
```

`util.dynamodb.fromS3ObjectJson(String)`

DynamoDB S3 객체의 문자열 값을 수락하고 키, 버킷, 리전 및 선택적 버전을 포함하는 맵을 반환합니다.

```
Input:      util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
 \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :
 "beep" }
```

util.http의 HTTP 헬퍼

이 `util.http` 유틸리티는 HTTP 요청 파라미터를 관리하고 응답 헤더를 추가하는 데 사용할 수 있는 도우미 메서드를 제공합니다.

util.http 유틸리티 목록

`util.http.copyHeaders(headers)`

제한된 HTTP 헤더 세트 없이 맵에서 헤더를 복사합니다. 이를 사용하여 요청 헤더를 다운스트림 HTTP 엔드포인트로 전달할 수 있습니다.

`util.http.addResponseHeader(String, Object)`

응답의 이름(`String`) 및 값(`Object`)이 포함된 단일 사용자 지정 헤더를 추가합니다. 다음과 같은 제한이 적용됩니다.

- 헤더 이름은 기존 헤더나 제한된 AWS 또는 AWS AppSync 헤더와 일치할 수 없습니다.
- 헤더 이름은 `x-amzn-` 또는 `x-amz-` 같은 제한된 접두사로 시작할 수 없습니다.
- 사용자 지정 응답 헤더의 크기는 4KB를 초과할 수 없습니다. 여기에는 헤더 이름과 값이 포함됩니다.
- GraphQL 작업당 각 응답 헤더를 한 번씩 정의해야 합니다. 하지만 이름이 같은 사용자 지정 헤더를 여러 번 정의하면 응답에 가장 최근의 정의가 나타납니다. 이름 지정과 상관없이 모든 헤더는 헤더 크기 제한에 포함됩니다.

`util.http.addResponseHeaders(Map)`

지정된 이름 맵(`String`) 및 값(`Object`) 의 응답에 여러 응답 헤더를 추가합니다. `addResponseHeader(String, Object)` 메서드에 나열된 것과 동일한 제한 사항이 이 메서드에도 적용됩니다.

util.transform의 변환 도우미

`util.transform`에는 데이터 소스에 대해 복잡한 작업을 더 쉽게 수행할 수 있게 해주는 도우미 메서드가 포함되어 있습니다.

변환 도우미 유틸리티 목록

```
util.transform.toDynamoDBFilterExpression(filterObject:
DynamoDBFilterObject) : string
```

입력 문자열을 DynamoDB에 사용할 필터 표현식으로 변환합니다. `toDynamoDBFilterExpression`을 기본 제공 모듈 함수와 함께 사용하는 것이 좋습니다.

`util.transform.toElasticsearchQueryDSL(object: OpenSearchQueryObject) : string`

주어진 입력을 이와 동등한 OpenSearch 쿼리 DSL 표현식으로 변환하여 JSON 문자열로 반환합니다.

입력 예:

```
util.transform.toElasticsearchQueryDSL({
    "upvotes":{
        "ne":15,
        "range":[
            10,
            20
        ]
    },
    "title":{
        "eq":"hihihi",
        "wildcard":"h*i"
    }
  })
```

출력 예:

```
{
    "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**
>
> 기본 연산자는 AND로 가정합니다.

`util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter`

Map 입력 객체를 SubscriptionFilter 표현식 객체로 변환합니다. `util.transform.toSubscriptionFilter` 메서드는 `extensions.setSubscriptionFilter()` 확장에 대한 입력으로 사용됩니다. 자세한 내용은 확장을 참조하세요.

> **Note**
>
> 파라미터와 반환 명령문은 다음과 같습니다.
>
> Parameters
>
> - `objFilter`: `SubscriptionFilterObject`
>
> `SubscriptionFilter` 표현식 객체로 변환되는 `Map` 입력 객체입니다.
> - `ignoredFields`: `SubscriptionFilterExcludeKeysType`(선택 사항)
>
> 무시될 첫 번째 객체의 필드 이름 `List`입니다.
> - `rules`: `SubscriptionFilterRuleObject`(선택 사항)
>
> `SubscriptionFilter` 표현식 객체를 생성할 때 포함되는 엄격한 규칙이 있는 `Map` 입력 객체입니다. 이러한 엄격한 규칙이 `SubscriptionFilter` 표현식 객체에 포함되므로 최소한 하나의 규칙이 충족되어 구독 필터를 통과할 수 있습니다.
>
> 응답
>
> `SubscriptionFilter`을 반환합니다.

`util.transform.toSubscriptionFilter(Map, List)`

`Map` 입력 객체를 `SubscriptionFilter` 표현식 객체로 변환합니다. `util.transform.toSubscriptionFilter` 메서드는 `extensions.setSubscriptionFilter()` 확장에 대한 입력으로 사용됩니다. 자세한 내용은 확장을 참조하세요.

첫 번째 인수는 `SubscriptionFilter` 표현식 객체로 변환되는 `Map` 입력 객체입니다. 두 번째 인수는 `SubscriptionFilter` 표현식 객체를 구성하는 동안 첫 번째 `Map` 입력 객체에서 무시되는 필드 이름의 `List`입니다.

`util.transform.toSubscriptionFilter(Map, List, Map)`

`Map` 입력 객체를 `SubscriptionFilter` 표현식 객체로 변환합니다. `util.transform.toSubscriptionFilter` 메서드는 `extensions.setSubscriptionFilter()` 확장에 대한 입력으로 사용됩니다. 자세한 내용은 확장을 참조하세요.

`util.transform.toDynamoDBConditionExpression(conditionObject)`

DynamoDB 조건 표현식을 생성합니다.

구독 필터 인수

다음 표에서는 다음 유틸리티의 인수를 정의하는 방법을 설명합니다.

- `Util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter`

Argument 1: Map

인수 1은 다음과 같은 키 값을 가진 `Map` 객체입니다.

- 필드 이름
- "and"
- "or"

필드 이름을 키로 사용하는 경우 해당 필드 항목의 조건은 `"operator" : "value"` 형식입니다.

다음 예에서는 `Map`에 항목을 추가하는 방법을 보여줍니다.

```
"field_name" : {
                    "operator1" : value
               }

## We can have multiple conditions for the same field_name:

"field_name" : {
                    "operator1" : value
                    "operator2" : value
                    .
                    .
                    .
               }
```

필드에 두 개 이상의 조건이 있는 경우 이러한 모든 조건은 OR 연산을 사용하는 것으로 간주됩니다.

입력 `Map`에 "and"와 "or"를 키로 사용할 수도 있습니다. 즉, 이들 항목 내의 모든 항목은 키에 따라 AND 또는 OR 논리를 사용하여 결합되어야 합니다. 키 값 "and"와 "or"에는 일련의 조건이 필요합니다.

```
"and" : [

            {
                "field_name1" : {
                    "operator1" : value
                }
             },

             {
                "field_name2" : {
                    "operator1" : value
                }
             },
             .
             .
        ].
```

참고로 "and"와 "or"는 중첩되지 않습니다. 즉, 다른 "and"/"or" 블록 내에 "and"/"or"를 중첩할 수 있습니다. 하지만 단순한 필드에서는 작동하지 않습니다.

```
"and" : [

            {
                "field_name1" : {
                    "operator" : value
                }
             },

             {
                "or" : [
                            {
                                "field_name2" : {
                                    "operator" : value
                                }
                            },

                            {
                                "field_name3" : {
                                    "operator" : value
                                }
                            }
```

```
                            ].
```

다음 예제에서는 `util.transform.toSubscriptionFilter(Map) : Map`를 사용하여 인수 1을 입력하는 방법을 보여줍니다.

입력

인수 1: 맵:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

출력

결과는 `Map` 객체입니다.

```
{
  "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

인수 2에는 SubscriptionFilter 표현식 객체를 생성하는 동안 입력 Map(인수 1)에서 고려하면 안 되는 필드 이름의 List가 포함되어 있습니다. List도 비어 있을 수 있습니다.

다음 예제에서는 util.transform.toSubscriptionFilter(Map, List) : Map을 사용하여 인수 1 및 인수 2를 입력하는 방법을 보여줍니다.

입력

인수 1: 맵:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
```

```
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

인수 2: 목록:

```
["percentageUp", "author"]
```

출력

결과는 Map 객체입니다.

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
```

```
}
```

Argument 3: Map

인수 3은 필드 이름을 키 값으로 갖는 Map 객체입니다("and" 또는 "or"를 가질 수 없음). 필드 이름을 키로 사용하는 경우 해당 필드의 조건은 "operator" : "value" 형식의 항목입니다. 인수 1과 달리 인수 3은 동일한 키에 여러 조건을 포함할 수 없습니다. 또한 인수 3에는 "and" 또는 "or" 절이 없으므로 중첩도 필요하지 않습니다.

인수 3은 엄격한 규칙 목록을 나타내며, 이러한 규칙 목록은 필터를 통과하기 위해 이러한 조건 중 하나 이상이 충족되도록 SubscriptionFilter 표현식 객체에 추가됩니다.

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.
```

다음 예제에서는 util.transform.toSubscriptionFilter(Map, List, Map) : Map을 사용하여 인수 1, 인수 2 및 인수 3을 입력하는 방법을 보여줍니다.

입력

인수 1: 맵:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
```

```
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

인수 2: 목록:

```
["percentageUp", "author"]
```

인수 3: 맵:

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
}
```

출력

결과는 Map 객체입니다.

```
{
  "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"
        }
      ]
```

```
    }
  ]
}
```

util.str의 문자열 도우미

`util.str`에는 일반적인 문자열 작업에 도움이 되는 메서드가 있습니다.

util.str 유틸리티 목록

`util.str.normalize(String, String)`

NFC, NFD, NFKC 또는 NFKD의 네 가지 유니코드 정규화 형식 중 하나를 사용하여 문자열을 정규화합니다. 첫 번째 인수는 정규화할 문자열입니다. 두 번째 인수는 정규화 프로세스에 사용할 정규화 유형을 지정하는 'nfc', 'nfd', 'nfkc' 또는 'nfkd'입니다.

확장 프로그램

`extensions`에는 해석기 내에서 추가 작업을 수행할 수 있는 일련의 메서드가 있습니다.

확장 사용

`extensions.evictFromApiCache(typeName: string, fieldName: string, keyValuePair: Record<string, string>) : Object`

AWS AppSync 서버 측 캐시에서 항목을 제거합니다. 첫 번째 인수는 형식 이름입니다. 두 번째 인수는 필드 이름입니다. 세 번째 인수는 캐싱 키 값을 지정하는 키-값 쌍 항목을 포함하는 객체입니다. 캐시된 해석기의 `cachingKey`에 있는 캐싱 키와 동일한 순서로 객체에 항목을 넣어야 합니다. 캐싱에 대한 자세한 정보는 캐싱 동작을 참조하세요.

예제 1:

이 예에서는 이름이 `context.arguments.semester`인 캐싱 키가 사용된 해석기 `Query.allClasses`에 대해 캐시된 항목을 제거합니다. 뮤테이션이 호출되고 해석기가 실행될 때 항목이 성공적으로 지워지면 응답에는 삭제된 항목 수를 표시하는 `apiCacheEntriesDeleted` 값이 확장 객체에 포함됩니다.

```
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**
>
> 이 함수는 뮤테이션에만 작동하고 쿼리에는 작동하지 않습니다.

구독 확장

`extensions.setSubscriptionFilter(filterJsonObject)`

향상된 구독 필터를 정의합니다. 각 구독 알림 이벤트는 제공된 구독 필터에 대해 평가되고 모든 필터가 `true`로 평가되면 클라이언트에 알림을 전달합니다. 인수는 `filterJsonObject`입니다(이 인수에 대한 자세한 내용은 아래 인수: filterJsonObject 섹션에서 확인). 향상된 구독 필터링을 참조하세요.

> **Note**
>
> 이 확장 함수는 구독 해석기의 응답 핸들러에서만 사용할 수 있습니다. 또한 필터를 만들 때는 `util.transform.toSubscriptionFilter`를 사용하는 것이 좋습니다.

`extensions.setSubscriptionInvalidationFilter(filterJsonObject)`

구독 무효화 필터를 정의합니다. 구독 필터는 무효화 페이로드에 대해 평가된 후 필터가 `true`로 평가되면 지정된 구독을 무효화합니다. 인수는 `filterJsonObject`입니다(이 인수에 대한 자세한 내용은 아래 인수: filterJsonObject 섹션에서 확인). 향상된 구독 필터링을 참조하세요.

> **Note**
>
> 이 확장 함수는 구독 해석기의 응답 핸들러에서만 사용할 수 있습니다. 또한 필터를 만들 때는 `util.transform.toSubscriptionFilter`를 사용하는 것이 좋습니다.

`extensions.invalidateSubscriptions(invalidationJsonObject)`

뮤테이션으로부터 구독 무효화를 시작하는 데 사용됩니다. 인수는 `invalidationJsonObject`입니다(이 인수에 대한 자세한 내용은 아래 인수: invalidationJsonObject 섹션에서 확인).

> **Note**
>
> 이 확장은 뮤테이션 해석기의 응답 매핑 템플릿에서만 사용할 수 있습니다.
> 단일 요청에서 고유한 `extensions.invalidateSubscriptions()` 메서드 호출을 최대 5개까지만 사용할 수 있습니다. 이 한도를 초과할 경우 GraphQL 오류가 발생합니다.

인수: filterJsonObject

JSON 객체는 구독 또는 무효화 필터를 정의합니다. `filterGroup`에 있는 필터 배열입니다. 각 필터는 개별 필터의 모음입니다.

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

각 필터에는 다음의 세 가지 속성이 있습니다.

- `fieldName` - GraphQL 스키마 필드
- `operator` - 연산자 유형
- `value` - 구독 알림 `fieldName` 값과 비교할 값.

다음은 이러한 속성에 대한 할당 예시입니다.

```
{
 "fieldName" : "severity",
 "operator" : "le",
 "value" : context.result.severity
}
```

인수: invalidationJsonObject

`invalidationJsonObject`는 다음을 정의합니다.

- `subscriptionField` - 무효화할 GraphQL 스키마 구독입니다. `subscriptionField`에서 문자열로 정의된 단일 구독은 무효화 대상으로 간주됩니다.
- `payload` – 무효화 필터가 해당 값에 대해 `true`로 평가되는 경우 구독 무효화를 위한 입력으로 사용되는 키-값 쌍 목록입니다.

  다음 예에서는 구독 해석기에 정의된 무효화 필터가 `payload` 값에 대해 `true`로 평가될 때 `onUserDelete` 구독을 사용하여 구독 및 연결된 클라이언트를 무효화합니다.

  ```
  export const request = (ctx) => ({ payload: null });

  export function response(ctx) {
   extensions.invalidateSubscriptions({
    subscriptionField: 'onUserDelete',
    payload: { group: 'Developer', type: 'Full-Time' },
   });
   return ctx.result;
  }
  ```

utils.xml의 XML 도우미

`util.xml`에는 XML 문자열 변환에 도움이 되는 메서드가 있습니다.

util.xml 유틸리티 목록

`util.xml.toMap(String) : Object`

XML 문자열을 사전으로 변환합니다.

예제 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"
      }
    }
}
```

예제 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`

XML 문자열을 JSON 문자열로 변환합니다. 출력이 문자열인 것을 제외하면 `toMap`과 비슷합니다. 이는 HTTP 객체에서 JSON으로 XML 응답을 직접 변환하고 반환하고자 하는 경우 유용합니다. 선택적 부울 파라미터를 설정하여 JSON을 문자열 인코딩할지 여부를 결정할 수 있습니다.

DynamoDB용 JavaScript 해석기 함수 참조

AWS AppSync DynamoDB 함수에서는 GraphQL을 사용하여 자신의 계정에 있는 기존 Amazon DynamoDB 테이블의 데이터를 저장 및 검색할 수 있습니다. 이 해석기는 수신되는 GraphQL 요청을 DynamoDB 호출로 매핑한 다음 DynamoDB 응답을 다시 GraphQL로 매핑하도록 하여 작동합니다. 이 섹션에서는 지원되는 DynamoDB 작업에 대한 요청 및 응답 핸들러를 설명합니다.

GetItem

`GetItem` 요청을 사용하면 AWS AppSync DynamoDB 함수에 DynamoDB에 대한 `GetItem` 요청을 하도록 지시하고 다음을 지정할 수 있습니다.

- DynamoDB의 항목 키
- 일관된 읽기를 사용할지 여부

GetItem 요청의 구조는 다음과 같습니다.

```
type DynamoDBGetItem = {
  operation: 'GetItem';
  key: { [key: string]: any };
  consistentRead?: ConsistentRead;
  projection?: {
    expression: string;
    expressionNames?: { [key: string]: string };
  };
};
```

필드는 다음과 같이 정의됩니다.

GetItem 필드

GetItem 필드 목록

`operation`

수행할 DynamoDB 작업입니다. `GetItem` DynamoDB 작업을 수행하려면 이 값을 `GetItem`으로 설정해야 합니다. 이 값은 필수입니다.

`key`

DynamoDB의 항목 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

`consistentRead`

DynamoDB에서 강력히 일관된 읽기를 수행할지 여부. 선택 사항으로, 기본값은 `false`입니다.

`projection`

DynamoDB 작업에서 반환할 속성을 지정하는 데 사용되는 프로젝션입니다. 프로젝션에 대한 자세한 내용은 프로젝션을 참조하세요. 이 필드는 선택 사항입니다.

DynamoDB에서 반환되는 항목은 자동으로 GraphQL 및 JSON 기본 유형으로 변환되며 매핑 컨텍스트(`context.result`)에서 사용할 수 있습니다.

DynamoDB 유형 변환에 대한 자세한 내용은 유형 시스템(응답 매핑)을 참조하세요.

JavaScript 해석기에 대한 자세한 내용은 JavaScript 해석기 개요를 참조하세요.

예

다음 예제는 GraphQL 쿼리 `getThing(foo: String!, bar: String!)`을 위한 함수 요청 핸들러입니다.

```
export function request(ctx) {
  const {foo, bar} = ctx.args
  return {
    operation : "GetItem",
    key : util.dynamodb.toMapValues({foo, bar}),
    consistentRead : true
  }
}
```

DynamoDB `GetItem` API에 대한 자세한 내용은 DynamoDB API 문서를 참조하십시오.

PutItem

`PutItem` 요청 매핑 문서를 사용하면 AWS AppSync DynamoDB 함수에 DynamoDB에 `PutItem` 요청을 하도록 지시하고 다음을 지정할 수 있습니다.

- DynamoDB의 항목 키
- 항목의 전체 내용(`key` 및 `attributeValues`로 구성됨)
- 성공할 작업의 조건

`PutItem` 요청의 구조는 다음과 같습니다.

```
type DynamoDBPutItemRequest = {
  operation: 'PutItem';
  key: { [key: string]: any };
  attributeValues: { [key: string]: any};
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

필드는 다음과 같이 정의됩니다.

PutItem 필드

PutItem 필드 목록

`operation`

수행할 DynamoDB 작업입니다. `PutItem` DynamoDB 작업을 수행하려면 이 값을 `PutItem`으로 설정해야 합니다. 이 값은 필수입니다.

`key`

DynamoDB의 항목 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

`attributeValues`

DynamoDB에 저장할 항목의 나머지 속성. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 필드는 선택 사항입니다.

`condition`

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건을 지정하지 않으면 `PutItem` 요청이 해당 항목에 대한 기존 입력을 덮어씁니다. 조건에 대한 자세한 내용은 조건 표현식을 참조하세요. 이 값은 선택 사항입니다.

`_version`

항목의 알려진 최신 버전을 나타내는 숫자 값입니다. 이 값은 선택 사항입니다. 이 필드는 충돌 감지에 사용되며 버전이 지정된 데이터 원본에만 지원됩니다.

`customPartitionKey`

활성화되면 이 문자열 값은 버전 관리가 활성화되었을 때 델타 동기화 테이블에서 사용되는 `ds_sk` 및 `ds_pk` 레코드의 형식을 수정합니다(자세한 내용은 AWS AppSync 개발자 안내서의 충돌 감지 및 동기화 참조). 활성화되면 `populateIndexFields` 항목 처리도 활성화됩니다. 이 필드는 선택 사항입니다.

`populateIndexFields`

`customPartitionKey`와 함께 활성화되면 델타 동기화 테이블, 특히 `gsi_ds_pk` 및 `gsi_ds_sk` 열의 각 레코드에 대해 새 항목을 생성하는 부울 값입니다. 자세한 내용은 AWS AppSync 개발자 안내서의 충돌 감지 및 동기화를 참조하세요. 이 필드는 선택 사항입니다.

DynamoDB에 기록된 항목은 자동으로 GraphQL 및 JSON 기본 유형으로 변환되며 컨텍스트 결과(`context.result`)에서 사용할 수 있습니다.

DynamoDB에 기록된 항목은 자동으로 GraphQL 및 JSON 기본 유형으로 변환되며 컨텍스트 결과(`context.result`)에서 사용할 수 있습니다.

DynamoDB 유형 변환에 대한 자세한 내용은 유형 시스템(응답 매핑)을 참조하세요.

JavaScript 해석기에 대한 자세한 내용은 JavaScript 해석기 개요를 참조하세요.

예 1

다음 예제는 GraphQL 뮤테이션 `updateThing(foo: String!, bar: String!, name: String!, version: Int!)`을 위한 함수 요청 핸들러입니다.

지정된 키가 있는 항목이 없으면 생성됩니다. 지정된 키가 있는 항목이 있으면 덮어씁니다.

```
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),
  };
}
```

예 2

다음 예제는 GraphQL 뮤테이션 `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)`을 위한 함수 요청 핸들러입니다.

이 예에서는 현재 DynamoDB에 있는 항목의 `version` 필드가 `expectedVersion`으로 설정되어 있는지 확인합니다.

```
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,
  };
}
```

DynamoDB `PutItem` API에 대한 자세한 내용은 DynamoDB API 문서를 참조하십시오.

UpdateItem

`UpdateItem` 요청을 사용하면 AWS AppSync DynamoDB 함수에 DynamoDB에 `UpdateItem` 요청을 하도록 지시하고 다음을 지정할 수 있습니다.

- DynamoDB의 항목 키
- DynamoDB에서 항목 업데이트 방법을 설명하는 업데이트 표현식
- 성공할 작업의 조건

`UpdateItem` 요청의 구조는 다음과 같습니다.

```
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;
};
```

필드는 다음과 같이 정의됩니다.

UpdateItem 필드

UpdateItem 필드 목록

`operation`

수행할 DynamoDB 작업입니다. `UpdateItem` DynamoDB 작업을 수행하려면 이 값을 `UpdateItem`으로 설정해야 합니다. 이 값은 필수입니다.

`key`

DynamoDB의 항목 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값' 지정에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

`update`

`update` 섹션에서는 DynamoDB의 항목 업데이트 방법을 설명하는 업데이트 표현식을 지정합니다. 업데이트 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB UpdateExpressions 문서를 참조하십시오. 이 섹션은 필수입니다.

`update` 섹션에는 다음 세 가지 구성 요소가 있습니다.

`expression`

업데이트 표현식. 이 값은 필수입니다.

`expressionNames`

표현식 속성 name 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 `expression`에 사용된 name 자리 표시자에 해당하고 값은 DynamoDB에 있는 항목의 속성 이름에 해당하는 문자열이어야 합니다. 이 필드는 선택 사항으로, `expression`에 사용된 표현식 속성인 name 자리 표시자의 대체 항목으로만 채워져야 합니다.

`expressionValues`

표현식 속성 value 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 `expression`에 사용되는 value 자리 표시자에 해당하고 값은 입력된 값이어야 합니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 입력된 값은 지정되어 있어야 합니다. 이 필드는 선택 사항으로, `expression`에 사용된 표현식 속성인 value 자리 표시자의 대체 항목으로만 채워져야 합니다.

`condition`

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건을 지정하지 않으면 `UpdateItem` 요청이 현재 상태와 상관 없이 기존 항목을 업데이트합니다. 조건에 대한 자세한 내용은 조건 표현식을 참조하세요. 이 값은 선택 사항입니다.

`_version`

항목의 알려진 최신 버전을 나타내는 숫자 값입니다. 이 값은 선택 사항입니다. 이 필드는 충돌 감지에 사용되며 버전이 지정된 데이터 원본에만 지원됩니다.

`customPartitionKey`

활성화되면 이 문자열 값은 버전 관리가 활성화되었을 때 델타 동기화 테이블에서 사용되는 `ds_sk` 및 `ds_pk` 레코드의 형식을 수정합니다(자세한 내용은 AWS AppSync 개발자 안내서의 충돌 감지 및 동기화 참조). 활성화되면 `populateIndexFields` 항목 처리도 활성화됩니다. 이 필드는 선택 사항입니다.

`populateIndexFields`

`customPartitionKey`와 함께 활성화되면 델타 동기화 테이블, 특히 `gsi_ds_pk` 및 `gsi_ds_sk` 열의 각 레코드에 대해 새 항목을 생성하는 부울 값입니다. 자세한 내용은 AWS AppSync 개발자 안내서의 충돌 감지 및 동기화를 참조하세요. 이 필드는 선택 사항입니다.

DynamoDB에서 업데이트된 항목은 자동으로 GraphQL 및 JSON 기본 유형으로 변환되며 컨텍스트 결과(`context.result`)에서 사용할 수 있습니다.

DynamoDB 유형 변환에 대한 자세한 내용은 유형 시스템(응답 매핑)을 참조하세요.

JavaScript 해석기에 대한 자세한 내용은 JavaScript 해석기 개요를 참조하세요.

예 1

다음 예제는 GraphQL 뮤테이션 `upvote(id: ID!)`를 위한 함수 요청 핸들러입니다.

이 예에서 DynamoDB의 항목에는 1씩 증가하는 `upvotes` 및 `version` 필드가 있습니다.

```
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 } },
    },
  };
}
```

예 2

다음 예제는 GraphQL 뮤테이션 `updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)`을 위한 함수 요청 핸들러입니다.

다음은 인수를 조사해 클라이언트에서 제공한 인수만 포함하는 업데이트 표현식을 동적으로 생성하는 복잡한 예입니다. 예를 들어, `title` 및 `author`는 생략되면 업데이트되지 않습니다. 인수가 지정되어 있으나 그 값이 `null`이면 DynamoDB의 객체에서 해당 필드가 삭제됩니다. 마지막으로, 이 작업에는 DynamoDB에 현재 있는 항목에 `expectedVersion`으로 설정된 `version` 필드가 있는지 확인하는 조건이 있습니다.

```
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),
    },
  };
}
```

DynamoDB `UpdateItem` API에 대한 자세한 내용은 DynamoDB API 문서를 참조하십시오.

DeleteItem

`DeleteItem` 요청을 사용하면 AWS AppSync DynamoDB 함수에 DynamoDB에 `DeleteItem` 요청을 하도록 지시하고 다음을 지정할 수 있습니다.

- DynamoDB의 항목 키
- 성공할 작업의 조건

DeleteItem 요청의 구조는 다음과 같습니다.

```
type DynamoDBDeleteItemRequest = {
  operation: 'DeleteItem';
  key: { [key: string]: any };
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

필드는 다음과 같이 정의됩니다.

DeleteItem 필드

DeleteItem 필드 목록

operation

수행할 DynamoDB 작업입니다. DeleteItem DynamoDB 작업을 수행하려면 이 값을 DeleteItem으로 설정해야 합니다. 이 값은 필수입니다.

key

DynamoDB의 항목 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값' 지정에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

condition

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건을 지정하지 않으면 DeleteItem 요청이 현재 상태와 상관 없이 항목을 삭제합니다. 조건에 대한 자세한 내용은 조건 표현식을 참조하세요. 이 값은 선택 사항입니다.

_version

항목의 알려진 최신 버전을 나타내는 숫자 값입니다. 이 값은 선택 사항입니다. 이 필드는 충돌 감지에 사용되며 버전이 지정된 데이터 원본에만 지원됩니다.

customPartitionKey

활성화되면 이 문자열 값은 버전 관리가 활성화되었을 때 델타 동기화 테이블에서 사용되는 ds_sk 및 ds_pk 레코드의 형식을 수정합니다(자세한 내용은 AWS AppSync 개발자 안내서의 충돌 감지

및 동기화 참조). 활성화되면 populateIndexFields 항목 처리도 활성화됩니다. 이 필드는 선택 사항입니다.

populateIndexFields

customPartitionKey와 함께 활성화되면 델타 동기화 테이블, 특히 gsi_ds_pk 및 gsi_ds_sk 열의 각 레코드에 대해 새 항목을 생성하는 부울 값입니다. 자세한 내용은 AWS AppSync 개발자 안내서의 충돌 감지 및 동기화를 참조하세요. 이 필드는 선택 사항입니다.

DynamoDB에서 삭제된 항목은 자동으로 GraphQL 및 JSON 기본 유형으로 변환되며 컨텍스트 결과(context.result)에서 사용할 수 있습니다.

DynamoDB 유형 변환에 대한 자세한 내용은 유형 시스템(응답 매핑)을 참조하세요.

JavaScript 해석기에 대한 자세한 내용은 JavaScript 해석기 개요를 참조하세요.

예 1

다음 예제는 GraphQL 뮤테이션 deleteItem(id: ID!)을 위한 함수 요청 핸들러입니다. 이 ID를 가진 항목이 있으면 해당 항목은 삭제됩니다.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'DeleteItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}
```

예 2

다음 예제는 GraphQL 뮤테이션 deleteItem(id: ID!, expectedVersion: Int!)을 위한 함수 요청 핸들러입니다. 이 ID를 가진 항목이 있으면 해당 항목은 version 필드가 expectedVersion으로 설정된 경우에만 삭제됩니다.

```
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),
  };
}
```

DynamoDB `DeleteItem` API에 대한 자세한 내용은 DynamoDB API 문서를 참조하십시오.

쿼리

`Query` 요청 객체를 사용하면 AWS AppSync DynamoDB 해석기에 DynamoDB에 `Query` 요청을 하도록 지시하고 다음을 지정할 수 있습니다.

- 키 표현식
- 사용할 인덱스
- 모든 추가 필터
- 반환할 항목 수
- 일관된 읽기를 사용할지 여부
- 쿼리 방향(앞으로 또는 뒤로)
- 페이지 매김 토큰입니다

`Query` 요청 객체는 다음과 같은 구조입니다.

```
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 };
  };
};
```

필드는 다음과 같이 정의됩니다.

쿼리 필드

쿼리 필드 목록

`operation`

수행할 DynamoDB 작업입니다. `Query` DynamoDB 작업을 수행하려면 이 값을 `Query`으로 설정해야 합니다. 이 값은 필수입니다.

`query`

`query` 섹션에서는 DynamoDB에서 가져올 항목을 설명하는 키 조건 표현식을 지정할 수 있습니다. 키 조건 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB KeyConditions 문서를 참조하십시오. 이 섹션은 지정되어 있어야 합니다.

`expression`

쿼리 표현식. 이 필드는 지정되어 있어야 합니다.

`expressionNames`

표현식 속성 name 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 `expression`에 사용된 name 자리 표시자에 해당하고 값은 DynamoDB에 있는 항목의 속성 이름에 해당하는 문자열이어야 합니다. 이 필드는 선택 사항으로, `expression`에 사용된 표현식 속성인 name 자리 표시자의 대체 항목으로만 채워져야 합니다.

`expressionValues`

표현식 속성 value 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 `expression`에 사용되는 value 자리 표시자에 해당하고 값은 입력된 값이어야 합니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수

입니다. 이 필드는 선택 사항으로, `expression`에 사용된 표현식 속성인 value 자리 표시자의 대체 항목으로만 채워져야 합니다.

`filter`

반환되기 전 DynamoDB의 결과를 필터링하는 데 사용할 수 있는 추가 필터입니다. 필터에 대한 자세한 내용은 필터를 참조하십시오. 이 필드는 선택 사항입니다.

`index`

쿼리할 인덱스의 이름. DynamoDB 쿼리 작업을 사용하면 해시 키의 프라이머리 키 인덱스 이외에 로컬 보조 인덱스 및 글로벌 보조 인덱스를 검사할 수 있습니다. 지정하면 이 값은 DynamoDB에 지정한 인덱스를 쿼리하라고 지시합니다. 이 값을 생략하면 기본 키 인덱스가 쿼리됩니다.

`nextToken`

이전 쿼리를 지속하는 페이지 매김 토큰. 이 토큰은 이전 쿼리에서 얻습니다. 이 필드는 선택 사항입니다.

`limit`

평가할 최대 항목 수입니다(반드시 일치하는 항목 수는 아님). 이 필드는 선택 사항입니다.

`scanIndexForward`

앞으로 또는 뒤로 쿼리할지 여부를 나타내는 부울. 이 필드는 선택 사항으로, 기본값은 `true`입니다.

`consistentRead`

DynamoDB 쿼리 시 일관된 읽기를 사용할지 여부를 나타내는 부울. 이 필드는 선택 사항으로, 기본값은 `false`입니다.

`select`

기본적으로 AWS AppSync DynamoDB 해석기는 인덱스로 프로젝션된 속성만 반환합니다. 추가 속성이 필요한 경우 이 필드를 설정할 수 있습니다. 이 필드는 선택 사항입니다. 지원되는 값은 다음과 같습니다.

`ALL_ATTRIBUTES`

지정한 테이블 또는 인덱스에서 항목 속성을 모두 반환합니다. 로컬 보조 인덱스를 쿼리하는 경우 DynamoDB는 인덱스의 일치하는 각 항목에 대해 상위 테이블의 전체 항목을 가져옵니다. 인덱스가 모든 항목 속성을 프로젝션하도록 구성된 경우, 모든 데이터를 로컬의 보조 인덱스에서 얻을 수 있기 때문에 가져올 필요가 없습니다.

ALL_PROJECTED_ATTRIBUTES

인덱스를 쿼리하는 경우에만 허용됩니다. 인덱스로 프로젝션된 모든 속성을 가져옵니다. 모든 속성을 프로젝션하도록 인덱스가 구성된 경우 이 반환 값은 ALL_ATTRIBUTES를 지정하는 것과 동일합니다.

SPECIFIC_ATTRIBUTES

projection의 expression에 나열된 속성만 반환합니다. 이 반환 값은 Select에 대한 값을 지정하지 않고 projection의 expression을 지정하는 것과 같습니다.

projection

DynamoDB 작업에서 반환할 속성을 지정하는 데 사용되는 프로젝션입니다. 프로젝션에 대한 자세한 내용은 프로젝션을 참조하세요. 이 필드는 선택 사항입니다.

DynamoDB의 결과는 자동으로 GraphQL 및 JSON 기본 유형으로 변환되며 컨텍스트 결과(context.result)에서 사용할 수 있습니다.

DynamoDB 유형 변환에 대한 자세한 내용은 유형 시스템(응답 매핑)을 참조하세요.

JavaScript 해석기에 대한 자세한 내용은 JavaScript 해석기 개요를 참조하세요.

결과의 구조는 다음과 같습니다.

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

필드는 다음과 같이 정의됩니다.

items

DynamoDB 쿼리에서 반환하는 항목이 포함된 목록.

nextToken

더 많은 결과가 있을 수 있는 경우 nextToken에는 다른 요청에 사용할 수 있는 페이지 매김 토큰이 포함됩니다. AWS AppSync 에서는 DynamoDB에서 반환된 페이지 매김 토큰을 암호화하고 난독화합니다. 따라서 테이블 데이터가 호출자에게 실수로 유출되는 일이 없습니다. 페이지 매김 토큰은 여러 해석기 간에 사용할 수 없습니다.

scannedCount

필터 표현식(있는 경우)을 적용하기 전에 쿼리 조건 표현식과 일치했던 항목 수

예

다음 예제는 GraphQL 쿼리 `getPosts(owner: ID!)`를 위한 함수 요청 핸들러입니다.

이 예에서는 테이블의 글로벌 보조 인덱스를 쿼리해 지정한 ID가 소유한 게시물을 모두 반환합니다.

```
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',
  };
}
```

DynamoDB `Query` API에 대한 자세한 내용은 DynamoDB API 문서를 참조하십시오.

스캔

`Scan` 요청을 사용하면 AWS AppSync DynamoDB 함수에 DynamoDB에 `Scan` 요청을 하도록 지시하고 다음을 지정할 수 있습니다.

- 결과를 제외하는 필터
- 사용할 인덱스
- 반환할 항목 수
- 일관된 읽기를 사용할지 여부
- 페이지 매김 토큰입니다
- 병렬 스캔

`Scan` 요청 객체는 다음과 같은 구조입니다.

```
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 };
  };
};
```

필드는 다음과 같이 정의됩니다.

Scan 필드

Scan 필드 목록

`operation`

수행할 DynamoDB 작업입니다. `Scan` DynamoDB 작업을 수행하려면 이 값을 `Scan`으로 설정해야 합니다. 이 값은 필수입니다.

`filter`

반환되기 전 DynamoDB의 결과를 필터링하는 데 사용할 수 있는 필터. 필터에 대한 자세한 내용은 필터를 참조하십시오. 이 필드는 선택 사항입니다.

`index`

쿼리할 인덱스의 이름. DynamoDB 쿼리 작업을 사용하면 해시 키의 프라이머리 키 인덱스 이외에 로컬 보조 인덱스 및 글로벌 보조 인덱스를 검사할 수 있습니다. 지정하면 이 값은 DynamoDB에 지정한 인덱스를 쿼리하라고 지시합니다. 이 값을 생략하면 기본 키 인덱스가 쿼리됩니다.

`limit`

한 번에 평가할 수 있는 최대 항목 수입니다. 이 필드는 선택 사항입니다.

consistentRead

DynamoDB 쿼리 시 일관된 읽기를 사용할지 여부를 나타내는 부울. 이 필드는 선택 사항으로, 기본값은 `false`입니다.

nextToken

이전 쿼리를 지속하는 페이지 매김 토큰. 이 토큰은 이전 쿼리에서 얻습니다. 이 필드는 선택 사항입니다.

select

기본적으로 AWS AppSync DynamoDB 함수는 인덱스로 프로젝션된 속성만 반환합니다. 추가 속성이 필요한 경우 이 필드를 설정할 수 있습니다. 이 필드는 선택 사항입니다. 지원되는 값은 다음과 같습니다.

ALL_ATTRIBUTES

지정한 테이블 또는 인덱스에서 항목 속성을 모두 반환합니다. 로컬 보조 인덱스를 쿼리하는 경우 DynamoDB는 인덱스의 일치하는 각 항목에 대해 상위 테이블의 전체 항목을 가져옵니다. 인덱스가 모든 항목 속성을 프로젝션하도록 구성된 경우, 모든 데이터를 로컬의 보조 인덱스에서 얻을 수 있기 때문에 가져올 필요가 없습니다.

ALL_PROJECTED_ATTRIBUTES

인덱스를 쿼리하는 경우에만 허용됩니다. 인덱스로 프로젝션된 모든 속성을 가져옵니다. 모든 속성을 프로젝션하도록 인덱스가 구성된 경우 이 반환 값은 `ALL_ATTRIBUTES`를 지정하는 것과 동일합니다.

SPECIFIC_ATTRIBUTES

`projection`의 `expression`에 나열된 속성만 반환합니다. 이 반환 값은 `Select`에 대한 값을 지정하지 않고 `projection`의 `expression`을 지정하는 것과 같습니다.

totalSegments

병렬 스캔을 수행할 때 테이블을 분할할 기준이 될 세그먼트 수. 이 필드는 선택 사항이지만 `segment`가 지정된 경우에는 반드시 지정해야 합니다.

segment

병렬 스캔을 수행할 때 작업의 테이블 세그먼트. 이 필드는 선택 사항이지만 `totalSegments`가 지정된 경우에는 반드시 지정해야 합니다.

`projection`

DynamoDB 작업에서 반환할 속성을 지정하는 데 사용되는 프로젝션입니다. 프로젝션에 대한 자세한 내용은 프로젝션을 참조하세요. 이 필드는 선택 사항입니다.

DynamoDB 스캔에서 반환하는 결과는 자동으로 GraphQL 및 JSON 기본 유형으로 변환되며 컨텍스트 결과(`context.result`)에서 사용할 수 있습니다.

DynamoDB 유형 변환에 대한 자세한 내용은 유형 시스템(응답 매핑)을 참조하세요.

JavaScript 해석기에 대한 자세한 내용은 JavaScript 해석기 개요를 참조하세요.

결과의 구조는 다음과 같습니다.

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

필드는 다음과 같이 정의됩니다.

`items`

DynamoDB 스캔에서 반환하는 항목이 포함된 목록

`nextToken`

더 많은 결과가 있을 수 있는 경우 `nextToken`에는 다른 요청에 사용할 수 있는 페이지 매김 토큰이 포함됩니다. AppSync에서는 DynamoDB에서 반환된 페이지 매김 토큰을 암호화하고 난독화합니다. 따라서 테이블 데이터가 호출자에게 실수로 유출되는 일이 없습니다. 또한 페이지 매김 토큰은 여러 함수 또는 해석기 간에 사용할 수 없습니다.

`scannedCount`

필터 표현식(있는 경우)을 적용하기 전에 DynamoDB에서 검색한 항목 수.

예 1

다음 예제는 GraphQL 쿼리 `allPosts`를 위한 함수 요청 핸들러입니다.

이 예에서는 테이블의 항목을 모두 반환합니다.

```
export function request(ctx) {
  return { operation: 'Scan' };
}
```

예 2

다음 예제는 GraphQL 쿼리 `postsMatching(title: String!)`을 위한 함수 요청 핸들러입니다.

이 예에서는 제목이 `title` 인수로 시작하는 경우 테이블의 모든 항목이 반환됩니다.

```
export function request(ctx) {
  const { title } = ctx.args;
  const filter = { filter: { beginsWith: title } };
  return {
    operation: 'Scan',
    filter: JSON.parse(util.transform.toDynamoDBFilterExpression(filter)),
  };
}
```

DynamoDB `Scan` API에 대한 자세한 내용은 DynamoDB API 문서를 참조하십시오.

동기화

Sync 요청 객체를 사용하면 DynamoDB 테이블에서 모든 결과를 가져온 다음, 마지막 쿼리(델타 업데이트) 이후에 변경된 데이터만 수신할 수 있습니다. Sync 요청은 버전이 지정된 DynamoDB 데이터 소스에만 수행할 수 있습니다. 지정할 수 있는 셰이프는 다음과 같습니다.

- 결과를 제외하는 필터
- 반환할 항목 수
- 페이지 매김 토큰
- 마지막 Sync 작업이 시작된 경우

Sync 요청 객체는 다음과 같은 구조입니다.

```
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 };
  };
};
```

필드는 다음과 같이 정의됩니다.

Sync 필드

Sync 필드 목록

`operation`

수행할 DynamoDB 작업입니다. Sync 작업을 수행하려면 이 값을 Sync으로 설정해야 합니다. 이 값은 필수입니다.

`filter`

반환되기 전 DynamoDB의 결과를 필터링하는 데 사용할 수 있는 필터. 필터에 대한 자세한 내용은 필터를 참조하십시오. 이 필드는 선택 사항입니다.

`limit`

한 번에 평가올 수 있는 최대 항목 수입니다. 이 필드는 선택 사항입니다. 생략된 경우, 기본 제한이 `100`개 항목으로 설정됩니다. 이 필드의 최대값은 `1000`개 항목입니다.

`nextToken`

이전 쿼리를 지속하는 페이지 매김 토큰. 이 토큰은 이전 쿼리에서 얻습니다. 이 필드는 선택 사항입니다.

`lastSync`

마지막으로 성공한 Sync 작업이 시작되었던 시간(epoch 밀리초)입니다. 지정된 경우, `lastSync` 이후에 변경된 항목만 반환됩니다. 이 필드는 선택 사항이며 초기 Sync 작업에서 모든 페이지를 가져온 이후에만 채워져야 합니다. 생략된 경우, 기본 테이블의 결과가 반환되고, 그렇지 않으면 델타 테이블의 결과가 반환됩니다.

basePartitionKey

Sync 작업을 수행할 때 사용되는 기본 테이블의 파티션 키입니다. 이 필드를 사용하면 테이블에서 사용자 지정 파티션 키를 활용할 때 Sync 작업을 수행할 수 있습니다. 이 필드는 선택 사항입니다.

deltaIndexName

Sync 작업에 사용되는 인덱스. 이 인덱스는 테이블에서 사용자 지정 파티션 키를 사용할 때 전체 델타 저장소 테이블에서 Sync 작업을 활성화하는 데 필요합니다. Sync 작업은 GSI(`gsi_ds_pk` 및 `gsi_ds_sk`에서 생성)에서 수행됩니다. 이 필드는 선택 사항입니다.

DynamoDB 동기화에서 반환하는 결과는 자동으로 GraphQL 및 JSON 기본 유형으로 변환되며 컨텍스트 결과(`context.result`)에서 사용할 수 있습니다.

DynamoDB 유형 변환에 대한 자세한 내용은 유형 시스템(응답 매핑)을 참조하세요.

JavaScript 해석기에 대한 자세한 내용은 JavaScript 해석기 개요를 참조하세요.

결과의 구조는 다음과 같습니다.

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10,
    startedAt = 1550000000000
}
```

필드는 다음과 같이 정의됩니다.

items

동기화에서 반환하는 항목이 포함된 목록.

nextToken

더 많은 결과가 있을 수 있는 경우 `nextToken`에는 다른 요청에 사용할 수 있는 페이지 매김 토큰이 포함됩니다. AppSync에서는 DynamoDB에서 반환된 페이지 매김 토큰을 암호화하고 난독화합니다. 따라서 테이블 데이터가 호출자에게 실수로 유출되는 일이 없습니다. 또한 페이지 매김 토큰은 여러 함수 또는 해석기 간에 사용할 수 없습니다.

scannedCount

필터 표현식(있는 경우)을 적용하기 전에 DynamoDB에서 검색한 항목 수.

`startedAt`

로컬로 저장하고 다른 요청에 `lastSync` 인수로 사용할 수 있는 동기화 작업이 시작된 시간(Epoch 밀리초)입니다. 페이지 매김 토큰이 요청에 포함된 경우, 이 값은 결과의 첫 페이지에 대한 요청에 의해 반환된 값과 동일합니다.

예 1

다음 예제는 GraphQL 쿼리 `syncPosts(nextToken: String, lastSync: AWSTimestamp)`을 위한 함수 요청 핸들러입니다.

이 예에서 `lastSync`가 생략된 경우, 기본 테이블의 모든 항목이 반환됩니다. `lastSync`가 제공된 경우 `lastSync` 이후 변경된 델타 동기화 테이블의 항목만 반환됩니다.

```
export function request(ctx) {
  const { nextToken, lastSync } = ctx.args;
  return { operation: 'Sync', limit: 100, nextToken, lastSync };
}
```

BatchGetItem

`BatchGetItem` 요청 객체를 사용하면 AWS AppSync DynamoDB 함수에 DynamoDB에 `BatchGetItem` 요청을 하여 잠재적으로 여러 테이블에 걸쳐 여러 항목을 검색하도록 지시할 수 있습니다. 이 요청 객체에 대해 다음을 지정해야 합니다.

- 항목을 가져올 테이블 이름
- 각 테이블에서 가져올 항목의 키

DynamoDB `BatchGetItem` 한도가 적용되고 표현식 없음이 제공될 수 있습니다.

`BatchGetItem` 요청 객체는 다음과 같은 구조입니다.

```
type DynamoDBBatchGetItemRequest = {
  operation: 'BatchGetItem';
  tables: {
    [tableName: string]: {
      keys: { [key: string]: any }[];
      consistentRead?: boolean;
      projection?: {
```

```
        expression: string;
        expressionNames?: { [key: string]: string };
      };
    };
  };
};
```

필드는 다음과 같이 정의됩니다.

BatchGetItem 필드

BatchGetItem 필드 목록

`operation`

수행할 DynamoDB 작업입니다. `BatchGetItem` DynamoDB 작업을 수행하려면 이 값을 `BatchGetItem`으로 설정해야 합니다. 이 값은 필수입니다.

`tables`

항목을 가져올 DynamoDB 테이블입니다. 이 값은 테이블 이름이 맵의 키로 지정되는 맵입니다. 테이블이 하나 이상 제공되어야 합니다. 이 `tables` 값은 필수입니다.

`keys`

검색할 항목의 프라이머리 키를 나타내는 DynamoDB 키 목록입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요.

`consistentRead`

GetItem 작업을 실행할 때 일관된 읽기를 사용할지 여부. 이 값은 선택 사항으로, 기본값은 false입니다.

`projection`

DynamoDB 작업에서 반환할 속성을 지정하는 데 사용되는 프로젝션입니다. 프로젝션에 대한 자세한 내용은 프로젝션을 참조하세요. 이 필드는 선택 사항입니다.

기억해야 할 내용:

- 테이블에서 항목을 가져오지 못하면 해당 테이블의 데이터 블록에 null 요소가 나타납니다.
- 호출 결과는 요청 객체 내에 제공된 순서에 따라 테이블별로 정렬됩니다.

- BatchGetItem 내부의 각 Get 명령은 원자성이지만 배치는 부분적으로 처리될 수 있습니다. 오류로 인해 배치가 부분적으로 처리된 경우 처리되지 않은 키는 unprocessedKeys 블록 내에 호출 결과의 일부로 반환됩니다.
- BatchGetItem은 100개의 키로 제한됩니다.

다음은 함수 요청 핸들러 예시입니다.

```
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 })],
    },
  };
}
```

ctx.result에서 사용할 수 있는 호출 결과는 다음과 같습니다.

```
{
   "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.error에는 오류에 대한 세부 정보가 포함됩니다. 함수 요청 객체의 결과에 제공된 키 data, unprocessedKeys 및 각 테이블 키는 간접 호출 결과에 존재하는 것이 보장됩니다. 삭제된 항목은 데이터 블록에 나타납니다. 처리되지 않은 항목은 데이터 블록 내에서 null로 표시되고 unprocessedKeys 블록 내에 배치됩니다.

BatchDeleteItem

BatchDeleteItem 요청 객체를 사용하면 AWS AppSync DynamoDB 함수에 DynamoDB에 BatchWriteItem 요청을 하여 잠재적으로 여러 테이블에 걸쳐 여러 항목을 삭제하도록 지시할 수 있습니다. 이 요청 객체에 대해 다음을 지정해야 합니다.

- 항목을 삭제할 테이블 이름
- 각 테이블에서 삭제할 항목의 키

DynamoDB BatchWriteItem 한도가 적용되고 표현식 없음이 제공될 수 있습니다.

BatchDeleteItem 요청 객체는 다음과 같은 구조입니다.

```
type DynamoDBBatchDeleteItemRequest = {
  operation: 'BatchDeleteItem';
  tables: {
    [tableName: string]: { [key: string]: any }[];
  };
};
```

필드는 다음과 같이 정의됩니다.

BatchDeleteItem 필드

BatchDeleteItem 필드 목록

operation

수행할 DynamoDB 작업입니다. BatchDeleteItem DynamoDB 작업을 수행하려면 이 값을 BatchDeleteItem으로 설정해야 합니다. 이 값은 필수입니다.

`tables`

항목을 삭제할 DynamoDB 테이블입니다. 각 테이블은 삭제할 항목의 프라이머리 키를 나타내는 DynamoDB 키 목록입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 테이블이 하나 이상 제공되어야 합니다. `tables` 값이 필요합니다.

기억해야 할 내용:

- `DeleteItem` 작업과 달리 완전히 삭제된 항목은 응답에서 반환되지 않습니다. 전달된 키만 반환됩니다.
- 테이블에서 항목을 삭제하지 못하면 해당 테이블의 데이터 블록에 null 요소가 나타납니다.
- 호출 결과는 요청 객체 내에 제공된 순서에 따라 테이블별로 정렬됩니다.
- `BatchDeleteItem` 내부의 각 `Delete` 명령은 원자성입니다. 그러나 배치는 부분적으로 처리될 수 있습니다. 오류로 인해 배치가 부분적으로 처리된 경우 처리되지 않은 키는 unprocessedKeys 블록 내에 호출 결과의 일부로 반환됩니다.
- `BatchDeleteItem`은 25개의 키로 제한됩니다.

다음은 함수 요청 핸들러 예시입니다.

```
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 })],
    },
  };
}
```

`ctx.result`에서 사용할 수 있는 호출 결과는 다음과 같습니다.

```
{
   "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.error에는 오류에 대한 세부 정보가 포함됩니다. 함수 요청 객체에 제공된 키 data, unprocessedKeys 및 각 테이블 키는 간접 호출 결과에 존재하는 것이 보장됩니다. 삭제된 항목은 데이터 블록에 나타납니다. 처리되지 않은 항목은 데이터 블록 내에서 null로 표시되고 unprocessedKeys 블록 내에 배치됩니다.

BatchPutItem

BatchPutItem 요청 객체를 사용하면 AWS AppSync DynamoDB 함수에 DynamoDB에 BatchWriteItem 요청을 하여 잠재적으로 여러 테이블에 걸쳐 여러 항목을 입력하도록 지시할 수 있습니다. 이 요청 객체에 대해 다음을 지정해야 합니다.

- 항목을 저장할 테이블 이름
- 각 테이블에 저장할 전체 항목

DynamoDB BatchWriteItem 한도가 적용되고 표현식 없음이 제공될 수 있습니다.

BatchPutItem 요청 객체는 다음과 같은 구조입니다.

```
type DynamoDBBatchPutItemRequest = {
  operation: 'BatchPutItem';
  tables: {
    [tableName: string]: { [key: string]: any}[];
  };
```

```
};
```

필드는 다음과 같이 정의됩니다.

BatchPutItem 필드

BatchPutItem 필드 목록

`operation`

수행할 DynamoDB 작업입니다. `BatchPutItem` DynamoDB 작업을 수행하려면 이 값을 `BatchPutItem`으로 설정해야 합니다. 이 값은 필수입니다.

`tables`

항목을 넣을 DynamoDB 테이블입니다. 각 테이블 항목은 이 특정 테이블에 삽입할 DynamoDB 항목 목록을 나타냅니다. 테이블이 하나 이상 제공되어야 합니다. 이 값은 필수입니다.

기억해야 할 내용:

- 성공하면, 완전히 삽입된 항목이 응답에서 반환됩니다.
- 테이블에 항목이 삽입되지 못하면 해당 테이블의 데이터 블록에 null 요소가 표시됩니다.
- 삽입된 항목은 요청 객체 내에 제공된 순서에 따라 테이블별로 정렬됩니다.
- `BatchPutItem` 내부의 각 `Put` 명령은 원자성이지만 배치는 부분적으로 처리될 수 있습니다. 오류로 인해 배치가 부분적으로 처리된 경우 처리되지 않은 키는 unprocessedKeys 블록 내에 호출 결과의 일부로 반환됩니다.
- `BatchPutItem`은 25개의 항목으로 제한됩니다.

다음은 함수 요청 핸들러 예시입니다.

```
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 })],
    },
```

```
  };
}
```

ctx.result에서 사용할 수 있는 호출 결과는 다음과 같습니다.

```
{
   "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.error에는 오류에 대한 세부 정보가 포함됩니다. 키 data, unprocessedItems 및 요청 객체에서 제공된 각 테이블 키는 호출 결과에 표시됩니다. 삽입된 항목은 데이터 블록에 나타납니다. 처리되지 않은 항목은 데이터 블록 내에서 null로 표시되고 unprocessedItems 블록 내에 배치됩니다.

TransactGetItems

TransactGetItems 요청 객체를 사용하면 AWS AppSync DynamoDB 함수에 DynamoDB에 TransactGetItems 요청을 하여 잠재적으로 여러 테이블에 걸쳐 여러 항목을 검색하도록 지시할 수 있습니다. 이 요청 객체에 대해 다음을 지정해야 합니다.

- 항목을 가져올 각 요청 항목의 테이블 이름

- 각 테이블에서 가져올 각 요청 항목의 키

DynamoDB `TransactGetItems` 한도가 적용되고 표현식 없음이 제공될 수 있습니다.

`TransactGetItems` 요청 객체는 다음과 같은 구조입니다.

```
type DynamoDBTransactGetItemsRequest = {
  operation: 'TransactGetItems';
  transactItems: { table: string; key: { [key: string]: any }; projection?:
 { expression: string; expressionNames?: { [key: string]: string }; }[];
  };
};
```

필드는 다음과 같이 정의됩니다.

TransactGetItems 필드

TransactGetItems 필드 목록

`operation`

수행할 DynamoDB 작업입니다. `TransactGetItems` DynamoDB 작업을 수행하려면 이 값을 `TransactGetItems`으로 설정해야 합니다. 이 값은 필수입니다.

`transactItems`

포함할 요청 항목입니다. 이 값은 요청 항목의 배열입니다. 하나 이상의 요청 항목이 제공되어야 합니다. 이 `transactItems` 값은 필수입니다.

`table`

항목을 가져올 DynamoDB 테이블입니다. 이 값은 테이블 이름의 문자열입니다. 이 `table` 값은 필수입니다.

`key`

검색할 항목의 프라이머리 키를 나타내는 DynamoDB 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요.

`projection`

DynamoDB 작업에서 반환할 속성을 지정하는 데 사용되는 프로젝션입니다. 프로젝션에 대한 자세한 내용은 프로젝션을 참조하세요. 이 필드는 선택 사항입니다.

기억해야 할 내용:

- 트랜잭션이 성공하면 `items` 블록에서 가져오는 항목의 순서는 요청 항목의 순서와 동일합니다.
- 트랜잭션은 전부 또는 전무 방식으로 수행됩니다. 요청 항목에 오류가 발생하면 전체 트랜잭션이 수행되지 않고 오류 세부 정보가 반환됩니다.
- 가져올 수 없는 요청 항목은 오류가 아닙니다. 대신 해당 위치의 항목 블록에 null 요소가 나타납니다.
- 트랜잭션의 오류가 TransactionCanceledException인 경우 `cancellationReasons` 블록이 채워집니다. `cancellationReasons` 블록 내의 취소 사유 순서는 요청 항목의 순서와 동일합니다.
- `TransactGetItems`는 25개의 요청 항목으로 제한됩니다.

다음은 함수 요청 핸들러 예시입니다.

```
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 }),
      },
    ],
  };
}
```

트랜잭션이 성공하고 첫 번째 요청된 항목만 검색되는 경우 `ctx.result`에서 사용할 수 있는 호출 결과는 다음과 같습니다.

```
{
    "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
}
```

첫 번째 요청 항목으로 인한 TransactionCanceledException 때문에 트랜잭션이 실패할 경우 `ctx.result`에서 사용할 수 있는 호출 결과는 다음과 같습니다.

```
{
    "items": null,
    "cancellationReasons": [
        {
            "type":"Sample error type",
            "message":"Sample error message"
        },
        {
            "type":"None",
            "message":"None"
        }
    ]
}
```

`ctx.error`에는 오류에 대한 세부 정보가 포함됩니다. 키 항목 과 cancellationReasons는 `ctx.result`에 있습니다.

TransactWriteItems

`TransactWriteItems` 요청 객체를 사용하면 AWS AppSync DynamoDB 함수에 DynamoDB에 `TransactWriteItems` 요청을 하여 잠재적으로 여러 테이블에 여러 항목을 쓰도록 지시할 수 있습니다. 이 요청 객체에 대해 다음을 지정해야 합니다.

- 각 요청 항목의 대상 테이블 이름
- 수행할 각 요청 항목의 작업입니다. 지원되는 작업에는 PutItem, UpdateItem, DeleteItem, ConditionCheck의 네 가지 유형이 있습니다.
- 작성할 각 요청 항목의 키

DynamoDB `TransactWriteItems` 제한이 적용됩니다.

TransactWriteItems 요청 객체는 다음과 같은 구조입니다.

```
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;
};
```

TransactWriteItems 필드

TransactWriteItems 필드 목록

필드는 다음과 같이 정의됩니다.

`operation`

수행할 DynamoDB 작업입니다. `TransactWriteItems` DynamoDB 작업을 수행하려면 이 값을 `TransactWriteItems`으로 설정해야 합니다. 이 값은 필수입니다.

`transactItems`

포함할 요청 항목입니다. 이 값은 요청 항목의 배열입니다. 하나 이상의 요청 항목이 제공되어야 합니다. 이 `transactItems` 값은 필수입니다.

`PutItem`의 필드는 다음과 같이 정의됩니다.

`table`

대상 DynamoDB 테이블입니다. 이 값은 테이블 이름의 문자열입니다. 이 `table` 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `PutItem` DynamoDB 작업을 수행하려면 이 값을 `PutItem`으로 설정해야 합니다. 이 값은 필수입니다.

`key`

입력할 항목의 프라이머리 키를 나타내는 DynamoDB 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

`attributeValues`

DynamoDB에 저장할 항목의 나머지 속성. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 필드는 선택 사항입니다.

`condition`

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건을 지정하지 않으면 `PutItem` 요청이 해당 항목에 대한 기존 입력을 덮어씁니다. 조건 검사에 실패할 경우 기존 항목을 다시 가져올지 여부를 지정할 수 있습니다. 트랜잭션 조건에 대한 자세한 내용은 트랜잭션 조건 표현식을 참조하세요. 이 값은 선택 사항입니다.

`UpdateItem`의 필드는 다음과 같이 정의됩니다.

`table`

업데이트할 DynamoDB 테이블입니다. 이 값은 테이블 이름의 문자열입니다. 이 `table` 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `UpdateItem` DynamoDB 작업을 수행하려면 이 값을 `UpdateItem`으로 설정해야 합니다. 이 값은 필수입니다.

`key`

업데이트할 항목의 프라이머리 키를 나타내는 DynamoDB 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

`update`

`update` 섹션에서는 DynamoDB의 항목 업데이트 방법을 설명하는 업데이트 표현식을 지정합니다. 업데이트 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB UpdateExpressions 문서를 참조하십시오. 이 섹션은 필수입니다.

`condition`

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건을 지정하지 않으면 `UpdateItem` 요청이 현재 상태와 상관 없이 기존 항목을 업데이트합니다. 조건 검사에 실패할 경우 기존 항목을 다시 가져올지 여부를 지정할 수 있습니다. 트랜잭션 조건에 대한 자세한 내용은 트랜잭션 조건 표현식을 참조하세요. 이 값은 선택 사항입니다.

`DeleteItem`의 필드는 다음과 같이 정의됩니다.

`table`

항목을 삭제할 DynamoDB 테이블입니다. 이 값은 테이블 이름의 문자열입니다. 이 `table` 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `DeleteItem` DynamoDB 작업을 수행하려면 이 값을 `DeleteItem`으로 설정해야 합니다. 이 값은 필수입니다.

key

삭제할 항목의 프라이머리 키를 나타내는 DynamoDB 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

condition

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건을 지정하지 않으면 `DeleteItem` 요청이 현재 상태와 상관 없이 항목을 삭제합니다. 조건 검사에 실패할 경우 기존 항목을 다시 가져올지 여부를 지정할 수 있습니다. 트랜잭션 조건에 대한 자세한 내용은 트랜잭션 조건 표현식을 참조하세요. 이 값은 선택 사항입니다.

`ConditionCheck`의 필드는 다음과 같이 정의됩니다.

table

조건을 검사할 DynamoDB 테이블입니다. 이 값은 테이블 이름의 문자열입니다. 이 `table` 값은 필수입니다.

operation

수행할 DynamoDB 작업입니다. `ConditionCheck` DynamoDB 작업을 수행하려면 이 값을 `ConditionCheck`으로 설정해야 합니다. 이 값은 필수입니다.

key

조건을 확인할 항목의 프라이머리 키를 나타내는 DynamoDB 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

condition

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건 검사에 실패할 경우 기존 항목을 다시 가져올지 여부를 지정할 수 있습니다. 트랜잭션 조건에 대한 자세한 내용은 트랜잭션 조건 표현식을 참조하세요. 이 값은 필수입니다.

기억해야 할 내용:

- 성공할 경우 요청 항목의 키만 응답에 반환됩니다. 키 순서는 요청 항목의 순서와 동일합니다.
- 트랜잭션은 전부 또는 전무 방식으로 수행됩니다. 요청 항목에 오류가 발생하면 전체 트랜잭션이 수행되지 않고 오류 세부 정보가 반환됩니다.

- 두 개의 요청 항목이 동일한 항목을 대상으로 할 수 없습니다. 그렇지 않으면 TransactionCanceledException 오류가 발생합니다.
- 트랜잭션의 오류가 TransactionCanceledException인 경우 `cancellationReasons` 블록이 채워집니다. 요청 항목의 조건 검사에 실패한 동시에 `returnValuesOnConditionCheckFailure`를 `false`로 지정하지 않은 경우, 테이블에 있는 항목이 검색되고 `item`에서 `cancellationReasons` 블록의 해당 위치에 저장됩니다.
- `TransactWriteItems`는 25개의 요청 항목으로 제한됩니다.

다음은 함수 요청 핸들러 예시입니다.

```
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 }),
        },
      },
    ],
  };
}
```

트랜잭션이 성공하면 `ctx.result`에서 사용할 수 있는 호출 결과는 다음과 같습니다.

```
{
    "keys": [
       // Key of the PutItem request
       {
           "post_id": "p1",
       },
       // Key of the UpdateItem request
       {
           "author_id": "a1"
       }
    ],
    "cancellationReasons": null
}
```

PutItem 요청의 조건 검사 실패로 인해 트랜잭션이 실패할 경우 ctx.result에서 사용할 수 있는 간접 호출 결과는 다음과 같습니다.

```
{
    "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.error에는 오류에 대한 세부 정보가 포함됩니다. 키 및 cancellationReasons 키는 ctx.result에 있습니다.

형식 시스템(요청 매핑)

AWS AppSync DynamoDB 함수를 사용하여 DynamoDB 테이블을 호출하는 경우 AWS AppSync 에서는 해당 호출에 사용된 각 값의 형식을 알고 있어야 합니다. 이는 DynamoDB가 GraphQL 또는 JSON보다 더 많은 유형 기본 형식(예: 세트 및 이진 데이터)을 지원하기 때문입니다. AWS AppSync는 GraphQL과 DynamoDB 간에 변환할 때 몇 가지 힌트가 필요합니다. 그렇지 않으면 테이블에서 데이터가 구성되는 방식에 대해 몇 가지 가정을 해야 합니다.

DynamoDB 데이터 유형에 대한 자세한 내용은 DynamoDB 데이터 유형 설명자 및 데이터 유형 설명서를 참조하세요.

DynamoDB 값은 단일 키-값 페어가 포함된 JSON 객체로 나타납니다. 여기서 키는 DynamoDB 형식을 지정하고 값은 값 자체를 지정합니다. 다음 예에서 키 `S`는 값이 문자열임을 나타내고 값 `identifier`는 문자열 값 자체를 나타냅니다.

```
{ "S" : "identifier" }
```

JSON 객체는 키-값 페어를 두 개 이상 포함할 수 없습니다. 키-값 페어를 두 개 이상 지정하면 요청 객체가 구문 분석되지 않습니다.

요청 객체에서 값을 지정해야 하는 모든 곳에 DynamoDB 값을 사용할 수 있습니다. 이렇게 해야 하는 위치로는 `key` 및 `attributeValue` 섹션과 표현식 섹션 중 `expressionValues` 섹션이 있습니다. 다음 예에서는 DynamoDB 문자열 값 `identifier`를 `key` 섹션(`GetItem` 요청 객체문서)의 `id` 필드에 할당합니다.

```
"key" : {
   "id" : { "S" : "identifier" }
}
```

지원되는 유형

AWS AppSync는 다음 DynamoDB 스칼라, 문서 및 집합 형식을 지원합니다.

String 형식 **S**

단일 문자열 값. DynamoDB String 값은 다음과 같이 표시됩니다.

```
{ "S" : "some string" }
```

사용 예제는 다음과 같습니다.

```
"key" : {
   "id" : { "S" : "some string" }
}
```

String set 형식 **SS**

문자열 값 집합. DynamoDB String Set 값은 다음과 같이 표시됩니다.

```
{ "SS" : [ "first value", "second value", ... ] }
```

사용 예제는 다음과 같습니다.

```
"attributeValues" : {
   "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
}
```

Number 형식 **N**

단일 숫자 값. DynamoDB Number 값은 다음과 같이 표시됩니다.

```
{ "N" : 1234 }
```

사용 예제는 다음과 같습니다.

```
"expressionValues" : {
   ":expectedVersion" : { "N" : 1 }
}
```

Number set 형식 **NS**

숫자 값 집합. DynamoDB Number Set 값은 다음과 같이 표시됩니다.

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

사용 예제는 다음과 같습니다.

```
"attributeValues" : {
   "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }
}
```

Binary 형식 **B**

이진 값. DynamoDB Binary 값은 다음과 같이 표시됩니다.

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

값은 실제로 문자열이며, 여기서 문자열은 이진 데이터의 base64 인코딩 표현입니다. AWS AppSync는 이 문자열을 DynamoDB로 보내기 전에 이 문자열을 이진 값으로 다시 디코딩합니다. AWS AppSync는 RFC 2045에 정의된 base64 디코딩 체계를 사용합니다. base64 알파벳에 없는 모든 문자는 무시됩니다.

사용 예제는 다음과 같습니다.

```
"attributeValues" : {
   "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

Binary set 형식 **BS**

이진 값 집합. DynamoDB Binary Set 값은 다음과 같이 표시됩니다.

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

값은 실제로 문자열이며, 여기서 문자열은 이진 데이터의 base64 인코딩 표현입니다. AWS AppSync는 이 문자열을 DynamoDB로 보내기 전에 이 문자열을 이진 값으로 다시 디코딩합니다. AWS AppSync는 RFC 2045에 정의된 base64 디코딩 체계를 사용합니다. base64 알파벳에 없는 모든 문자는 무시됩니다.

사용 예제는 다음과 같습니다.

```
"attributeValues" : {
   "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }
}
```

Boolean 형식 **BOOL**

부울 값. DynamoDB Boolean 값은 다음과 같이 표시됩니다.

```
{ "BOOL" : true }
```

`true` 및 `false`만 유효한 값입니다.

사용 예제는 다음과 같습니다.

```
"attributeValues" : {
   "orderComplete" : { "BOOL" : false }
}
```

List 형식 **L**

기타 지원되는 DynamoDB 값 목록입니다. DynamoDB List 값은 다음과 같이 표시됩니다.

```
{ "L" : [ ... ] }
```

값은 복합 값으로, 목록에는 지원되는 DynamoDB 값(다른 목록 포함)이 0개 이상 포함될 수 있습니다. 또한 목록에는 여러 형식이 혼합되어 포함될 수 있습니다.

사용 예제는 다음과 같습니다.

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

Map 형식 **M**

지원되는 다른 DynamoDB 값의 키-값 페어로 구성된 순서 없는 모음을 나타냅니다. DynamoDB Map 값은 다음과 같이 표시됩니다.

```
{ "M" : { ... } }
```

맵에는 키-값 페어가 0개 이상 포함될 수 있습니다. 키는 문자열이어야 하며 값은 지원되는 모든 DynamoDB 값(다른 맵 포함)일 수 있습니다. 또한 맵에는 여러 형식이 혼합되어 포함될 수 있습니다.

사용 예제는 다음과 같습니다.

```
{ "M" : {
```

```
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

Null 형식 **NULL**

null 값. DynamoDB Null 값은 다음과 같이 표시됩니다.

```
{ "NULL" : null }
```

사용 예제는 다음과 같습니다.

```
"attributeValues" : {
   "phoneNumbers" : { "NULL" : null }
}
```

각 형식에 대한 자세한 내용은 DynamoDB 문서를 참조하십시오.

형식 시스템(응답 매핑)

DynamoDB의 응답을 수신하면 AWS AppSync에서는 수신한 응답을 GraphQL 및 JSON 기본 형식으로 자동 변환합니다. DynamoDB의 각 속성은 디코딩되어 응답 핸들러의 컨텍스트에서 반환됩니다.

예를 들어 DynamoDB에서 다음을 반환한 경우:

```
{
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

파이프라인 해석기에서 결과가 반환되면 AWS AppSync는 이를 다음과 같이 GraphQL 및 JSON 유형으로 변환합니다.

```
{
    "id" : "1234",
    "name" : "Nadia",
```

```
    "age" : 25
}
```

이 단원에서는 AWS AppSync에서 다음 DynamoDB 스칼라, 문서 및 집합 형식을 변환하는 방법을 설명합니다.

String 형식 **S**

단일 문자열 값. DynamoDB String 값은 문자열로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB String 값을 반환한 경우:

```
{ "S" : "some string" }
```

AWS AppSync는 반환된 값을 문자열로 변환합니다.

```
"some string"
```

String set 형식 **SS**

문자열 값 집합. DynamoDB String Set 값은 문자열 목록으로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB String Set 값을 반환한 경우:

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync는 반환된 값을 문자열 목록으로 변환합니다.

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

Number 형식 **N**

단일 숫자 값. DynamoDB Number 값은 숫자로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB Number 값을 반환한 경우:

```
{ "N" : 1234 }
```

AWS AppSync는 반환된 값을 숫자로 변환합니다.

```
1234
```

Number set 형식 **NS**

숫자 값 집합. DynamoDB Number Set 값은 숫자 목록으로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB Number Set 값을 반환한 경우:

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync는 반환된 값을 숫자 목록으로 변환합니다.

```
[ 67.8, 12.2, 70 ]
```

Binary 형식 **B**

이진 값. DynamoDB Binary 값은 base64로 표시된 값이 포함된 문자열로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB Binary 값을 반환한 경우:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync는 반환된 값을 base64로 표시된 값이 포함된 문자열로 변환합니다.

```
"SGVsbG8sIFdvcmxkIQo="
```

이진 데이터는 RFC 4648 및 RFC 2045에 지정된 base64 인코딩 체계로 인코딩됩니다.

Binary set 형식 **BS**

이진 값 집합. DynamoDB Binary Set 값은 base64로 표시된 값이 포함된 문자열 목록으로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB Binary Set 값을 반환한 경우:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync는 반환된 값을 base64로 표시된 값이 포함된 문자열 목록으로 변환합니다.

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

이진 데이터는 RFC 4648 및 RFC 2045에 지정된 base64 인코딩 체계로 인코딩됩니다.

Boolean 형식 **BOOL**

부울 값. DynamoDB Boolean 값은 부울로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB Boolean 값을 반환한 경우:

```
{ "BOOL" : true }
```

AWS AppSync는 반환된 값을 부울로 변환합니다.

```
true
```

List 형식 **L**

기타 지원되는 DynamoDB 값 목록입니다. DynamoDB List 값은 값 목록으로 반환되며, 여기서 각 내부 값 역시 변환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB List 값을 반환한 경우:

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

AWS AppSync는 반환된 값을 변환된 값 목록으로 변환합니다.

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

Map 형식 **M**

지원되는 기타 모든 DynamoDB 값의 키/값 모음입니다. DynamoDB Map 값은 JSON 객체로 반환되며, 여기서 각 키/값 역시 변환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB Map 값을 반환한 경우:

```
{ "M" : {
```

```
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

AWS AppSync는 반환된 값을 JSON 객체로 변환합니다.

```
{
   "someString" : "A string value",
   "someNumber" : 1,
   "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

Null 형식 **NULL**

null 값.

예를 들어 DynamoDB에서 다음 DynamoDB Null 값을 반환한 경우:

```
{ "NULL" : null }
```

AWS AppSync는 반환된 값을 null로 변환합니다.

```
null
```

Filters

Query 및 Scan 작업을 사용하여 DynamoDB에서 객체를 쿼리하는 경우, 결과를 평가해 원하는 값만 반환하는 filter를 선택적으로 지정할 수 있습니다.

Query 또는 Scan 요청의 필터 속성은 다음과 같은 구조를 갖습니다.

```
type DynamoDBExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
};
```

필드는 다음과 같이 정의됩니다.

expression

쿼리 표현식. 필터 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB QueryFilter 및 DynamoDB ScanFilter 문서를 참조하십시오. 이 필드는 지정되어 있어야 합니다.

expressionNames

표현식 속성 name 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 `expression`에 사용된 이름 자리 표시자에 해당합니다. 이 값은 DynamoDB에서 항목의 속성 이름에 해당하는 문자열이어야 합니다. 이 필드는 선택 사항으로, `expression`에 사용된 표현식 속성인 name 자리 표시자의 대체 항목으로만 채워져야 합니다.

expressionValues

표현식 속성 value 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 `expression`에 사용되는 value 자리 표시자에 해당하고 값은 입력된 값이어야 합니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 입력된 값은 지정되어 있어야 합니다. 이 필드는 선택 사항으로, `expression`에 사용된 표현식 속성인 value 자리 표시자의 대체 항목으로만 채워져야 합니다.

예

다음 예제는 요청의 필터 섹션으로, 여기서 DynamoDB에서 가져온 항목은 제목이 `title` 인수로 시작되는 경우에만 반환됩니다.

여기서는 `util.transform.toDynamoDBFilterExpression`을 사용하여 객체에서 필터를 자동으로 생성합니다.

```
const filter = util.transform.toDynamoDBFilterExpression({
  title: { beginsWith: 'far away' },
});

const request = {};
request.filter = JSON.parse(filter);
```

그러면 다음과 같은 필터가 생성됩니다.

```
{
  "filter": {
```

```
    "expression": "(begins_with(#title,:title_beginsWith))",
    "expressionNames": { "#title": "title" },
    "expressionValues": {
      ":title_beginsWith": { "S": "far away" }
    }
  }
}
```

조건 표현식

DynamoDB에서 `PutItem`, `UpdateItem` 및 `DeleteItem` DynamoDB 작업을 사용하여 객체를 변경하는 경우, 작업을 수행하기 전에 DynamoDB에 이미 있는 객체의 상태를 기준으로 요청에 성공할지 여부를 제어하는 조건 표현식을 선택적으로 지정할 수 있습니다.

AWS AppSync DynamoDB 함수에서는 `PutItem`, `UpdateItem` 및 `DeleteItem` 요청 객체에 조건 표현식을 지정할 수 있고 조건에 실패해 객체가 업데이트되지 않은 경우 따를 전략을 지정할 수 있습니다.

예 1

다음 `PutItem` 요청 객체에는 조건식은 포함되어 있지 않습니다. 따라서 키가 같은 항목이 이미 존재하더라도 DynamoDB에 항목을 저장하기 때문에 기존 항목을 덮어 씁니다.

```
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),
  };
}
```

예 2

다음 `PutItem` 객체에는 동일한 키를 가진 항목이 DynamoDB에 없는 경우에만 작업을 성공시킬 수 있는 조건 표현식이 있습니다.

```
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)" }
  };
}
```

기본적으로 조건 확인이 실패하면 AWS AppSync DynamoDB 함수는 `ctx.error`에 오류를 제공합니다. GraphQL 응답의 `error` 섹션에 있는 `data` 필드에 DynamoDB에 있는 객체의 현재 값과 변형에 대한 오류를 반환할 수 있습니다.

그러나 AWS AppSync DynamoDB 함수에서는 개발자가 몇 가지 일반 엣지 케이스를 처리할 수 있도록 일부 추가 기능을 제공합니다.

- AWS AppSync DynamoDB 함수가 DynamoDB의 현재 값이 원하는 결과와 일치하는지 확인할 수 있는 경우 작업을 성공한 것으로 간주합니다.
- 따라서 오류를 반환하는 대신 AWS AppSync DynamoDB 함수가 실패를 처리해야 하는 방식을 결정하는 사용자 지정 Lambda 함수를 호출하도록 함수를 구성할 수 있습니다.

이러한 내용은 조건 확인 실패 처리 단원에 자세히 설명되어 있습니다.

DynamoDB 조건 표현식에 대한 자세한 내용은 DynamoDB ConditionExpressions 문서를 참조하세요.

조건 지정

`PutItem`, `UpdateItem` 및 `DeleteItem` 요청 객체 모두에서 선택적 `condition` 섹션을 지정할 수 있습니다. 이 섹션을 지정하지 않으면 조건 검사가 수행되지 않습니다. 지정한 경우, 해당 조건을 충족해야 작업이 성공합니다.

`condition` 섹션의 구조는 다음과 같습니다.

```
type ConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
  equalsIgnore?: string[];
  consistentRead?: boolean;
  conditionalCheckFailedHandler?: {
```

```
    strategy: 'Custom' | 'Reject';
    lambdaArn?: string;
  };
};
```

조건을 지정하는 필드는 다음과 같습니다.

expression

업데이트 표현식 자체. 조건 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB ConditionExpressions 문서를 참조하십시오. 이 필드는 지정되어 있어야 합니다.

expressionNames

표현식 속성 name 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 expression에 사용된 name 자리 표시자에 해당하고 값은 DynamoDB에 있는 항목의 속성 이름에 해당하는 문자열이어야 합니다. 이 필드는 선택 사항으로, expression에 사용된 표현식 속성인 name 자리 표시자의 대체 항목으로만 채워져야 합니다.

expressionValues

표현식 속성 value 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 expression에 사용되는 value 자리 표시자에 해당하고 값은 입력된 값이어야 합니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 입력된 값은 지정되어 있어야 합니다. 이 필드는 선택 사항으로, expression에 사용된 표현식 속성인 value 자리 표시자의 대체 항목으로만 채워져야 합니다.

나머지 필드는 AWS AppSync DynamoDB 함수에 조건 검사 실패를 처리하는 방법을 지시합니다.

equalsIgnore

`PutItem` 작업 사용 시 조건 검사에 실패하면 AWS AppSync DynamoDB 함수는 DynamoDB에 현재 있는 항목을 쓰려고 하는 항목과 비교합니다. 이러한 두 항목이 동일하면 해석기에서는 해당 작업을 마치 성공한 것처럼 간주합니다. `equalsIgnore` 필드를 사용하여 비교 수행 시 AWS AppSync에서 무시해야 하는 속성 목록을 지정할 수 있습니다. 예를 들어, `version` 속성만 차이가 나는 경우에는 해당 작업을 성공한 작업으로 취급합니다. 이 필드는 선택 사항입니다.

consistentRead

조건 검사에 실패하면 AWS AppSync에서는 강력히 일관된 읽기를 사용하여 DynamoDB에서 항목의 현재 값을 가져옵니다. 이 필드를 사용하여 AWS AppSync DynamoDB 함수에 최종 읽기 일관성을 대신 사용하도록 지시할 수 있습니다. 이 필드는 선택 사항으로, 기본값은 `true`입니다.

conditionalCheckFailedHandler

이 섹션에서는 AWS AppSync DynamoDB 함수가 DynamoDB의 현재 값을 예상 결과와 비교한 후 조건 검사를 처리하는 방법을 지정할 수 있습니다. 이 섹션은 선택 사항으로, 방법을 지정하지 않으면 `Reject` 전략이 기본값으로 지정됩니다.

strategy

AWS AppSync DynamoDB 함수가 DynamoDB의 현재 값을 예상 결과와 비교한 후 따르는 전략. 이 필드는 필수 필드로, 다음과 같은 두 가지 값을 가질 수 있습니다.

Reject

변형에 실패하고, GraphQL 응답의 `error` 섹션에 있는 `data` 필드에 DynamoDB에 있는 객체의 현재 값과 변형에 대한 오류를 반환합니다.

Custom

AWS AppSync DynamoDB 함수는 조건 검사 실패를 처리하는 방법을 결정하는 사용자 지정 Lambda 함수를 호출합니다. `strategy`를 `Custom`으로 설정하면 `lambdaArn` 필드에 호출할 Lambda 함수의 ARN이 포함되어 있어야 합니다.

lambdaArn

AWS AppSync DynamoDB 함수가 조건 검사 실패를 처리하는 방법을 결정하기 위해 호출하는 Lambda 함수의 ARN. 이 필드는 `strategy`를 `Custom`으로 설정한 경우에만 지정해야 합니다. 이 기능을 사용하는 방법에 대한 자세한 내용은 조건 검사 실패 처리를 참조하세요.

조건 검사 실패 처리

조건 확인이 실패하면 AWS AppSync DynamoDB 함수는 `util.appendError` 유틸리티를 사용하여 변형에 대한 오류와 객체의 현재 값을 전달할 수 있습니다. 그러면 GraphQL 응답의 `error` 섹션에 `data` 필드가 추가됩니다. 그러나 AWS AppSync DynamoDB 함수에서는 개발자가 몇 가지 일반 엣지 케이스를 처리할 수 있도록 일부 추가 기능을 제공합니다.

- AWS AppSync DynamoDB 함수가 DynamoDB의 현재 값이 원하는 결과와 일치하는지 확인할 수 있는 경우 작업을 성공한 것으로 간주합니다.
- 따라서 오류를 반환하는 대신 AWS AppSync DynamoDB 함수가 실패를 처리해야 하는 방식을 결정하는 사용자 지정 Lambda 함수를 호출하도록 함수를 구성할 수 있습니다.

이 프로세스의 흐름 차트입니다.

원하는 결과 확인

조건 검사에 실패하면 AWS AppSync DynamoDB 함수는 `GetItem` DynamoDB 요청을 수행하여 DynamoDB에서 항목의 현재 값을 가져옵니다. 기본적으로 이 해석기는 강력히 일관된 읽기를 사용하지만, `condition` 블록의 `consistentRead` 필드를 사용하여 예상 결과와 비교하도록 구성할 수 있습니다.

- `PutItem` 작업의 경우 AWS AppSync DynamoDB 함수는 현재 값을 쓰려는 값과 비교하여 비교 결과에서 `equalsIgnore`에 나열된 모든 속성을 제외합니다. 항목이 동일한 경우 해당 작업을 성공한 작업으로 간주하고 DynamoDB에서 가져온 항목을 반환합니다. 그렇지 않으면 구성된 전략을 따릅니다.

  예를 들어, `PutItem` 요청 함수가 다음과 같은 경우:

```
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']
    }
  };
}
```

  그리고 DynamoDB에 현재 있는 항목이 다음과 같은 경우:

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

  AWS AppSync DynamoDB 함수는 쓰려는 항목을 현재 값과 비교해 `version` 필드만 차이가 난다는 사실을 파악합니다. 그러나 `version` 필드를 무시하도록 구성되어 있기 때문에 해당 작업을 성공으로 간주하고 DynamoDB에서 가져온 항목을 반환합니다.

- `DeleteItem` 작업의 경우 AWS AppSync DynamoDB 함수는 항목이 DynamoDB에서 반환되었는지 확인합니다. 반환되는 항목이 없는 경우 해당 작업을 성공으로 간주합니다. 그렇지 않으면 구성된 전략을 따릅니다.
- `UpdateItem` 작업의 경우 AWS AppSync DynamoDB 함수는 현재 DynamoDB에 있는 항목이 예상 결과와 일치하는지 확인하는데 필요한 충분한 정보가 없기 때문에 구성된 전략을 따릅니다.

DynamoDB에 있는 객체의 현재 상태가 예상 결과와 다르면 AWS AppSync DynamoDB 함수는 구성된 전략에 따라 변경을 거부하거나 다음에 수행할 작업을 결정하기 위해 Lambda 함수를 호출합니다.

'Reject' 전략 따르기

`Reject` 전략을 따르는 경우 AWS AppSync DynamoDB 함수는 변형에 대한 오류를 반환하고 GraphQL 응답의 `error` 섹션에 있는 `data` 필드에 DynamoDB에 있는 객체의 현재 값을 반환합니다. DynamoDB에서 반환된 항목은 클라이언트에서 기대하는 형식으로 항목을 변환하는 함수 응답 핸들러를 통해 입력되며 선택 세트를 기준으로 필터링됩니다.

예를 들어 다음과 같은 변형 요청이 있습니다.

```
mutation {
    updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
        Name
        theVersion
    }
}
```

DynamoDB에서 반환된 항목이 다음과 같은 경우:

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

그리고 함수 응답 핸들러는 다음과 같습니다.

```
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
}
```

GraphQL 응답은 다음과 같습니다.

```
{
  "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
      },
      ...
    }
  ]
}
```

반환된 객체의 필드를 다른 해석기가 채우고 변형에 성공했는데 이 객체가 `error` 섹션에서 반환되면 반환된 객체의 필드가 확인되지 않습니다.

'Custom' 전략 따르기

`Custom` 전략을 따르면 AWS AppSync DynamoDB 함수는 다음에 수행해야 할 작업을 결정하는 Lambda 함수를 호출합니다. Lambda 함수는 다음 옵션 중 하나를 선택합니다.

- 변형 `reject`. 이 옵션은 AWS AppSync DynamoDB 함수에 구성된 전략이 `Reject`인 것처럼 동작하도록 지시해 이전의 단원에서 설명한 것처럼 DynamoDB에 있는 객체의 현재 값과 변형에 대한 오류를 반환하도록 합니다.
- 변형 `discard`. 이 옵션은 AWS AppSync DynamoDB 함수에 조건 검사 실패를 자동으로 무시하고 DynamoDB의 값을 반환하도록 지시합니다.
- 변형 `retry`. 이 옵션은 AWS AppSync DynamoDB 함수에 새 요청 객체를 사용하여 변형을 다시 시도하도록 지시합니다.

Lambda 호출 요청

AWS AppSync DynamoDB 함수가 `lambdaArn`에 지정된 Lambda 함수를 호출합니다. 데이터 원본에 대해 구성된 동일한 `service-role-arn`을 사용합니다. 호출의 페이로드 구조는 다음과 같습니다.

```
{
    "arguments": { ... },
    "requestMapping": {... },
    "currentValue": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

필드는 다음과 같이 정의됩니다.

`arguments`

GraphQL 변형의 인수. `context.arguments`에서 요청 객체서에 사용할 수 있는 인수와 동일합니다.

`requestMapping`

이 작업에 대한 요청 객체입니다.

`currentValue`

DynamoDB에 있는 객체의 현재 값

`resolver`

AWS AppSync 해석기 또는 함수에 대한 정보입니다.

`identity`

호출자에 대한 정보. `context.identity`에서 요청 객체에 사용할 수 있는 자격 증명 정보와 동일합니다.

페이로드 전체를 보여주는 예:

```
{
    "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"
    }
}
```

Lambda 호출 응답

Lambda 함수는 호출 페이로드를 검사해 AWS AppSync DynamoDB 함수에서 실패를 처리해야 하는 방법을 결정하는 비즈니스 로직을 어떠한 것이든 적용할 수 있습니다. 조건 검사 실패를 처리하기 위한 옵션에는 다음 3가지가 있습니다.

- 변형 `reject`. 이 옵션에 대한 응답 페이로드의 구조는 다음과 같아야 합니다.

```
{
    "action": "reject"
}
```

  이 옵션은 AWS AppSync DynamoDB 함수에 구성된 전략이 `Reject`인 것처럼 동작하도록 지시해 위의 섹션에서 설명한 것처럼 DynamoDB에 있는 객체의 현재 값과 변형에 대한 오류를 반환하도록 합니다.

- 변형 `discard`. 이 옵션에 대한 응답 페이로드의 구조는 다음과 같아야 합니다.

```
{
    "action": "discard"
}
```

  이 옵션은 AWS AppSync DynamoDB 함수에 조건 검사 실패를 자동으로 무시하고 DynamoDB의 값을 반환하도록 지시합니다.

- 변형 `retry`. 이 옵션에 대한 응답 페이로드의 구조는 다음과 같아야 합니다.

```
{
    "action": "retry",
    "retryMapping": { ... }
}
```

  이 옵션은 AWS AppSync DynamoDB 함수에 새 요청 객체를 사용하여 변형을 다시 시도하도록 지시합니다. `retryMapping` 섹션의 구조는 DynamoDB 작업에 따라 달라지며 해당 작업에 대한 전체 요청 객체의 하위 집합입니다.

  `PutItem`의 경우 `retryMapping` 섹션의 구조는 다음과 같습니다. `attributeValues` 필드에 대한 설명은 PutItem을 참조하십시오.

```
{
    "attributeValues": { ... },
    "condition": {
        "equalsIgnore" = [ ... ],
        "consistentRead" = true
    }
}
```

UpdateItem의 경우 retryMapping 섹션의 구조는 다음과 같습니다. update 섹션에 대한 설명은 UpdateItem을 참조하십시오.

```
{
    "update" : {
        "expression" : "someExpression"
        "expressionNames" : {
            "#foo" : "foo"
        },
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "condition": {
        "consistentRead" = true
    }
}
```

DeleteItem의 경우 retryMapping 섹션의 구조는 다음과 같습니다.

```
{
    "condition": {
        "consistentRead" = true
    }
}
```

작동할 수 있는 다른 작업 또는 키를 지정할 수 있는 방법은 없습니다. AWS AppSync DynamoDB 함수는 동일한 객체에 대해 동일한 작업만 재시도하도록 허용합니다. condition 섹션에서는 conditionalCheckFailedHandler를 지정할 수 없습니다. 재시도에 실패하면 AWS AppSync DynamoDB 함수는 Reject 전략을 따릅니다.

다음은 실패한 PutItem 요청을 처리하는 Lambda 함수의 예입니다. 이 비즈니스 로직은 호출한 사람이 누구인지 확인합니다. jeffTheAdmin이 호출한 경우 요청을 재시도해 현재 DynamoDB에 있는 항목에서 version 및 expectedVersion을 업데이트합니다. 그렇지 않으면 변형을 거부합니다.

```
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)
};
```

트랜잭션 조건 표현식

트랜잭션 조건 표현식은 `TransactWriteItems`에서 네 가지 작업 유형 모두(`PutItem`, `DeleteItem`, `UpdateItem`, `ConditionCheck`)의 요청에 사용할 수 있습니다.

`PutItem`, `DeleteItem`, `UpdateItem`의 경우 트랜잭션 조건 표현식은 선택 사항입니다. `ConditionCheck`의 경우 트랜잭션 조건 표현식이 필수입니다.

예 1

다음 트랜잭션 `DeleteItem` 함수 요청 핸들러에는 조건 표현식이 없습니다. 따라서 DynamoDB에서 항목을 삭제합니다.

```
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 }),
      }
    ],
  };
}
```

예 2

다음 트랜잭션 DeleteItem 함수 요청 핸들러에는 해당 게시물의 작성자가 특정 이름과 같은 경우에만 작업이 성공할 수 있는 트랜잭션 조건식이 있습니다.

```
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 },
        }),
      }
    ],
  };
}
```

조건 검사에 실패하면 `TransactionCanceledException`이 발생하고, `ctx.result.cancellationReasons`에 오류 세부 정보가 반환됩니다. 기본적으로, 조건 검사가 실패하게 된 원인인 DynamoDB의 이전 항목이 `ctx.result.cancellationReasons`에 반환됩니다.

조건 지정

`PutItem`, `UpdateItem` 및 `DeleteItem` 요청 객체 모두에서 선택적 `condition` 섹션을 지정할 수 있습니다. 이 섹션을 지정하지 않으면 조건 검사가 수행되지 않습니다. 지정한 경우, 해당 조건을 충족해야 작업이 성공합니다. `ConditionCheck`에는 지정할 `condition` 섹션이 있어야 합니다. 전체 트랜잭션이 성공하려면 조건이 true여야 합니다.

`condition` 섹션의 구조는 다음과 같습니다.

```
type TransactConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string };
  expressionValues?: { [key: string]: string };
  returnValuesOnConditionCheckFailure: boolean;
};
```

조건을 지정하는 필드는 다음과 같습니다.

`expression`

업데이트 표현식 자체. 조건 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB ConditionExpressions 문서를 참조하십시오. 이 필드는 지정되어 있어야 합니다.

`expressionNames`

표현식 속성 name 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 expression에 사용된 name 자리 표시자에 해당하고 값은 DynamoDB에 있는 항목의 속성 이름에 해당하는 문자열이어야 합니다. 이 필드는 선택 사항으로, expression에 사용된 표현식 속성인 name 자리 표시자의 대체 항목으로만 채워져야 합니다.

`expressionValues`

표현식 속성 value 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 expression에 사용되는 value 자리 표시자에 해당하고 값은 입력된 값이어야 합니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 입력된 값은 지정되어 있어야 합니다. 이 필드는 선택 사항으로, expression에 사용된 표현식 속성인 value 자리 표시자의 대체 항목으로만 채워져야 합니다.

`returnValuesOnConditionCheckFailure`

조건 검사에 실패할 경우 DynamoDB의 항목을 다시 가져올지 여부를 지정합니다. 가져온 항목은 `ctx.result.cancellationReasons[<index>].item`에 있습니다. 여기서 `<index>`는 조건 검사에 실패한 요청 항목의 인덱스입니다. 기본값은 true입니다.

프로젝션

`GetItem`, `Scan`, `Query`, `BatchGetItem` 및 `TransactGetItems` 작업을 사용하여 DynamoDB에서 객체를 읽을 때 원하는 속성을 식별하는 프로젝션을 선택적으로 지정할 수 있습니다. 프로젝션 속성의 구조는 필터와 비슷하며 다음과 같습니다.

```
type DynamoDBExpression = {
  expression: string;
  expressionNames?: { [key: string]: string}
};
```

필드는 다음과 같이 정의됩니다.

`expression`

프로젝션 표현식은 문자열입니다. 단일 속성을 가져오려면 속성의 이름을 지정합니다. 여러 속성의 경우 이름은 쉼표로 구분된 값이어야 합니다. 프로젝션 표현식 작성에 대한 자세한 내용은 DynamoDB 프로젝션 표현식 설명서를 참조하세요. 이 필드는 필수 항목입니다.

`expressionNames`

표현식 속성 name 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 `expression`에 사용된 이름 자리 표시자에 해당합니다. 이 값은 DynamoDB에서 항목의 속성 이름에 해당하는 문자열이어야 합니다. 이 필드는 선택 사항으로, `expression`에 사용된 표현식 속성인 name 자리 표시자의 대체 항목으로만 채워져야 합니다. `expressionNames`에 대한 자세한 내용은 DynamoDB 설명서를 참조하세요.

예 1

다음 예는 DynamoDB에서 `author` 및 `id` 속성만 반환되는 JavaScript 함수의 프로젝션 섹션입니다.

```
projection : {
    expression : "#author, id",
    expressionNames : {
        "#author" : "author"
    }
}
```

> **Tip**
>
> selectionSetList를 사용하여 GraphQL 요청 설정에 액세스할 수 있습니다. 이 필드를 사용하면 요구 사항에 따라 프로젝션 표현식을 동적으로 프레이밍할 수 있습니다.

> **Note**
>
> `Query` 및 `Scan` 작업과 함께 프로젝션 표현식을 사용하는 경우 `select`의 값은 반드시 `SPECIFIC_ATTRIBUTES`이어야 합니다. 자세한 내용은 DynamoDB 설명서를 참조하세요.

OpenSearch용 JavaScript 해석기 함수 참조

Amazon OpenSearch Service용 AWS AppSync 해석기에서는 GraphQL을 사용하여 계정에 있는 기존 OpenSearch Service 도메인의 데이터를 저장 및 검색할 수 있습니다. 이 해석기는 수신되는 GraphQL 요청을 OpenSearch Service 요청으로 매핑한 다음 OpenSearch Service 응답을 다시 GraphQL로 매핑하도록 하여 작동합니다. 이 섹션에서는 지원되는 OpenSearch Service 작업에 대한 요청 및 응답 핸들러를 설명합니다.

요청

대부분의 OpenSearch Service 요청 객체에는 몇 부분만 변경되는 공통 구조가 있습니다. 다음 예제는 문서가 `post` 유형이고 `id`에서 인덱싱되는 OpenSearch Service 도메인에 대해 검색을 실행합니다. 검색 파라미터는 `body` 섹션에 정의되어 있고, 공통 쿼리 절은 대부분 `query` 필드에서 정의합니다. 이 예에서는 문서의 `"Nadia"` 필드에서 `"Bailey"`, `author` 또는 둘 다가 포함된 문서를 검색합니다.

```
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' } },
            ],
          },
        },
      },
    },
  };
}
```

응답

다른 데이터 소스와 마찬가지로, OpenSearch Service는 GraphQL로 변환해야 하는 응답을 AWS AppSync에 전송합니다.

대부분의 GraphQL 쿼리는 OpenSearch Service 응답에서 `_source` 필드를 검색합니다. 개별 문서 또는 문서 목록을 반환하도록 검색을 수행할 수 있기 때문에 OpenSearch Service에서는 두 가지 공통 응답 패턴이 사용됩니다.

결과 목록

```
export function response(ctx) {
  const entries = [];
  for (const entry of ctx.result.hits.hits) {
    entries.push(entry['_source']);
  }
  return entries;
}
```

개별 항목

```
export function response(ctx) {
  return ctx.result['_source']
}
```

`operation` field

(요청 핸들러만 해당)

AWS AppSync에서 OpenSearch Service 도메인에 보내는 HTTP 메서드 또는 동사(GET, POST, PUT, HEAD 또는 DELETE). 키와 값은 둘 다 문자열이어야 합니다.

```
"operation" : "PUT"
```

`path` field

(요청 핸들러만 해당)

AWS AppSync에서 보낸 OpenSearch Service 요청에 대한 검색 경로입니다. 이 경로는 작업의 HTTP 동사에 대한 URL을 구성합니다. 키와 값은 둘 다 문자열이어야 합니다.

```
"path" : "/indexname/type"

"path" : "/indexname/type/_search"
```

요청 핸들러가 평가되면 이 경로는 OpenSearch Service 도메인을 포함해 HTTP 요청의 일부로 전송됩니다. 예를 들어, 이전 예제는 다음과 같이 변환될 수 있습니다.

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

`params` field

(요청 핸들러만 해당)

검색에서 수행할 작업을 지정하는 데 사용되며, 가장 일반적으로 본문 내에 쿼리 값을 설정하여 지정합니다. 그러나 응답 형식 지정 등과 같이 구성할 수 있는 기타 여러 가지 기능이 있습니다.

- headers

  키-값 페어로 표시되는 헤더 정보. 키와 값은 둘 다 문자열이어야 합니다. 예:

  ```
  "headers" : {
      "Content-Type" : "application/json"
  }
  ```

  > **Note**
  >
  > AWS AppSync는 현재 JSON만 `Content-Type`으로 지원합니다.

- queryString

JSON 응답에 대한 코드 형식 지정 등과 같이 공통 옵션을 지정하는 키-값 페어. 키와 값은 둘 다 문자열이어야 합니다. 예를 들어, 형식이 잘 지정된 JSON을 가져오려면 다음을 사용합니다.

```
"queryString" : {
    "pretty" : "true"
}
```

- body

  요청의 주요 부분으로, AWS AppSync에서 OpenSearch Service 도메인에 대해 잘 구성된 검색 요청을 작성하도록 합니다. 키는 객체로 구성된 문자열이어야 합니다. 아래에 몇 가지 경우가 나와 있습니다.

예 1

"시애틀"과 일치하는 도시가 포함된 모든 문서를 반환합니다.

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: { from: 0, size: 50, query: { match: { city: 'seattle' } } },
    },
  };
}
```

예제 2

도시 또는 주가 "워싱턴"과 일치하는 모든 문서를 반환합니다.

```
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'] },
        },
      },
    },
  };
}
```

변수 전달

(요청 핸들러만 해당)

요청 핸들러에서 평가의 일부로 변수를 전달할 수도 있습니다. 예를 들어, 다음과 같은 GraphQL 쿼리가 있다고 가정해 보겠습니다.

```
query {
    searchForState(state: "washington"){
        ...
    }
}
```

함수 요청 핸들러는 다음과 같을 수 있습니다.

```
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 Lambda용 리졸버 함수 참조

AWS AppSync 함수와 리졸버를 사용하여 계정에 있는 Lambda 함수를 호출할 수 있습니다. Lambda 함수를 클라이언트에 반환하기 전에 요청 페이로드와 Lambda 함수의 응답을 조정할 수 있습니다. 요청 객체에서 수행할 작업 유형을 지정할 수도 있습니다. 이 단원에서는 지원되는 Lambda 작업에 대한 요청에 대해 설명합니다.

요청 객체

Lambda 요청 객체는 Lambda 함수와 관련된 필드를 처리합니다.

```
export type LambdaRequest = {
  operation: 'Invoke' | 'BatchInvoke';
  invocationType?: 'RequestResponse' | 'Event';
  payload: unknown;
};
```

다음은 GraphQL 스키마의 `getPost` 필드로 페이로드 데이터를 컨텍스트의 인수와 함께 사용하는 `invoke` 작업을 사용하는 예제입니다.

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

전체 매핑 문서가 Lambda 함수에 입력으로 전달되므로 이전 예제는 이제 다음과 같습니다.

```
{
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "input": {
        "id": "postId1",
      }
    }
```

```
  }
}
```

Operation

Lambda 데이터 소스를 사용하면 필드에 두 가지 작업 (및) 을 정의할 수 있습니다`operation`. `Invoke` `BatchInvoke` 이 `Invoke` 작업은 모든 GraphQL 필드 리졸버에 대해 Lambda 함수를 호출하도록 AWS AppSync 알려줍니다. `BatchInvoke`현재 GraphQL 필드에 대한 요청을 일괄 AWS AppSync 처리하도록 지시합니다. `operation` 필드는 필수 사항입니다.

의 경우`Invoke`, 해결된 요청은 Lambda 함수의 입력 페이로드와 일치합니다. 위 예제를 수정해 보겠습니다.

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

이 문제는 해결되어 Lambda 함수로 전달되며, 다음과 같이 보일 수 있습니다.

```
{
  "operation": "Invoke",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

의 경우`BatchInvoke`, 요청은 일괄 처리의 모든 필드 리졸버에 적용됩니다. 간결하게 정리하기 위해 모든 요청 `payload` 값을 요청 객체와 일치하는 단일 객체 아래의 목록으로 AWS AppSync 병합합니다. 다음 예제 요청 핸들러는 병합을 보여줍니다.

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: ctx,
  };
```

```
}
```

이 요청은 평가되어 다음 매핑 문서로 해결됩니다.

```
{
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

payload목록의 각 요소는 단일 배치 항목에 해당합니다. 또한 Lambda 함수는 요청에 전송된 항목의 순서와 일치하는 목록 형태의 응답을 반환할 것으로 예상됩니다.

```
[
  { "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
]
```

페이로드

payload필드는 Lambda 함수에 데이터를 전달하는 데 사용되는 컨테이너입니다. operation필드가로 BatchInvoke 설정된 경우 기존 payload 값을 목록으로 AWS AppSync 래핑합니다. payload 필드는 선택 사항입니다.

호출 유형

Lambda 데이터 소스를 사용하면 두 가지 호출 유형인 및 을 정의할 수 있습니다. RequestResponse Event 호출 유형은 Lambda API에 정의된 호출 유형과 동일합니다. RequestResponse호출 유형을 사용하면 AWS AppSync Lambda 함수를 동기적으로 호출하여 응답을 기다릴 수 있습니다. Event호출을 통해 Lambda 함수를 비동기적으로 호출할 수 있습니다. Lambda가 호출 유형 요청을 Event 처리하는 방법에 대한 자세한 내용은 비동기 호출을 참조하십시오. invocationType 필드는 선택 사항입니다. 이 필드가 요청에 포함되지 않은 경우 기본적으로 호출 유형이 사용됩니다. AWS AppSync RequestResponse

모든 invocationType 필드에 대해 해결된 요청은 Lambda 함수의 입력 페이로드와 일치합니다. 위 예제를 수정해 보겠습니다.

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    invocationType: 'Event',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

이 문제는 해결되어 Lambda 함수로 전달되며, 다음과 같이 보일 수 있습니다.

```
{
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

BatchInvoke작업이 Event 호출 유형 필드와 함께 사용될 경우 위에서 언급한 것과 동일한 방식으로 필드 확인자를 AWS AppSync 병합하고, 요청은 값 목록이 되는 비동기 이벤트로 Lambda 함수에 전달됩니다. payload Event호출 유형 요청의 응답으로 인해 응답 핸들러가 없는 값이 생성됩니다. null

```
{
  "data": {
    "field": null
  }
}
```

캐시 적중이 발생할 경우 Lambda로 전송되지 않으므로 Event 호출 유형 확인자에 대해서는 확인자 캐싱을 비활성화하는 것이 좋습니다.

응답 객체

다른 데이터 소스와 마찬가지로 Lambda 함수는 GraphQL 유형으로 변환되어야 AWS AppSync 하는 응답을 보냅니다. Lambda 함수의 결과는 결과 속성 () context 에 포함되어 있습니다. context.result

Lambda 함수 응답의 모양이 GraphQL 유형의 모양과 일치하는 경우 다음 함수 응답 핸들러를 사용하여 응답을 전달할 수 있습니다.

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

응답 객체에 적용되는 필수 필드 또는 모양 제한이 없습니다. 하지만 GraphQL은 강력한 형식이므로 해석된 응답이 필요한 GraphQL 유형과 일치해야 합니다.

Lambda 함수 일괄 처리 응답

`operation`필드가 로 `BatchInvoke` 설정된 경우 Lambda 함수에서 항목 목록이 반환될 AWS AppSync 것으로 예상됩니다. 각 결과를 원래 요청 항목에 다시 AWS AppSync 매핑하려면 응답 목록의 크기와 순서가 일치해야 합니다. 응답 목록에 `null` 항목이 있어도 유효합니다. 이에 따라 null로 *`ctx.result`* 설정됩니다.

JavaScript 데이터 소스에 대한 EventBridge 리졸버 함수 참조

EventBridge 데이터 소스와 함께 사용되는 AWS AppSync 리졸버 함수 요청 및 응답을 통해 Amazon EventBridge 버스에 사용자 지정 이벤트를 보낼 수 있습니다.

요청

요청 핸들러를 사용하면 여러 개의 사용자 지정 이벤트를 이벤트 버스에 보낼 수 있습니다 EventBridge .

```
export function request(ctx) {
  return {
    "operation" : "PutEvents",
    "events" : [{}]
  }
}
```

EventBridge `PutEvents`요청의 유형 정의는 다음과 같습니다.

```
type PutEventsRequest = {
  operation: 'PutEvents'
```

```
  events: {
    source: string
    detail: { [key: string]: any }
    detailType: string
    resources?: string[]
    time?: string // RFC3339 Timestamp format
  }[]
}
```

응답

PutEvents작업이 성공하면 응답 EventBridge 양식이 다음에 포함됩니다ctx.result.

```
export function response(ctx) {
  if(ctx.error)
    util.error(ctx.error.message, ctx.error.type, ctx.result)
  else
    return ctx.result
}
```

InternalExceptions 또는 Timeouts와 같은 PutEvents 작업을 수행하는 동안 발생하는 오류는 ctx.error에 표시됩니다. EventBridge의 일반적인 오류 목록은 일반 오류 참조를 EventBridge 참조하십시오.

result에는 다음과 같은 유형 정의가 있습니다.

```
type PutEventsResult = {
  Entries: {
    ErrorCode: string
    ErrorMessage: string
    EventId: string
  }[]
  FailedEntryCount: number
}
```

- 항목

  수집된 이벤트 결과(성공 및 실패 모두) 수집이 성공하면 항목에 EventID가 포함됩니다. 그렇지 않으면 ErrorCode 및 ErrorMessage를 사용하여 항목의 문제를 식별할 수 있습니다.

  각 레코드의 응답 요소의 인덱스는 요청 배열의 인덱스와 동일합니다.

- FailedEntryCount

  실패한 항목 수입니다. 이 값은 정수로 표시됩니다.

의 응답에 대한 자세한 내용은 PutEvents 을 참조하십시오 PutEvents.

예제 샘플 응답 1

다음 예는 두 개의 성공적인 이벤트가 있는 PutEvents 작업입니다.

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ],
    "FailedEntryCount" : 0
}
```

예제 샘플 응답 2

다음 예는 3개의 이벤트(성공 2개, 실패 1개)가 있는 PutEvents 작업입니다.

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        },
        {
            "ErrorCode" : "SampleErrorCode",
            "ErrorMessage" : "Sample Error Message"
        }
    ],
    "FailedEntryCount" : 1
}
```

`PutEvents` field

- 버전

  모든 요청 매핑 템플릿에 공통적으로 적용되는 `version` 필드는 템플릿이 사용하는 버전을 정의합니다. 이 필드는 필수 항목입니다. `2018-05-29`값은 EventBridge 매핑 템플릿에 지원되는 유일한 버전입니다.

- Operation

  지원되는 유일한 작업은 `PutEvents`입니다. 이 작업을 통해 이벤트 버스에 사용자 지정 이벤트를 추가할 수 있습니다.

- 이벤트

  이벤트 버스에 추가될 이벤트 배열입니다. 이 배열에는 1~10개의 항목이 할당되어야 합니다.

  `Event` 객체에는 다음의 필드가 있습니다.

  - `"source"`: 이벤트 소스를 정의하는 문자열입니다.
  - `"detail"`: 이벤트에 대한 정보를 첨부하는 데 사용할 수 있는 JSON 객체입니다. 이 필드는 빈 맵(`{ }`)일 수 있습니다.
  - `"detailType`: 스토리지의 유형을 식별하는 문자열입니다.
  - `"resources"`: 이벤트와 관련된 리소스를 식별하는 문자열의 JSON 배열입니다. 이 필드는 빈 배열일 수 있습니다.
  - `"time"`: 이벤트 타임스탬프는 문자열로 제공됩니다. 이는 RFC3339 타임스탬프 형식을 따라야 합니다.

다음은 유효한 `Event` 객체의 몇 가지 예시입니다.

예 1

```
{
    "source" : "source1",
    "detail" : {
        "key1" : [1,2,3,4],
        "key2" : "strval"
    },
    "detailType" : "sampleDetailType",
    "resources" : ["Resouce1", "Resource2"],
    "time" : "2022-01-10T05:00:10Z"
```

```
}
```

예제 2

```
{
    "source" : "source1",
    "detail" : {},
    "detailType" : "sampleDetailType"
}
```

예 3

```
{
    "source" : "source1",
    "detail" : {
        "key1" : 1200
    },
    "detailType" : "sampleDetailType",
    "resources" : []
}
```

None 데이터 소스용 JavaScript 해석기 함수 참조

데이터 소스 유형이 없음인 AWS AppSync 해석기 함수 요청 및 응답을 사용하면 AWS AppSync 로컬 작업에 대한 요청을 조정할 수 있습니다.

요청

요청 핸들러는 단순하며 payload 필드를 통해 가능한 많은 컨텍스트 정보를 전달할 수 있습니다.

```
type NONERequest = {
  payload: any;
};
```

다음은 필드 인수가 페이로드에 전달되는 예시입니다.

```
export function request(ctx) {
  return {
    payload: context.args
  };
```

```
}
```

payload 필드 값은 함수 응답 핸들러로 전달되며 context.result에서 사용할 수 있습니다.

Payload

payload 필드는 모든 데이터를 전달한 다음 함수 응답 핸들러에서 사용할 수 있는 컨테이너입니다.

payload 필드는 선택 사항입니다.

응답

데이터 소스가 없으므로 payload 필드의 값이 함수 응답 핸들러로 전달되고 context.result 속성에서 설정됩니다.

payload 필드 값의 모양이 GraphQL 유형의 모양과 정확히 일치하는 경우 다음 응답 핸들러를 사용하여 응답을 전달할 수 있습니다.

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

반환 응답에 적용되는 필수 필드 또는 모양 제한이 없습니다. 하지만 GraphQL은 강력한 형식이므로 해석된 응답이 필요한 GraphQL 유형과 일치해야 합니다.

JavaScript HTTP에 대한 리졸버 함수 참조

AWS AppSync HTTP 해석기 함수를 사용하면 모든 HTTP 엔드포인트에서 요청을 보내고 HTTP 엔드포인트에서 받은 AWS AppSync 응답을 다시 HTTP 엔드포인트로 보낼 수 있습니다. AWS AppSync 요청 핸들러를 사용하여 호출할 작업의 특성에 AWS AppSync 대한 힌트를 제공할 수 있습니다. 이 단원에서는 지원되는 HTTP 해석기에 대한 다양한 구성에 대해 설명합니다.

요청

```
type HTTPRequest = {
  method: 'PUT' | 'POST' | 'GET' | 'DELETE' | 'PATCH';
  params?: {
    query?: { [key: string]: any };
    headers?: { [key: string]: string };
    body?: any;
```

```
  };
  resourcePath: string;
};
```

다음 코드 조각은 text/plain 본문을 포함한 HTTP POST 요청의 예입니다.

```
export function request(ctx) {
  return {
    method: 'POST',
    params: {
      headers: { 'Content-Type': 'text/plain' },
      body: 'this is an example of text body',
    },
    resourcePath: '/',
  };
}
```

메서드

요청 핸들러만 해당

AWS AppSync 에서 HTTP 엔드포인트에 보내는 HTTP 메서드 또는 동사(GET, POST, PUT, PATCH 또는 DELETE).

```
"method": "PUT"
```

ResourcePath

요청 핸들러만 해당

액세스하고자 하는 리소스 경로입니다. 리소스 경로는 HTTP 데이터 원본의 엔드포인트와 함께 AWS AppSync 서비스에서 요청을 수행하는 URL을 형성합니다.

```
"resourcePath": "/v1/users"
```

요청이 평가되면 이 경로는 HTTP 엔드포인트를 포함해 HTTP 요청의 일부로 전송됩니다. 예를 들어, 이전 예제는 다음과 같이 변환될 수 있습니다.

```
PUT <endpoint>/v1/users
```

Params Field

요청 핸들러만 해당

검색에서 수행할 작업을 지정하는 데 사용되며, 가장 일반적으로 본문 내에 쿼리 값을 설정하여 지정합니다. 그러나 응답 형식 지정 등과 같이 구성할 수 있는 기타 여러 가지 기능이 있습니다.

headers

키-값 페어로 표시되는 헤더 정보. 키와 값은 둘 다 문자열이어야 합니다.

예:

```
"headers" : {
    "Content-Type" : "application/json"
}
```

현재 지원되는 Content-Type 헤더는 다음과 같습니다.

```
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
```

다음 HTTP 헤더를 설정할 수 없습니다.

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

JSON 응답에 대한 코드 형식 지정 등과 같이 공통 옵션을 지정하는 키-값 페어. 키와 값은 둘 다 문자열이어야 합니다. 다음 예제는 쿼리 문자열을 ?type=json으로 보낼 방법을 보여줍니다.

```
"query" : {
    "type" : "json"
}
```

body

본문에는 설정을 선택하는 HTTP 요청 본문이 포함됩니다. 콘텐츠 유형이 charset을 지정하지 않는 한 요청 본문은 항상 UTF-8 인코딩된 문자열입니다.

```
"body":"body string"
```

응답

여기에서 예시를 확인하세요.

JavaScript Amazon RDS용 리졸버 함수 레퍼런스

AWS AppSync RDS 함수 및 리졸버를 사용하면 개발자가 RDS Data API를 사용하여 Amazon Aurora 클러스터 데이터베이스에 SQL 쿼리를 보내고 이러한 쿼리의 결과를 가져올 수 있습니다. 의 모듈 `sql` 태그가 지정된 템플릿을 사용하거나 AWS AppSync`rds` 모듈의,, 및 도우미 함수를 사용하여 데이터 API로 전송되는 SQL 명령문을 작성할 수 있습니다. `rds` `select` `insert` `update` `remove` AWS AppSync RDS 데이터 서비스의 `ExecuteStatement`작업을 활용하여 데이터베이스에 대해 SQL 문을 실행합니다.

주제

- SQL 태그가 지정된 템플릿
- 문 생성
- 데이터 불러오기
- 유틸리티 함수
- SQL Select
- SQL Insert
- SQL Update
- SQL Delete

- 캐스팅

SQL 태그가 지정된 템플릿

AWS AppSync의 sql 태그가 지정된 템플릿을 사용하면 템플릿 표현식을 사용하여 런타임에 동적 값을 받을 수 있는 정적 명령문을 만들 수 있습니다. AWS AppSync 표현식 값으로부터 변수 맵을 작성하여 Amazon Aurora 서버리스 데이터 API로 전송되는 SqlParameterized쿼리를 구성합니다. 이 방법을 사용하면 런타임에 전달된 동적 값이 원래 문을 수정할 수 없으며, 이로 인해 의도하지 않은 실행이 발생할 수 있습니다. 모든 동적 값은 파라미터로 전달되며 원래 문을 수정할 수 없으며 데이터베이스에서 실행되지 않습니다. 이렇게 하면 쿼리가 SQL 삽입 공격에 덜 취약해집니다.

> **Note**
>
> 모든 경우에 SQL 문을 작성할 때는 입력으로 받는 데이터를 적절하게 처리하기 위한 보안 지침을 따라야 합니다.

> **Note**
>
> sql 태그가 지정된 템플릿은 변수 값 전달만 지원합니다. 표현식을 사용하여 열 또는 테이블 이름을 동적으로 지정할 수 없습니다. 그러나 유틸리티 함수를 사용하여 다음 작업을 수행할 수도 있습니다.

다음 예시에서는 런타임 중 GraphQL 쿼리에 동적으로 설정된 col 인수 값을 기반으로 필터링하는 쿼리를 만듭니다. 해당 값은 태그 표현식을 사용해서만 문에 추가할 수 있습니다.

```
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);
}
```

변수 맵을 통해 모든 동적 값을 전달함으로써 데이터베이스 엔진을 사용하여 값을 안전하게 처리하고 삭제합니다.

문 생성

함수 및 해석기는 MySQL 및 PostgreSQL 데이터베이스와 상호 작용할 수 있습니다. 문을 작성할 때는 각각 `createMySQLStatement` 및 `createPgStatement`를 사용합니다. 예를 들어, `createMySQLStatement`는 MySQL 쿼리를 생성할 수 있습니다. 이러한 함수는 두 개까지의 문을 허용하므로 요청이 결과를 즉시 검색해야 할 때 유용합니다. MySQL으로 다음을 할 수 있습니다.

```
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` 및 `createMySQLStatement`는 `sql` 태그가 지정된 템플릿으로 작성된 문을 이스케이프 처리하거나 인용하지 않습니다.

데이터 불러오기

실행된 SQL 문의 결과는 `context.result` 객체의 응답 핸들러에서 확인할 수 있습니다. 결과는 `ExecuteStatement` 작업의 응답 요소가 포함된 JSON 문자열입니다. 파싱 시 결과는 다음 형태를 가집니다.

```
type SQLStatementResults = {
    sqlStatementResults: {
        records: any[];
        columnMetadata: any[];
        numberOfRecordsUpdated: number;
        generatedFields?: any[]
    }[]
}
```

반환된 행을 나타내는 JSON 객체 목록으로 `toJsonObject` 유틸리티를 사용하여 결과를 변환할 수 있습니다. 예:

```
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]
}
```

단, `toJsonObject`는 문 결과 배열을 반환합니다. 문 하나를 제공한 경우 배열 길이는 1입니다. 문 두 개를 제공한 경우 배열 길이는 2입니다. 배열의 각 결과에는 `0` 이상의 행이 포함됩니다. 결과 값이 유효하지 않거나 예상치 못한 경우 `toJsonObject`는 `null`을 반환합니다.

유틸리티 함수

AWS AppSync RDS 모듈의 유틸리티 도우미를 사용하여 데이터베이스와 상호 작용할 수 있습니다.

SQL Select

`select` 유틸리티는 관계형 데이터베이스를 쿼리하기 위한 SELECT 문을 생성합니다.

기본 사용

기본 형식에서 쿼리하려는 테이블을 지정할 수 있습니다.

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // "SELECT * FROM "persons"
    return createPgStatement(select({table: 'persons'}));
}
```

테이블 식별자에 스키마를 지정할 수도 있습니다.

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';
```

```
export function request(ctx) {

    // Generates statement:
    // SELECT * FROM "private"."persons"
    return createPgStatement(select({table: 'private.persons'}));
}
```

열 지정

columns 속성을 사용하여 열을 지정할 수 있습니다. 값으로 설정되지 않은 경우 기본값은 *입니다.

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name']
    }));
}
```

열 테이블도 지정할 수 있습니다.

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "persons"."name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'persons.name']
    }));
}
```

제한 및 오프셋

쿼리에 limit 및 offset을 적용할 수 있습니다.

```
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
    }));
}
```

순서 기준

orderBy 속성을 기준으로 결과를 정렬할 수 있습니다. 열을 지정하는 객체 배열과 선택적 dir 속성을 제공하십시오.

```
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'}]
    }));
}
```

필터

특수 조건 객체를 사용하여 필터를 구축할 수 있습니다.

```
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'}}
```

```
    }));
}
```

필터를 결합할 수도 있습니다.

```
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}}
    }));
}
```

OR 문을 작성할 수 있습니다.

```
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 } }
        ]}
    }));
}
```

not으로 조건을 무효화할 수도 있습니다.

```
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 } }
        ]}
    }));
}
```

다음 작업을 사용하여 값을 비교할 수도 있습니다.

연산자	설명	가능한 값 유형
eq	같음	숫자, 문자열, 부울
ne	같지 않음	숫자, 문자열, 부울
le	작거나 같음	숫자, 문자열
lt	보다 작음	숫자, 문자열
ge	크거나 같음	숫자, 문자열
gt	보다 큼	숫자, 문자열
다음을 포함함	좋아요	문자열
포함되지 않음	마음에 들지 않음	문자열
로 시작	접두사로 시작	문자열
사이	두 값 사이	숫자, 문자열
속성이 존재함	속성이 null이 아닙니다.	숫자, 문자열, 부울
size	요소의 길이를 확인합니다.	문자열

SQL Insert

이 `insert` 유틸리티는 `INSERT` 작업과 함께 데이터베이스에 단일 행 항목을 삽입하는 간단한 방법을 제공합니다.

단일 항목 삽입

항목을 삽입하려면 테이블을 지정한 다음 값 객체를 전달하십시오. 객체 키는 테이블 열에 매핑됩니다. 열 이름은 자동으로 이스케이프 처리되고 변수 맵을 사용하여 값이 데이터베이스로 전송됩니다.

```
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)
}
```

MySQL 사용 사례

`select`가 뒤따르는 `insert`를 결합하여 삽입된 행을 검색할 수 있습니다.

```
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)
}
```

Postgres 사용 사례

Postgres를 사용하면 삽입한 행에서 데이터를 가져오는 데 returning을 사용할 수 있습니다. * 또는 열 이름 배열을 받아들입니다.

```
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)
}
```

SQL Update

update 유틸리티를 사용하여 기존 행을 업데이트할 수 있습니다. 조건 객체를 사용하여 조건을 충족하는 모든 행의 지정된 열에 변경 내용을 적용할 수 있습니다. 예를 들어 이러한 변형을 만들 수 있는 스키마가 있다고 가정해 보겠습니다. 해당 2000 연도부터 인식(known_since)하게 된 경우에만 Person의name을 3의 id 값으로 업데이트하려고 합니다.

```
mutation Update {
    updatePerson(
        input: {id: 3, name: "Jon"},
        condition: {known_since: {ge: "2000"}}
    ) {
    id
    name
  }
```

```
}
```

업데이트 해석기는 다음과 같이 생겼습니다.

```
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)
}
```

조건에 검사를 추가하여 3와 동등한 프라이머리 키 id를 가진 행만 업데이트되도록 할 수 있습니다. 마찬가지로 Postgres inserts의 경우 수정된 데이터를 반환하는 데 returning을 사용할 수 있습니다.

SQL Delete

remove 유틸리티를 사용하여 기존 행을 제거할 수 있습니다. 조건을 충족하는 모든 행에서 조건 객체를 사용할 수 있습니다. 참고로 delete 이 키워드는 에서 예약된 JavaScript 키워드입니다. remove대신 사용해야 합니다.

```
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)
}
```

캐스팅

올바른 객체 유형을 더 구체적으로 지정하여 문에 사용하려는 경우가 있을 수 있습니다. 제공된 유형 힌트를 사용하여 매개변수 유형을 지정할 수 있습니다. AWS AppSync 데이터 API와 동일한 유형 힌트를 지원합니다. AWS AppSync `rds`모듈의 `typeHint` 함수를 사용하여 파라미터를 캐스팅할 수 있습니다.

다음 예시에서는 JSON 객체로 캐스팅된 값으로 배열을 보낼 수 있습니다. `->` 연산자를 사용하여 JSON 배열의 `index` 2에서 요소를 검색합니다.

```
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
}
```

캐스팅은 `DATE`, `TIME` 및 `TIMESTAMP` 등을 처리하고 비교할 때도 유용합니다.

```
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)
}
```

다음에서는 현재 날짜와 시간을 보낼 수 있는 방법을 보여줍니다.

```
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)}`)
}
```

사용 가능한 유형 힌트

- `typeHint.DATE` – 해당 파라미터는 `DATE` 유형의 객체로 데이터베이스에 전송됩니다. 승인된 형식은 `YYYY-MM-DD`입니다.
- `typeHint.DECIMAL` – 해당 파라미터는 `DECIMAL` 유형의 객체로 데이터베이스에 전송됩니다.
- `typeHint.JSON` – 해당 파라미터는 `JSON` 유형의 객체로 데이터베이스에 전송됩니다.
- `typeHint.TIME` – 해당 문자열 파라미터 값은 `TIME` 유형의 객체로 데이터베이스에 전송됩니다. 승인된 형식은 `HH:MM:SS[.FFF]`입니다.
- `typeHint.TIMESTAMP` – 해당 문자열 파라미터 값은 `TIMESTAMP` 유형의 객체로 데이터베이스에 전송됩니다. 승인된 형식은 `YYYY-MM-DD HH:MM:SS[.FFF]`입니다.
- `typeHint.UUID` – 해당 문자열 파라미터 값은 `UUID` 유형의 객체로 데이터베이스에 전송됩니다.

해석기 매핑 템플릿 참조(VTL)

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

다음 섹션에서는 매핑 템플릿에서 유틸리티 작업을 사용하는 방법을 설명합니다.

주제

- 해석기 매핑 템플릿 개요
- 해석기 매핑 템플릿 프로그래밍 안내서
- 해석기 매핑 템플릿 컨텍스트 참조
- 해석기 매핑 템플릿 유틸리티 참조
- DynamoDB에 대한 해석기 매핑 템플릿 참조
- RDS에 대한 해석기 매핑 템플릿 참조
- OpenSearch에 대한 해석기 매핑 템플릿 참조입니다.
- Lambda용 해석기 매핑 템플릿 참조
- 에 대한 리졸버 매핑 템플릿 참조 EventBridge
- None 데이터 소스에 대한 해석기 매핑 템플릿 참조
- HTTP에 대한 해석기 매핑 템플릿 참조
- 해석기 매핑 템플릿 변경 로그

해석기 매핑 템플릿 개요

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync에서는 리소스에 대한 작업을 수행하여 GraphQL 요청에 응답할 수 있습니다. 쿼리 또는 변형을 실행하려는 각 GraphQL 필드에 대해 데이터 소스와 통신하려면 해석기를 연결해야 합니다. 일반적으로 이 통신은 데이터 소스에 대해 고유한 파라미터 또는 작업을 통해 이루어집니다.

해석기는 GraphQL과 데이터 소스를 연결하는 커넥터입니다. 들어오는 GraphQL 요청을 백엔드 데이터 소스에 대한 지침으로 변환하는 방법과 해당 데이터 소스를 GraphQL 응답으로 변환하는 방법을 AWS AppSync에 알려줍니다. 매핑 템플릿은 Apache Velocity Template Language(VTL)로 작성되는데, VTL에서는 요청을 입력으로 받아서 해석기에 대한 명령문을 포함하는 JSON 문서를 출력합니다. GraphQL 필드의 인수 전달과 같은 단순 명령은 물론, 항목을 DynamoDB에 삽입하기 전에 인수를 루핑하여 항목을 빌드하는 등 복잡한 명령에도 매핑 템플릿을 사용할 수 있습니다.

AWS AppSync에는 매핑 템플릿을 약간 다른 방식으로 활용하는 두 가지 유형이 있습니다.

- 유닛 해석기
- 파이프라인 해석기

유닛 해석기

유닛 해석기는 요청 및 응답 템플릿만 포함하는 독립형 객체입니다. 단일 데이터 원본에서 항목 나열과 같은 간단한 단일 작업에 사용합니다.

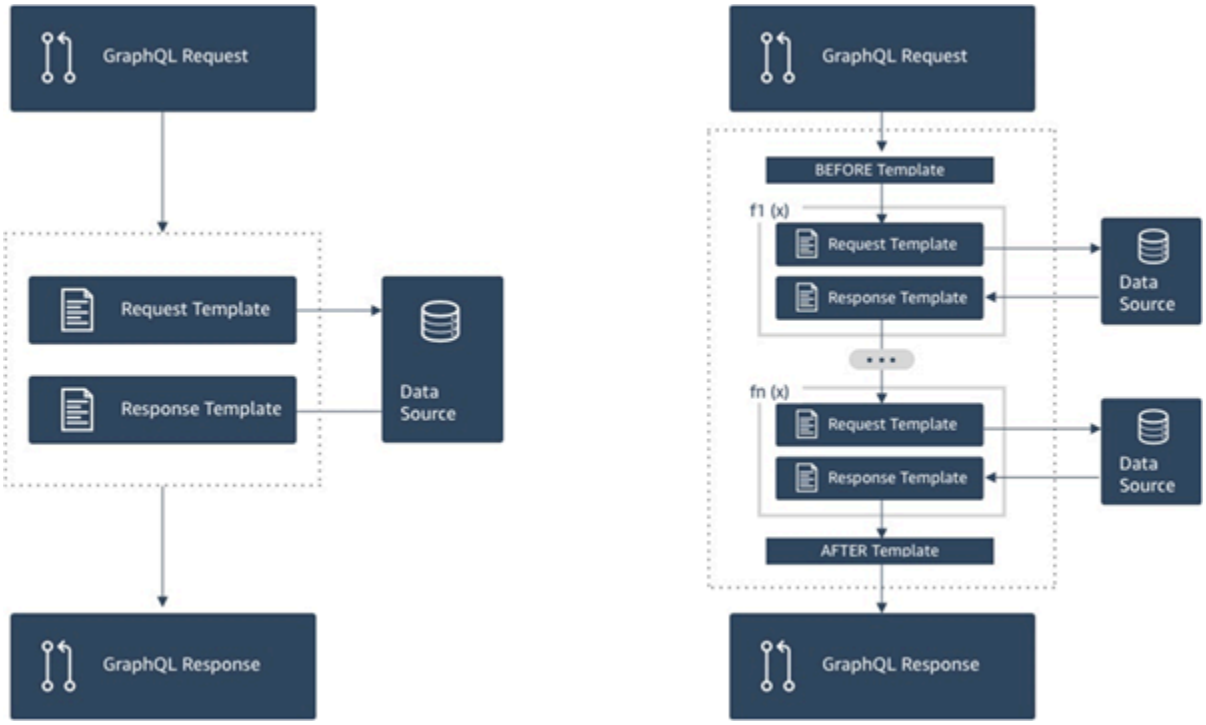
- 요청 템플릿: GraphQL 작업이 구문 분석된 후 들어오는 요청을 받아서 선택한 데이터 소스 작업에 대한 요청 구성으로 변환합니다.
- 응답 템플릿: 데이터 소스의 응답을 해석하여 GraphQL 필드 출력 유형의 셰이프에 매핑합니다.

파이프라인 해석기

파이프라인 해석기에는 순차적으로 실행되는 함수가 하나 이상 포함되어 있습니다. 각 함수에는 요청 템플릿과 응답 템플릿이 포함되어 있습니다. 파이프라인 해석기에는 템플릿에 포함된 함수 시퀀스를 둘러싸는 이전 템플릿과 이후 템플릿도 있습니다. 이후 템플릿은 GraphQL 필드 출력 유형에 매핑됩니다. 파이프라인 해석기는 응답 템플릿이 출력을 매핑하는 방식이 유닛 해석기와 다릅니다. 파이프라인 해석기는 다른 함수의 입력값이나 파이프라인 해석기의 이후 템플릿을 포함하여 원하는 모든 출력에 매핑할 수 있습니다.

파이프라인 해석기 함수를 사용하면 스키마의 여러 해석기에서 재사용할 수 있는 공통 로직을 작성할 수 있습니다. 함수는 데이터 소스에 직접 연결되고 유닛 해석기와 마찬가지로 동일한 요청 및 응답 매핑 템플릿 형식을 포함하고 있습니다.

다음 다이어그램은 왼쪽은 유닛 해석기, 오른쪽은 파이프라인 해석기의 프로세스 흐름을 보여줍니다.



파이프라인 해석기에는 유닛 해석기가 지원하는 기능의 상위 집합이 포함되어 있지만 약간 더 복잡합니다.

파이프라인 해석기의 구조

파이프라인 해석기는 이전 매핑 템플릿, 이후 매핑 템플릿과 함수 목록으로 구성됩니다. 각 함수에는 데이터 소스에 대해 실행되는 요청 및 응답 매핑 템플릿이 있습니다. 파이프라인 해석기는 함수 목록에 실행을 위임하기 때문에 자신은 어떠한 데이터 원본에도 연결되지 않습니다. 유닛 해석기와 함수는 데이터 원본에 대해 작업을 실행하는 기본 요소입니다. 자세한 내용은 해석기 매핑 템플릿 개요를 참조하세요.

이전 매핑 템플릿

이 파이프라인 해석기의 요청 매핑 템플릿(또는 이전 단계)을 사용하면 정의된 함수를 실행하기 전에 몇 가지 준비 로직을 수행할 수 있습니다.

함수 목록

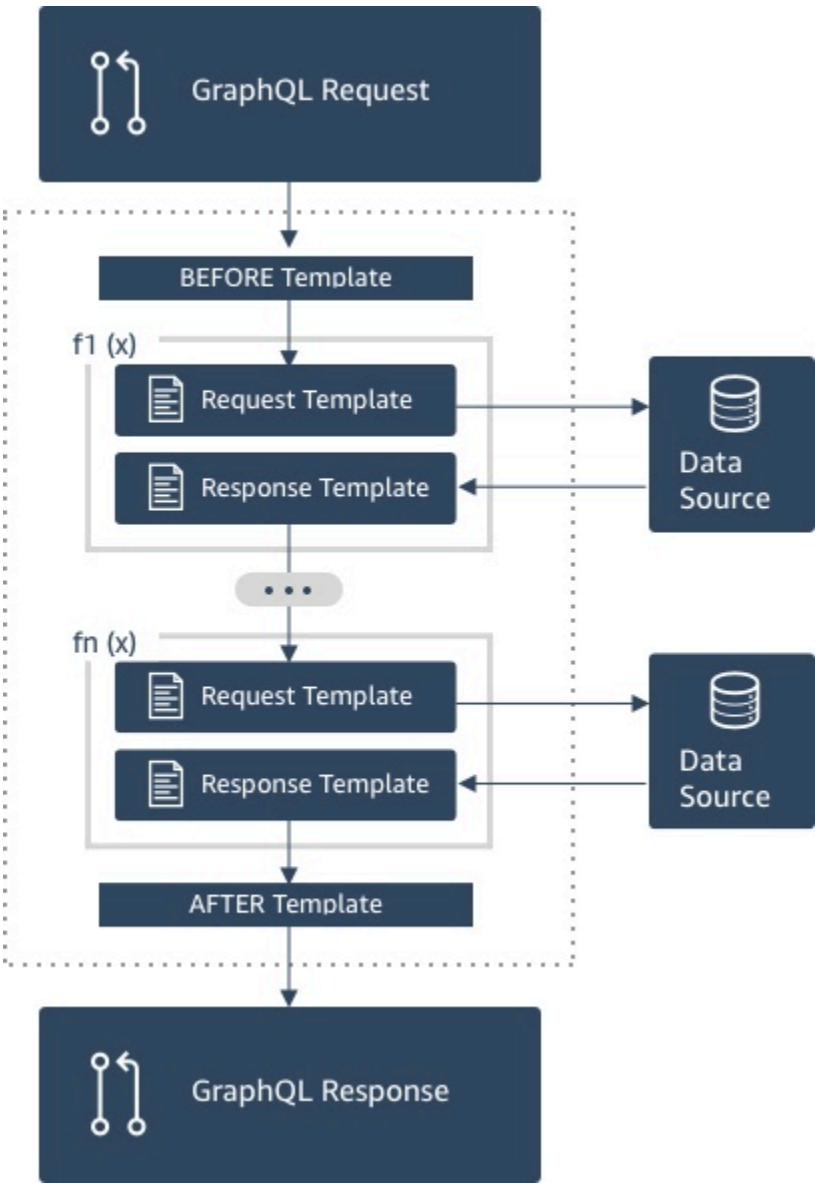
파이프라인 해석기가 순서대로 실행하는 함수 목록입니다. 파이프라인 해석기 요청 매핑 템플릿 평가 결과는 첫 번째 함수에 `$ctx.prev.result`로 사용할 수 있습니다. 각 함수 출력은 다음 함수에 `$ctx.prev.result`로 사용할 수 있습니다.

After 매핑 템플릿

이 파이프라인 해석기의 응답 매핑 템플릿(또는 이후 단계)을 사용하면 마지막 함수의 출력에서 예상 GraphQL 필드 유형으로 몇 가지 최종 매핑 로직을 수행할 수 있습니다. 함수 목록의 마지막 함수 출력은 파이프라인 해석기 매핑 템플릿에서 `$ctx.prev.result` 또는 `$ctx.result`로 사용할 수 있습니다.

실행 흐름

함수 2개로 구성된 파이프라인 해석기의 경우, 아래 목록은 해석기 호출 시 실행 흐름을 나타냅니다.



1. 파이프라인 해석기 이전 매핑 템플릿
2. 함수 1: 함수 요청 매핑 템플릿

3. 함수 1: 데이터 원본 호출
4. 함수 1: 함수 응답 매핑 템플릿
5. 함수 2: 함수 요청 매핑 템플릿
6. 함수 2: 데이터 원본 호출
7. 함수 2: 함수 응답 매핑 템플릿
8. 파이프라인 해석기 이후 매핑 템플릿

> **Note**
>
> 파이프라인 해석기 실행 흐름은 단방향이고 해석기에서 정적으로 정의됩니다.

유용한 Apache Velocity Template Language(VTL) 유틸리티

애플리케이션의 복잡성이 증가함에 따라 개발 생산성을 높이기 위해 여기서 VTL 유틸리티 및 명령이 사용됩니다. 다음 유틸리티는 파이프라인 해석기로 작업할 때 유용할 수 있습니다.

$ctx.stash

stash는 각 해석기 및 함수 매핑 템플릿 내에서 사용할 수 있는 `Map`입니다. 단일 해석기가 실행되는 동안에는 동일한 stash 인스턴스가 사용됩니다. 이는 파이프라인 해석기 내에서 stash를 사용해 요청 및 응답 매핑 템플릿 간에 그리고 함수 간에 임의 데이터를 전달할 수 있다는 의미입니다. stash는 동일한 메서드를 Java 맵 데이터 구조로 노출합니다.

$ctx.prev.result

`$ctx.prev.result`는 파이프라인 해석기에서 실행된 이전 작업의 결과를 나타냅니다.

이전 작업이 파이프라인 해석기의 이전 매핑 템플릿인 경우 `$ctx.prev.result`는 템플릿 평가의 출력을 나타내며 파이프라인의 첫 번째 함수에서 사용할 수 있게 됩니다. 이전 작업이 첫 번째 함수였으면 `$ctx.prev.result`는 첫 번째 함수의 결과를 나타내고, 파이프라인의 두 번째 함수에 사용할 수 있습니다. 이전 작업이 마지막 함수였으면 `$ctx.prev.result`는 마지막 함수의 결과를 나타내고, 파이프라인 해석기 이후 매핑 템플릿에 사용할 수 있습니다.

#return(data: Object)

`#return(data: Object)` 명령은 임의의 매핑 템플릿에서 영구적으로 반환해야 하는 경우 유용합니다. 로직의 가장 가까운 범위 지정 블록에서 반환하기 때문에 `#return(data: Object)`은 프로그래밍 언어의 return 키워드와 유사합니다. 이는 해석기 매핑 템플릿 내에서 `#return`을 사용하면

해석기에서 반환함을 의미합니다. 해석기 매핑 템플릿에서 `#return(data: Object)`를 사용하면 GraphQL 필드에 `data`가 설정됩니다. 또는 함수 매핑 템플릿의 `#return(data: Object)`를 사용하면 함수에서 반환하고, 파이프라인의 다음 함수 또는 해석기 응답 매핑 템플릿으로 실행합니다.

#return

이는 `#return(data: Object)`와 동일하지만 대신 `null`이 반환됩니다.

$util.error

`$util.error` 유틸리티는 필드 오류를 발생시키는 데 유용합니다. 함수 매핑 템플릿 내에서 `$util.error`를 사용하면 즉시 필드 오류가 발생해 후속 함수가 실행되지 않도록 방지합니다. 세부 정보 및 기타 `$util.error` 서명은 해석기 매핑 템플릿 유틸리티 참조를 참조하세요.

$util.appendError

`$util.appendError`는 `$util.error()`와 유사하지만 매핑 템플릿 평가를 중단하지 않는다는 큰 차이점이 있습니다. 대신 필드에 오류가 있다는 신호를 보내지만 템플릿이 계속해서 평가되도록 하고 이어서 데이터를 반환합니다. 함수 내에서 `$util.appendError`를 사용하면 파이프라인의 실행 흐름이 중단되지 않습니다. 세부 정보 및 기타 `$util.error` 서명은 해석기 매핑 템플릿 유틸리티 참조를 참조하세요.

템플릿 예제

다음 GraphQL 쿼리를 사용하여 `Post` 유형을 반환하는 `getPost(id:ID!)`라는 필드에 DynamoDB 데이터 소스와 단위 해석기가 있다고 가정해 보겠습니다.

```
getPost(id:1){
    id
    title
    content
}
```

해석기 템플릿은 다음과 같습니다.

```
    {
    "version" : "2018-05-29",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
```

```
}
```

이 템플릿은 id 입력 파라미터 값 1을 ${ctx.args.id} 대신에 사용하고 다음 JSON을 생성합니다.

```
     {
    "version" : "2018-05-29",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "1" }
    }
}
```

AWS AppSync에서는 이 템플릿을 사용하여 DynamoDB와 통신하고 데이터를 가져오는(또는 해당되는 경우 다른 작업을 수행하는) 명령을 생성합니다. 데이터가 반환된 후에는 AWS AppSync에서 선택적 응답 매핑 템플릿을 실행합니다. 이 템플릿은 데이터 셰이핑 또는 로직을 수행할 때 사용할 수 있습니다. 예를 들어 DynamoDB에서 결과를 가져오는 경우 결과는 다음과 같습니다.

```
{
        "id" : 1,
        "theTitle" : "AWS AppSync works offline!",
        "theContent-part1" : "It also has realtime functionality",
        "theContent-part2" : "using GraphQL"
}
```

다음 응답 매핑 템플릿을 사용하여 필드 중 두 개를 단일 필드로 결합할 수 있습니다.

```
{
        "id" : $util.toJson($context.data.id),
        "title" : $util.toJson($context.data.theTitle),
        "content" : $util.toJson("${context.data.theContent-part1}
 ${context.data.theContent-part2}")
}
```

템플릿이 데이터에 적용된 이후의 데이터 형태 형성은 다음과 같습니다.

```
{
        "id" : 1,
        "title" : "AWS AppSync works offline!",
        "content" : "It also has realtime functionality using GraphQL"
}
```

이 데이터는 다음과 같이 클라이언트에 대한 응답으로 제공됩니다.

```
{
        "data": {
                "getPost":      {
                        "id" : 1,
                        "title" : "AWS AppSync works offline!",
                        "content" : "It also has realtime functionality using GraphQL"
                }
        }
}
```

대부분 환경에서 응답 매핑 템플릿은 데이터의 단순 전달로, 개별 항목 또는 항목 목록을 반환하는 경우 대부분 달라집니다. 개별 항목에 대한 전달:

```
$util.toJson($context.result)
```

목록에 대한 전달:

```
$util.toJson($context.result.items)
```

유닛 해석기와 파이프라인 해석기 둘 다에 대한 예를 더 보려면 해석기 자습서를 참조하세요.

평가된 맵핑 템플릿 역직렬화 규칙

매핑 템플릿은 문자열로 평가됩니다. AWS AppSync에서 출력 문자열은 JSON 구조를 따라야 유효합니다.

또한 다음과 같은 역직렬화 규칙이 적용됩니다.

JSON 객체에는 중복 키가 허용되지 않습니다.

평가된 매핑 템플릿 문자열이 JSON 객체를 나타내거나 중복 키가 있는 객체를 포함하는 경우 매핑 템플릿은 다음 오류 메시지를 반환합니다.

`Duplicate field 'aField' detected on Object. Duplicate JSON keys are not allowed.`

평가된 요청 매핑 템플릿의 중복 키 예:

```
{
    "version": "2018-05-29",
```

```
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "1",
        "field": "getPost" ## key 'field' has been redefined
    }
}
```

이 오류를 수정하려면 JSON 객체에서 키를 다시 정의하지 마십시오.

후행 문자는 JSON 객체에서 허용되지 않습니다.

평가된 매핑 템플릿 문자열이 JSON 객체를 나타내며 후행 외부 문자를 포함하는 경우 매핑 템플릿은 다음 오류 메시지를 반환합니다.

`Trailing characters at the end of the JSON string are not allowed.`

평가된 요청 매핑 템플릿의 후행 문자 예:

```
{
    "version": "2018-05-29",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "1",
    }
}extraneouschars
```

이 오류를 수정하려면 평가된 템플릿이 JSON으로 엄격하게 평가되는지 확인합니다.

해석기 매핑 템플릿 프로그래밍 안내서

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

이 안내서는 AWS AppSync에서 Apache Velocity Template Language(VTL)를 이용한 프로그래밍을 설명하는 쿡북 스타일의 자습서입니다. JavaScript, C 또는 Java 같은 다른 프로그래밍에 대해 잘 알고 있다면 매우 쉬울 것입니다.

AWS AppSync에서는 VTL을 사용하여 클라이언트의 GraphQL 요청을 데이터 소스에 대한 요청으로 변환합니다. 그런 다음 이 프로세스를 역순으로 수행하여 데이터 원본 응답을 GraphQL 응답으로 다시 변환합니다. VTL은 다음과 같은 기술을 사용하여 웹 애플리케이션의 표준 요청/응답 흐름 요청과 응답을 둘 다 처리할 수 있는 논리적인 템플릿 언어입니다.

- 새 항목의 기본값
- 입력 확인 및 서식 지정
- 데이터 변환 및 셰이핑
- 목록, 맵 및 배열을 반복하여 값을 추출하거나 변경
- 사용자 자격 증명 기반 응답 필터링/변경
- 복잡한 권한 부여 검사

예를 들면 GraphQL 인수에서 서비스의 전화번호를 확인하거나 DynamoDB에 저장하기 전에 입력 파라미터를 대문자로 변환해야 할 수도 있습니다. 또는 클라이언트 시스템에서 GraphQL 인수, JWT 토큰 클레임 또는 HTTP 헤더에 코드를 포함시켜 제공하도록 하거나, 코드가 목록의 특정 문자열과 일치할 경우에만 데이터로 응답하도록 할 수 있습니다. 이 모든 것이 AWS AppSync에서 VTL을 사용하여 수행할 수 있는 로직 점검입니다.

VTL에서는 익숙한 프로그래밍 기술로 로직을 적용할 수 있습니다. 하지만 사용자 기반의 성장에 따라 GraphQL API를 확장하려면 표준 요청/응답 흐름 내에서만 실행해야 합니다. AWS AppSync는 해석기로 AWS Lambda도 지원하므로 유연성이 더 많이 필요할 경우 선택한 프로그래밍 언어(Node.js, Python, Go, Java 등)로 Lambda 함수를 작성할 수 있습니다.

설정

언어 학습 시 사용하는 일반적인 기술 중 하나는 결과(예: JavaScript의 `console.log(variable)`)를 출력하여 어떻게 되는지 보는 것입니다. 이 자습서에서는 단순 GraphQL 스키마를 생성하고 값 맵을 Lambda 함수에 전달함으로써 이 기술을 설명합니다. Lambda 함수가 값을 출력하고 나서 값에 응답합니다. 따라서 요청/응답 흐름을 이해하고 다양한 프로그래밍 기법을 볼 수 있습니다.

다음 GraphQL 스키마를 생성하는 것부터 시작합니다.

```
type Query {
    get(id: ID, meta: String): Thing
}

type Thing {
```

```
    id: ID!
    title: String!
    meta: String
}

schema {
    query: Query
}
```

지금 Node.js를 언어로 사용하여 다음 AWS Lambda 함수를 생성합니다.

```
exports.handler = (event, context, callback) => {
    console.log('VTL details: ', event);
    callback(null, event);
};
```

AWS AppSync 콘솔의 데이터 소스 패널에서 이 Lambda 함수를 새 데이터 소스로 추가합니다. 콘솔의 스키마 페이지로 돌아가서 오른쪽 `get(...):Thing` 쿼리 옆에 있는 연결 버튼을 클릭합니다. 요청 템플릿으로는 Invoke and forward arguments(인수 호출 및 전달) 메뉴에서 기존 템플릿을 선택합니다. 응답 템플릿으로는 Lambda 결과 반환(Return Lambda result)을 선택합니다.

한 위치에서 Lambda 함수에 대한 Amazon CloudWatch Logs를 열고 AWS AppSync 콘솔의 쿼리 탭에서 다음 GraphQL 쿼리를 실행합니다.

```
query test {
  get(id:123 meta:"testing"){
    id
    meta
  }
}
```

Lambda 함수가 `id:123` 및 `meta:testing`를 다시 되풀이하여 표시하므로 GraphQL 응답에 이 두 요소가 포함되어야 합니다. 몇 초 후에는 CloudWatch Logs에서 이러한 세부 정보와 함께 레코드를 볼 수 있습니다.

Variables

VTL은 참조를 사용합니다. 이 참조는 데이터를 저장하거나 조작하는 데 사용할 수 있습니다. VTL에는 변수, 속성 및 메서드라는 세 가지 유형이 있습니다. 변수는 앞에 $ 기호가 붙으며 `#set` 지시문을 사용하여 생성됩니다.

```
#set($var = "a string")
```

변수는 숫자, 문자열, 배열, 목록 및 맵 등 다른 언어에서 친숙한 유형과 비슷한 유형을 저장합니다. Lambda 해석기의 기본 요청 템플릿에서 JSON 페이로드가 전송됨을 눈치채셨을 것입니다.

```
"payload": $util.toJson($context.arguments)
```

여기에는 몇 가지 주목해야 할 사항이 있는데, 첫 번째, AWS AppSync에서는 일반 작업에 사용되는 여러 편의 함수를 제공합니다. 이 예에서는 `$util.toJson`이 변수를 JSON으로 변환합니다. 두 번째, `$context.arguments` 변수가 GraphQL 요청에서 맵 객체로 자동 채워집니다. 다음과 같이 새 맵을 생성할 수 있습니다.

```
#set( $myMap = {
  "id": $context.arguments.id,
  "meta": "stuff",
  "upperMeta" : $context.arguments.meta.toUpperCase()
} )
```

이제 `$myMap`이라는 변수를 만들었으며, 이 변수에는 `id`, `meta` 및`upperMeta` 키가 있습니다. 이 코드는 몇 가지 다른 것도 보여줍니다.

- `id`가 GraphQL 인수의 키로 채워집니다. VTL에서는 클라이언트의 인수를 가져오기 위해 일반적으로 이렇게 작동합니다.
- `meta`가 기본값을 보여주는 값으로 하드코딩됩니다.
- `upperMeta`가 `meta` 메서드를 사용하여 `.toUpperCase()` 인수로 변환됩니다.

요청 템플릿 맨 위에 이전 코드를 추가하고 `payload`를 새 `$myMap` 변수로 변경합니다.

```
"payload": $util.toJson($myMap)
```

Lambda 함수를 실행하고 나면 CloudWatch logs에서 응답 변경 내용은 물론 이 데이터도 확인할 수 있습니다. 이 자습서의 나머지를 진행하면서 비슷한 테스트를 실행할 수 있도록 `$myMap`을 채운 상태로 유지하겠습니다.

또한 변수에서 properties_를 설정할 수도 있습니다. 이러한 항목은 단순 문자열, 배열 또는 JSON입니다.

```
#set($myMap.myProperty = "ABC")
```

```
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])
#set($myMap.jsonProperty = {
    "AppSync" : "Offline and Realtime",
    "Cognito" : "AuthN and AuthZ"
})
```

자동 참조

VTL은 기본적으로 템플릿 언어이므로 개발자가 제공하는 모든 참조가 `.toString()`을 수행합니다. 참조가 정의되지 않은 경우 실제 참조 표현을 문자열로 출력합니다. 예:

```
#set($myValue = 5)
##Prints '5'
$myValue

##Prints '$somethingelse'
$somethingelse
```

이를 해결하도록 VTL에 quiet reference 또는 silent reference 구문이 있으며, 이 구문은 템플릿 엔진에 이 동작을 숨기도록 지시합니다. 이에 대한 구문은 `$!{}`입니다. 예를 들어 `$!{somethingelse}`를 사용하도록 이전 코드를 조금 변경한 경우 출력이 표시되지 않습니다.

```
#set($myValue = 5)
##Prints '5'
$myValue

##Nothing prints out
$!{somethingelse}
```

메서드 호출

이전 예제에서 변수를 생성하고 동시에 값을 설정하는 방법을 보여드렸습니다. 맵에 데이터를 추가하여 이 작업을 다음과 같이 두 단계로 수행할 수도 있었습니다.

```
#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")}
```

하지만 이 동작에 대해 알아야 할 것이 있습니다. 위와 같이 자동 참조 표기 `$!{}`를 사용하여 메서드를 호출할 수 있더라도, 실행된 메서드의 반환 값은 숨길 수 없습니다. 위에서 `##Prints "first value"` 및 `##Prints true`를 사용한 이유는 바로 이 때문입니다. 이 문으로 키가 이미 존재하는 값을 삽입하는 등 맵이나 목록을 반복하려 할 때 오류가 발생할 수 있습니다. 출력이 평가 시 예기치 않은 문자열을 템플릿에 추가하기 때문입니다.

경우에 따라 `#set` 지시문을 사용하여 메서드를 호출하고 변수를 무시하는 것이 이 문제를 해결하는 방법이 되기도 합니다. 예:

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

템플릿에 예상치 못한 문자열이 인쇄되는 것을 방지할 수 있으므로 템플릿에서 이 기법을 사용할 수 있습니다. AWS AppSync는 동일한 동작을 보다 간결한 표기법으로 제공하는 대체 편의 함수를 제공합니다. 이 함수를 사용하면 이러한 구현 세부 사항에 대해 개발자가 생각할 필요가 없습니다. `$util.quiet()` 또는 `$util.qr()` 별칭으로 이 함수에 액세스할 수 있습니다. 예:

```
#set ($myMap = {})
#set ($myList = [])

##Nothing prints out
$util.quiet($myMap.put("id", "first value"))
##Nothing prints out
$util.qr($myList.add("something"))
```

문자열

여러 프로그래밍 언어와 마찬가지로, 문자열을 다루기가 어려울 수 있으며, 특히 변수에서 문자열을 구성하려는 경우에는 더욱 더 그렇습니다. VTL과 함께 제공되는 공통된 사항이 있습니다.

DynamoDB 같은 데이터 소스에 데이터를 문자열로 삽입하려 하지만 이 문자열이 GraphQL 인수 같은 변수에서 채워진다고 가정해 보겠습니다. 문자열에는 큰따옴표가 있으며 문자열에서 변수를 참조하려면 `"${}"`만 있으면 됩니다(따라서 자동 참조 표기에서는 `!`가 없습니다). 이 형태는 JavaScript의 템플릿 리터럴과 비슷합니다(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals).

```
#set($firstname = "Jeff")
```

```
$!{myMap.put("Firstname", "${firstname}")}
```

"id" : { "S" : "$util.autoId()"}와 같은 자동 ID를 생성하려 할 때 또는 GraphQL 클라이언트에서 인수를 사용할 때 "author": { "S" : "${context.arguments.author}"}처럼 DynamoDB 요청 템플릿에서 이 형태를 볼 수 있습니다. 다시 말해서 문자열 내부에서 메서드의 결과나 변수를 참조하여 데이터를 채울 수 있다는 뜻입니다.

또한 하위 문자열을 추출하는 등 Java String 클래스의 퍼블릭 메서드를 사용할 수도 있습니다.

```
#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}"))
```

문자열 연결도 매우 일반적인 작업입니다. 변수 참조만을 사용하여 또는 정적 값과 함께 이 작업을 수행할 수 있습니다.

```
#set($s1 = "Hello")
#set($s2 = " World")

$util.qr($myMap.put("concat","$s1$s2"))
$util.qr($myMap.put("concat2","Second $s1 World"))
```

Loops

이제 변수를 생성하고 메서드를 호출했으므로 몇 가지 로직을 코드에 추가할 수 있습니다. 다른 언어와는 달리, VTL에서는 루프만 허용됩니다. 루프에서 반복 횟수는 미리 지정되어 있습니다. Velocity에는 do..while이 없습니다. 이 설계는 평가 프로세스가 항상 종료되도록 하고 GraphQL 작업 실행 시 확장성 범위를 제공합니다.

루프는 #foreach와 함께 생성되며 루프 변수와 반복 가능한 객체(예: 배열, 목록, 맵 또는 모음)를 제공해야 합니다. #foreach 루프와 관련된 클래식 프로그래밍 예제는 모음에서 항목을 반복하고 출력하는 것이므로, 여기서는 이러한 항목을 추출하여 맵에 추가합니다.

```
#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
```

이 예제는 몇 가지 사항을 보여줍니다. 첫 번째는 범위 `[..]` 연산자와 함께 변수를 사용하여 반복 가능한 객체를 생성합니다. 그러고 각 항목이 작업에 사용할 수 있는 변수 `$i`에 의해 참조됩니다. 이전 예제에서는 2파운드 ##으로 표시된 주석도 볼 수 있습니다. 또한 이 예제는 키 또는 값의 루프 변수 사용과 문자열을 사용한 다양한 연결 방법도 보여줍니다.

`$i`는 정수이므로 `.toString()` 메서드를 호출할 수 있습니다. GraphQL INT 유형의 경우 이 메서드가 유용할 수 있습니다.

또한 범위 연산자를 직접 사용할 수도 있습니다. 예를 들면 다음과 같습니다.

```
#foreach($item in [1..5])
    ...
#end
```

배열

지금까지는 맵을 잘 조작해 왔지만, VTL에서는 배열도 일반적으로 사용합니다. 아래와 같이 배열을 사용하여 `.isEmpty()`, `.size()`, `.set()`, `.get()` 및 `.add()` 같은 기본 메서드에 액세스할 수도 있습니다.

```
#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)))
```

이전 예에서는 배열 인덱스 표기를 사용하여 arr2[$idx]가 있는 요소를 검색했습니다. 비슷한 방법으로 맵/딕셔너리에서 이름별로 조회할 수 있습니다.

```
#set($result = {
    "Author" : "Nadia",
    "Topic" : "GraphQL"
})

$util.qr($myMap.put("Author", $result["Author"]))
```

이 방법은 조건 사용 시 응답 템플릿의 데이터 원본에서 가져오는 결과를 필터링할 때 매우 일반적으로 사용합니다.

조건부 확인

#foreach에 관한 이전 단원에서는 VTL에서 로직을 사용하여 데이터를 변환하는 몇 가지 예를 살펴보았습니다. 또한 실행 시간에 조건부 확인을 적용하여 데이터를 평가할 수 있습니다.

```
#if(!$array.isEmpty())
    $util.qr($myMap.put("ifCheck", "Array not empty"))
#else
    $util.qr($myMap.put("ifCheck", "Your array is empty"))
#end
```

부울 표현식에 대한 위의 #if() 점검은 잘 작동하지만, 연산자와 #elseif()를 사용하여 분기시킬 수도 있습니다.

```
#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
```

이 두 가지 예제에는 부정(!) 및 같음(==)이 나와 있습니다. 또한 ||, &&, >, <, >=, <= 및 !=을 사용할 수 있습니다.

```
#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
```

참고: Boolean.FALSE 및 null만 조건문에서 false로 간주됩니다. 제로(0)와 빈 문자열("")은 false로 간주되지 않습니다.

연산자

작업자가 연산자 없이 산술 작업을 수행할 수 있는 완벽한 프로그래밍 언어는 없습니다. 다음은 시작하는 데 도움이 될 몇 가지 예제입니다.

```
#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))
```

루프와 조건부 함께 사용

데이터 원본 쓰기 또는 읽기 등 VTL에서 데이터를 변환할 때는 객체를 루핑하고 나서 확인을 수행한 다음 작업을 수행하는 것이 일반적입니다. 이전 단원의 도구 중 일부를 결합하여 다양한 기능을 이용할 수 있습니다. 한 가지 유용한 점이라면, #foreach가 각 항목에 .count를 자동으로 제공한다는 사실입니다.

```
#foreach ($item in $arr2)
```

```
  #set($idx = "item" + $foreach.count)
  $util.qr($myMap.put($idx, $item))
#end
```

예를 들면 단순히 맵에서 특정 크기 이하인 값을 추출하고자 하는 경우가 있습니다. 이 개수 값에 조건문 및 #break 문을 사용하여 다음을 수행할 수 있습니다.

```
#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
```

이전 #foreach는 .keySet()를 사용하여 반복되는 데, 이것을 맵에서 사용할 수 있습니다. 이렇게 하면 $key를 가져와서 .get($key)를 사용하여 값을 참조할 수 있는 액세스 권한이 부여됩니다. AWS AppSync의 클라이언트에서 GraphQL 인수는 맵으로 저장됩니다. 이러한 인수는 .entrySet()을 통해 반복할 수 있으며, 이러한 반복을 통해 키와 값 모두에 한 번에 액세스하고, 입력 변환이나 확인 등 복잡한 조건부 검증을 수행하거나 다른 변수를 채울 수 있습니다.

```
#foreach( $entry in $context.arguments.entrySet() )
#if ($entry.key == "XYZ" && $entry.value == "BAD")
    #set($myvar = "...")
  #else
    #break
  #end
#end
```

다른 일반적인 예제에서는 데이터 동기화 시 초기 객체 버전(충돌 해결 시 매우 중요) 또는 권한 부여 확인을 위한 객체의 기본 소유자(이 블록 게시물을 생성한 Mary) 같은 기본 정보가 자동으로 채워지므로, 다음과 같습니다.

```
#set($myMap.owner ="Mary")
```

```
#set($myMap.defaultOwners = ["Admins", "Editors"])
```

컨텍스트

VTL을 사용하여 AWS AppSync 해석기에서 로직 점검을 수행하는 데 익숙해지셨을 테니, 지금부터는 컨텍스트 객체를 살펴보겠습니다.

```
$util.qr($myMap.put("context", $context))
```

여기에는 GraphQL 요청에서 액세스할 수 있는 모든 정보가 들어 있습니다. 자세한 설명은 컨텍스트 참조를 참조하십시오.

필터링

지금까지 이 자습서에서는 매우 단순한 JSON 변환을 사용하여 Lambda 함수의 모든 정보가 GraphQL 쿼리로 반환되었습니다.

```
$util.toJson($context.result)
```

VTL 로직은 데이터 원본에서 응답을 가져올 때, 특히 리소스에 대한 권한 부여 확인을 수행할 때 매우 유용합니다. 몇 가지 예를 살펴보겠습니다. 먼저 다음과 같이 응답 템플릿을 변경해 보겠습니다.

```
#set($data = {
    "id" : "456",
    "meta" : "Valid Response"
})

$util.toJson($data)
```

GraphQL 작업에서 어떤 일이 일어나든, 하드코딩된 값이 클라이언트로 반환됩니다. 이 자습서의 앞부분에서 조건문에 대해 알아볼 때 `meta` 값에서 설정했던 Lambda 응답으로 `elseIfCheck` 필드가 채워지도록 이 코드를 조금 변경합니다.

```
#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.result는 맵이므로 entrySet()을 사용하여 반환된 키나 값에 대해 로직을 수행할 수 있습니다. $context.identity에는 GraphQL 작업을 수행한 사용자에 대한 정보가 포함되어 있으므로, 데이터 원본에서 권한 부여 정보를 반환할 때 로직에 근거하여 사용자에게 데이터 전체 또는 일부를 반환하거나 데이터를 반환하지 않을 수 있습니다. 응답 템플릿을 다음과 같은 형태로 변경합니다.

```
#if($context.result["id"] == 123)
    $util.toJson($context.result)
  #else
    $util.unauthorized()
#end
```

GraphQL 쿼리를 실행할 경우 데이터가 보통 때처럼 반환됩니다. 하지만 ID 인수를 123(query test { get(id:456 meta:"badrequest"){} }) 이외의 것으로 변경할 경우 권한 부여 실패 메시지가 표시됩니다.

권한 부여 사용 사례 단원에서 권한 부여 시나리오의 다른 예를 확인할 수 있습니다.

부록 - 템플릿 샘플

자습서를 따라 진행했다면 단계별로 이 템플릿을 빌드했을 것입니다. 그렇지 않은 경우, 테스트를 위해 복사할 수 있는 템플릿이 아래에 포함되어 있습니다.

요청 템플릿

```
#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)
}
```

응답 템플릿

```
#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
```

해석기 매핑 템플릿 컨텍스트 참조

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync는 해석기 매핑 템플릿 작업을 위한 변수 및 함수 세트를 정의합니다. 이렇게 하면 GraphQL을 사용하는 데이터에 대한 논리적 조작이 더 쉬워집니다. 이 문서에서는 이러한 함수를 설명하고 템플릿을 사용하는 예를 제공합니다.

`$context`에 액세스

`$context` 변수는 해석기 호출에 대한 모든 컨텍스트 정보를 보관하는 맵입니다. 이 변수의 구조는 다음과 같습니다.

```
{
   "arguments" : { ... },
   "source" : { ... },
   "result" : { ... },
   "identity" : { ... },
   "request" : { ... },
```

```
    "info": { ... }
}
```

> **Note**
>
> 값을 검색하기 위해 키로 사전/맵 항목(예: `context`의 항목)에 액세스하려는 경우 Velocity Template Language(VTL)를 통해 `<dictionary-element>.<key-name>` 표기법을 직접 사용할 수 있습니다. 하지만 이는 키 이름이 특수 문자인 경우(예: 밑줄(_))와 같이 모든 경우에서 작동하지 않을 수 있습니다. 항상 `<dictionary-element>.get("<key-name>")` 주석을 사용하는 것이 좋습니다.

`$context` 맵의 각 필드는 다음과 같이 정의됩니다.

`$context` 필드

`arguments`

이 필드에 대한 모든 GraphQL 인수를 포함하는 맵입니다.

`identity`

호출자에 대한 정보를 포함하는 객체입니다. 이 필드의 구조에 대한 자세한 내용은 자격 증명을 참조하십시오.

`source`

상위 필드의 해결 방법을 포함하는 맵입니다.

`stash`

stash는 각 해석기 및 함수 매핑 템플릿 내에서 사용할 수 있는 맵입니다. 단일 해석기가 실행되는 동안에는 동일한 stash 인스턴스가 사용됩니다. 이는 파이프라인 해석기 내에서 stash를 사용해 요청 및 응답 매핑 템플릿 간에 그리고 함수 간에 임의 데이터를 전달할 수 있다는 의미입니다. stash는 동일한 메서드를 Java Map 데이터 구조로 노출합니다.

`result`

해석기의 결과를 포함하고 있는 컨테이너입니다. 이 필드는 매핑 템플릿에 응답하는 경우에만 사용할 수 있습니다.

예를 들어, 다음 쿼리의 `author` 필드를 해석하는 경우:

```
query {
    getPost(id: 1234) {
        postId
        title
        content
        author {
            id
            name
        }
    }
}
```

응답 매핑 템플릿을 처리하는 경우 사용할 수 있는 전체 $context 변수는 다음과 같습니다.

```
{
  "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

파이프라인 해석기에서 실행된 이전 작업의 결과입니다.

이전 작업이 파이프라인 해석기 이전 매핑 템플릿이었으면 $ctx.prev.result는 템플릿의 평가 결과를 나타내고 파이프라인의 첫 번째 함수에 사용할 수 있습니다.

이전 작업이 첫 번째 함수였으면 `$ctx.prev.result`는 첫 번째 함수의 결과를 나타내고, 파이프라인의 두 번째 함수에 사용할 수 있습니다.

이전 작업이 마지막 함수였으면 `$ctx.prev.result`는 마지막 함수의 결과를 나타내고, 파이프라인 해석기 이후 매핑 템플릿에 사용할 수 있습니다.

`info`

GraphQL 요청에 대한 정보가 포함된 객체입니다. 이 필드의 구조는 정보를 참조하십시오.

Identity

`identity` 섹션에는 호출자에 대한 정보가 포함되어 있습니다. 이 섹션의 모양은 AWS AppSync API의 권한 부여 유형에 따라 달라집니다.

AWS AppSync 보안 옵션에 대한 자세한 내용은 권한 부여 및 인증을 참조하세요.

`API_KEY` 권한 부여

`identity` 필드는 채워져 있지 않습니다.

`AWS_LAMBDA` 권한 부여

`identity`에는 요청을 승인하는 Lambda 함수가 반환한 동일한 `resolverContext` 콘텐츠가 포함된 `resolverContext` 키가 포함되어 있습니다.

`AWS_IAM` 권한 부여

`identity`의 형식은 다음과 같습니다.

```
{
    "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
}
```

AMAZON_COGNITO_USER_POOLS 권한 부여

identity의 형식은 다음과 같습니다.

```
{
    "sub" : "uuid",
    "issuer" : "string",
    "username" : "string"
    "claims" : { ... },
    "sourceIp" : ["x.x.x.x"],
    "defaultAuthStrategy" : "string"
}
```

각 필드는 다음과 같이 정의됩니다.

accountId

호출자의 AWS 계정 ID입니다.

claims

사용자의 클레임입니다.

cognitoIdentityAuthType

자격 증명 유형을 기반으로 인증되거나 인증되지 않습니다.

cognitoIdentityAuthProvider

요청에 서명하는 데 사용되는 보안 인증 정보를 얻는 데 사용되는 외부 자격 증명 공급자 정보를 쉼표로 구분한 목록입니다.

cognitoIdentityId

호출자의 Amazon Cognito 자격 증명 ID입니다.

cognitoIdentityPoolId

호출자와 연결된 Amazon Cognito 자격 증명 풀 ID입니다.

defaultAuthStrategy

이 호출자의 기본 권한 부여 전략(ALLOW 또는 DENY)입니다.

issuer

토큰 발행자입니다.

sourceIp

AWS AppSync를 수신하는 호출자의 소스 IP 주소입니다. 요청에 `x-forwarded-for` 헤더가 포함되어 있지 않으면 소스 IP 값에 TCP 연결의 단일 IP 주소만 포함됩니다. 요청에 `x-forwarded-for` 헤더가 포함되어 있으면 소스 IP는 TCP 연결의 IP 주소 이외에 `x-forwarded-for` 헤더의 IP 주소 목록입니다.

sub

인증된 사용자의 UUID입니다.

user

IAM 사용자입니다.

userArn

IAM 사용자의 Amazon 리소스 이름(ARN)입니다.

username

인증된 사용자의 사용자 이름입니다. `AMAZON_COGNITO_USER_POOLS` 권한 부여 시 username의 값은 cognito:username 속성의 값입니다. `AWS_IAM` 권한 부여 시 username의 값은 AWS 사용자 보안 주체의 값입니다. Amazon Cognito 자격 증명 풀에서 판매된 자격 증명으로 IAM 인증을 사용하는 경우 `cognitoIdentityId`를 사용하는 것이 좋습니다.

요청 헤더에 액세스

AWS AppSync에서는 클라이언트에서 사용자 지정 헤더를 전달하고, GraphQL 해석기에서 `$context.request.headers`를 사용하여 이러한 헤더에 액세스하는 것을 지원합니다. 그런 다음 데이터 소스에 데이터 삽입 또는 권한 부여 검사 등과 같은 작업에 헤더 값을 사용할 수 있습니다. 다음 예에 표시된 것처럼 명령줄에서 API 키와 함께 `$curl`을 사용하여 단일 또는 다중 요청 헤더를 사용할 수 있습니다.

단일 헤더의 예

다음과 같이 값이 `nadia`인 `custom`의 헤더를 설정한다고 가정해 보겠습니다.

```
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
```

그러면 $context.request.headers.custom을 사용하여 이 헤더에 액세스할 수 있습니다. 예를 들어, DynamoDB에 대한 다음 VTL에 있을 수 있습니다.

```
"custom": $util.dynamodb.toDynamoDBJson($context.request.headers.custom)
```

다중 헤더의 예

또한 단일 요청에서 여러 헤더를 전달하고 해석기 매핑 템플릿에서 이러한 헤더에 액세스할 수 있습니다. 예를 들어, custom 헤더가 다음 두 값으로 설정되어 있는 경우:

```
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
```

이러한 두 값에는 배열(예: $context.request.headers.custom[1])로 액세스할 수 있습니다.

> **Note**
>
> AWS AppSync는 $context.request.headers의 쿠키 헤더를 노출하지 않습니다.

사용자 지정 도메인 이름 요청 액세스

AWS AppSync는 GraphQL 및 API의 실시간 엔드포인트에 액세스하는 데 사용할 수 있는 사용자 지정 도메인 구성을 지원합니다. 사용자 지정 도메인 이름으로 요청하는 경우 $context.request.domainName을 사용하여 도메인 이름을 가져올 수 있습니다.

기본 GraphQL 엔드포인트 도메인 이름을 사용하는 경우 값은 null입니다.

Info

이 info 섹션에는 GraphQL 요청에 대한 정보가 포함되어 있습니다. 이 섹션은 다음과 같은 형식으로 되어 있습니다.

```
{
    "fieldName": "string",
    "parentTypeName": "string",
    "variables": { ... },
    "selectionSetList": ["string"],
```

```
    "selectionSetGraphQL": "string"
}
```

각 필드는 다음과 같이 정의됩니다.

`fieldName`

현재 확인 중인 필드의 이름입니다.

`parentTypeName`

현재 확인 중인 필드에 대한 상위 유형의 이름입니다.

`variables`

GraphQL 요청에 전달된 모든 변수를 포함하는 맵입니다.

`selectionSetList`

GraphQL 선택 세트에 있는 필드의 목록 표현입니다. 별칭이 있는 필드는 필드 이름이 아닌 별칭 이름으로만 참조됩니다. 다음 예제는 이 구조를 자세히 보여 줍니다.

`selectionSetGraphQL`

GraphQL 스키마 정의 언어(SDL)로 형식이 지정된 선택 세트의 문자열 표현입니다. 조각이 선택 세트에 병합되지는 않지만 다음 예제와 같이 인라인 조각은 유지됩니다.

> **Note**
>
> `context.info`에서 `$utils.toJson()`을 사용하는 경우 `selectionSetGraphQL` 및 `selectionSetList`가 반환하는 값은 기본적으로 직렬화되지 않습니다.

예를 들어, 다음 쿼리의 `getPost` 필드를 해석하는 경우:

```
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
  }
}
```

매핑 템플릿을 처리할 때 사용할 수 있는 전체 $context.info 변수는 다음과 같습니다.

```
{
  "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}"
}
```

selectionSetList는 현재 유형에 속하는 필드만 노출합니다. 현재 유형이 인터페이스 또는 집합인 경우 인터페이스에 속하는 선택된 필드만 노출됩니다. 예를 들어 다음과 같은 스키마가 있습니다.

```
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
}
```

그리고 다음과 같은 쿼리가 있습니다.

```
query {
    node(id: "post1") {
        id
        ... on Post {
            title
        }

        ... on Blog {
            title
        }
    }
```

```
}
```

Query.node 필드 해상도에서 $ctx.info.selectionSetList를 호출하면 id만 노출됩니다.

```
"selectionSetList": [
    "id"
]
```

입력 삭제

애플리케이션은 외부 대상이 원래 목적 외에 애플리케이션을 사용하지 못하도록 신뢰할 수 없는 입력을 삭제해야 합니다. $context에는 $context.arguments, $context.identity, $context.result, $context.info.variables 및 $context.request.headers와 같은 속성의 사용자 입력이 포함되어 있으므로 매핑 템플릿에서 해당 값을 삭제하는 데 주의를 기울여야 합니다.

매핑 템플릿은 JSON을 나타내므로, 입력 삭제는 사용자 입력을 나타내는 문자열에서 JSON 예약 문자를 이스케이프 처리하는 형식을 사용합니다. 매핑 템플릿에 배치할 때 민감한 문자열 값에서 JSON 예약 문자를 이스케이프 처리하는 데 $util.toJson() 유틸리티를 사용하는 것이 좋습니다.

예를 들어 다음의 Lambda 요청 매핑 템플릿에서 안전하지 않은 고객 입력 문자열($context.arguments.id)에 액세스했기 때문에 이스케이프 처리되지 않은 JSON 문자가 JSON 템플릿을 손상시키지 않도록 $util.toJson()으로 이 문자열을 래핑했습니다.

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": $util.toJson($context.arguments.id)
    }
}
```

아래의 매핑 템플릿과는 반대로, 삭제 없이 $context.arguments.id를 직접 삽입합니다. 이러한 작업은 이스케이프 처리되지 않은 따옴표 또는 다른 JSON 예약 문자가 포함되어 있는 문자열에는 사용할 수 없으며, 수행할 경우 템플릿을 열지 못할 수도 있습니다.

```
## 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.
    }
}
```

해석기 매핑 템플릿 유틸리티 참조

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSyncGraphQL 리졸버 내에서 데이터 소스와의 상호 작용을 단순화하는 데 사용할 수 있는 유틸리티 세트를 정의합니다. 이러한 유틸리티 중 일부는 ID 또는 타임스탬프 생성과 같이 모든 데이터 소스에서 일반적으로 사용됩니다. 다른 유틸리티는 데이터 소스 유형에 따라 다릅니다.

주제

- $util의 유틸리티 도우미
- AWS AppSync 디렉티브
- $util.time의 시간 도우미
- $util.list의 목록 도우미
- $util.map의 맵 도우미
- $util.dynamodb의 DynamoDB 헬퍼
- $util.rds의 Amazon RDS 도우미
- $util.http의 HTTP 도우미
- $util.xml의 XML 도우미
- $util.transform의 변환 도우미
- $util.math의 수학 연산 도우미
- $util.str의 문자열 도우미
- 확장 프로그램

$util의 유틸리티 도우미

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

`$util` 변수에는 데이터 작업에 도움이 되는 일반 유틸리티 메서드가 포함되어 있습니다. 달리 지정하지 않는 한, 모든 유틸리티는 UTF-8 문자 집합을 사용합니다.

JSON 구문 분석 유틸리티

JSON 구문 분석 유틸리티 목록

`$util.parseJson(String) : Object`

"문자열화된" JSON을 가져와서 결과의 객체 표현을 반환합니다.

`$util.toJson(Object) : String`

객체를 받아 해당 객체의 "문자열화된" JSON 표현을 반환합니다.

인코딩 유틸리티

인코딩 유틸리티 목록

`$util.urlEncode(String) : String`

입력 문자열을 `application/x-www-form-urlencoded` 인코딩 문자열로 반환합니다.

`$util.urlDecode(String) : String`

`application/x-www-form-urlencoded` 인코딩 문자열을 인코딩되지 않은 형식으로 다시 디코딩합니다.

`$util.base64Encode( byte[] ) : String`

입력을 base64 인코딩 문자열로 인코딩합니다.

`$util.base64Decode(String) : byte[]`

데이터를 base64 인코딩 문자열에서 디코딩합니다.

ID 생성 유틸리티

ID 생성 유틸리티

`$util.autoId() : String`

임의로 생성된 128비트 UUID를 반환합니다.

`$util.autoUlid() : String`

무작위로 생성된 128비트 ULID(Universally Unique Lexicographically Sortable Identifier)를 반환합니다.

`$util.autoKsuid() : String`

길이가 27인 문자열로 인코딩된 무작위로 생성된 128비트 KSUID(K-Sortable Unique Identifier) base62를 반환합니다.

오류 유틸리티

오류 유틸리티 목록

`$util.error(String)`

사용자 지정 오류를 발생시킵니다. 요청 또는 응답 매핑 템플릿에서 사용하여 요청 또는 호출 결과와 관련된 오류를 감지할 수 있습니다.

`$util.error(String, String)`

사용자 지정 오류를 발생시킵니다. 요청 또는 응답 매핑 템플릿에서 사용하여 요청 또는 호출 결과와 관련된 오류를 감지할 수 있습니다. `errorType`도 지정할 수 있습니다.

`$util.error(String, String, Object)`

사용자 지정 오류를 발생시킵니다. 요청 또는 응답 매핑 템플릿에서 사용하여 요청 또는 호출 결과와 관련된 오류를 감지할 수 있습니다. `errorType` 및 `data` 필드를 지정할 수도 있습니다. GraphQL 응답에서 `data` 값은 `error` 내 해당 `errors` 블록에 추가됩니다.

> **Note**
>
> 참고: `data`는 쿼리 선택 세트를 기반으로 필터링됩니다.

`$util.error(String, String, Object, Object)`

사용자 지정 오류를 발생시킵니다. 이 필드는 요청 또는 응답 매핑 템플릿에서 템플릿이 요청 또는 호출 결과와 관련된 오류를 감지하는 경우 사용할 수 있습니다. 또한 `errorType`, `data` 및 `errorInfo` 필드를 지정할 수 있습니다. GraphQL 응답에서 `data` 값은 `error` 내 해당 `errors` 블록에 추가됩니다.

> **Note**
>
> 참고: `data`는 쿼리 선택 세트를 기반으로 필터링됩니다. GraphQL 응답에서 `errorInfo` 값은 `error` 내 해당 `errors` 블록에 추가됩니다.
> `errorInfo`는 쿼리 선택 세트를 기반으로 필터링되지 않습니다.

`$util.appendError(String)`

사용자 지정 오류를 추가합니다. 이 필드는 요청 또는 응답 매핑 템플릿에서 템플릿이 요청 또는 호출 결과와 관련된 오류를 감지하는 경우 사용할 수 있습니다. `$util.error(String)`와 달리 템플릿 평가가 중단되지 않기 때문에 데이터를 호출자에게 반환할 수 있습니다.

`$util.appendError(String, String)`

사용자 지정 오류를 추가합니다. 이 필드는 요청 또는 응답 매핑 템플릿에서 템플릿이 요청 또는 호출 결과와 관련된 오류를 감지하는 경우 사용할 수 있습니다. 또한 `errorType`을 지정할 수 있습니다. `$util.error(String, String)`와 달리 템플릿 평가가 중단되지 않기 때문에 데이터를 호출자에게 반환할 수 있습니다.

`$util.appendError(String, String, Object)`

사용자 지정 오류를 추가합니다. 이 필드는 요청 또는 응답 매핑 템플릿에서 템플릿이 요청 또는 호출 결과와 관련된 오류를 감지하는 경우 사용할 수 있습니다. 또한 `errorType`과 `data` 필드를 지정할 수 있습니다. `$util.error(String, String, Object)`와 달리 템플릿 평가가 중단되지 않기 때문에 데이터를 호출자에게 반환할 수 있습니다. GraphQL 응답에서 `data` 값은 `error` 내 해당 `errors` 블록에 추가됩니다.

> **Note**
>
> 참고: `data`는 쿼리 선택 세트를 기반으로 필터링됩니다.

`$util.appendError(String, String, Object, Object)`

사용자 지정 오류를 추가합니다. 이 필드는 요청 또는 응답 매핑 템플릿에서 템플릿이 요청 또는 호출 결과와 관련된 오류를 감지하는 경우 사용할 수 있습니다. 또한 `errorType`, `data` 및 `errorInfo` 필드를 지정할 수 있습니다. `$util.error(String, String, Object, Object)`와 달리 템플릿 평가가 중단되지 않기 때문에 데이터를 호출자에게 반환할 수 있습니다. GraphQL 응답에서 `data` 값은 `error` 내 해당 `errors` 블록에 추가됩니다.

> **Note**
>
> 참고: `data`는 쿼리 선택 세트를 기반으로 필터링됩니다. GraphQL 응답에서 `errorInfo` 값은 `error` 내 해당 `errors` 블록에 추가됩니다.
> `errorInfo`는 쿼리 선택 세트를 기반으로 필터링되지 않습니다.

조건 검증 유틸리티

조건 검증 유틸리티 목록

`$util.validate(Boolean, String) : void`

조건이 거짓이면 지정된 CustomTemplateException 메시지와 함께 a를 던지십시오.

`$util.validate(Boolean, String, String) : void`

조건이 거짓이면 지정된 메시지 및 오류 유형과 CustomTemplateException 함께 a를 던지십시오.

`$util.validate(Boolean, String, String, Object) : void`

조건이 거짓이면 지정된 메시지 및 오류 유형과 CustomTemplateException 함께 응답에서 반환할 데이터를 포함하는 a를 던지십시오.

Null 동작 유틸리티

Null 동작 유틸리티 목록

`$util.isNull(Object) : Boolean`

제공되는 객체가 null이면 true를 반환합니다.

`$util.isNullOrEmpty(String) : Boolean`

제공되는 데이터가 null이거나 빈 문자열이면 true를 반환합니다. 그렇지 않을 경우 false를 반환합니다.

`$util.isNullOrBlank(String) : Boolean`

제공되는 데이터가 null이거나 빈 문자열이면 true를 반환합니다. 그렇지 않을 경우 false를 반환합니다.

`$util.defaultIfNull(Object, Object) : Object`

첫 번째 객체가 null이 아니면 첫 번째 객체를 반환합니다. 그렇지 않은 경우 두 번째 객체를 "기본 객체"로 반환합니다.

`$util.defaultIfNullOrEmpty(String, String) : String`

첫 번째 문자열이 null이 아니거나 비어 있지 않으면 첫 번째 문자열을 반환합니다. 그렇지 않은 경우 두 번째 문자열을 "기본 문자열"로 반환합니다.

`$util.defaultIfNullOrBlank(String, String) : String`

첫 번째 문자열이 null이 아니거나 공백이 아니면 첫 번째 문자열을 반환합니다. 그렇지 않은 경우 두 번째 문자열을 "기본 문자열"로 반환합니다.

패턴 매칭 유틸리티

유형 및 패턴 매칭 유틸리티 목록

`$util.typeOf(Object) : String`

객체 유형을 설명하는 문자열을 반환합니다. 지원되는 유형 식별은 "Null", "숫자", "문자열", "맵", "목록", "부울"입니다. 유형을 식별할 수 없는 경우 반환 유형은 "객체"입니다.

`$util.matches(String, String) : Boolean`

첫 번째 인수의 지정된 패턴이 두 번째 인수에서 제공되는 데이터와 일치하는 경우 true를 반환합니다. 패턴은 `$util.matches("a*b", "aaaaab")` 등과 같은 정규식이어야 합니다. 이 기능은 Pattern을 기반으로 하며, 여기서 자세한 설명을 참조할 수 있습니다.

`$util.authType() : String`

요청에 사용되는 다중 인증 유형을 설명하는 문자열을 반환하고 'IAM 권한 부여', '사용자 풀 권한 부여', 'Open ID Connect 권한 부여' 또는 'API 키 인증'을 반환합니다.

객체 검증 유틸리티

객체 검증 유틸리티 목록

```
$util.isString(Object) : Boolean
```

객체가 문자열인 경우 true를 반환합니다.

```
$util.isNumber(Object) : Boolean
```

객체가 숫자인 경우 true를 반환합니다.

```
$util.isBoolean(Object) : Boolean
```

객체가 부울인 경우 true를 반환합니다.

```
$util.isList(Object) : Boolean
```

객체가 목록인 경우 true를 반환합니다.

```
$util.isMap(Object) : Boolean
```

객체가 맵인 경우 true를 반환합니다.

CloudWatch 로깅 유틸리티

CloudWatch 로깅 유틸리티 목록

```
$util.log.info(Object) : Void
```

API의 로그 수준에서 요청 수준 및 필드 수준 CloudWatch 로깅이 활성화된 경우 제공된 Object의 문자열 표현을 요청된 로그 스트림에 기록합니다. ALL

```
$util.log.info(String, Object...) : Void
```

API의 로그 수준에서 요청 수준 및 필드 수준 CloudWatch 로깅이 활성화된 경우 제공된 Object의 문자열 표현을 요청된 로그 스트림에 기록합니다. ALL 이 유틸리티는 첫 번째 입력 형식 문자열에서 '{}'로 표시된 모든 변수를 제공된 객체의 문자열 표현으로 순서대로 바꿉니다.

```
$util.log.error(Object) : Void
```

API의 로그 수준 또는 로그 수준에서 필드 수준 CloudWatch 로깅이 활성화된 경우 제공된 객체의 문자열 표현을 요청된 로그 스트림에 기록합니다. ERROR ALL

`$util.log.error(String, Object...) : Void`

API의 로그 수준 ERROR 또는 로그 수준에서 필드 수준 CloudWatch 로깅이 활성화된 경우 제공된 개체의 문자열 표현을 요청된 로그 스트림에 기록합니다. ALL 이 유틸리티는 첫 번째 입력 형식 문자열에서 '{}'로 표시된 모든 변수를 제공된 객체의 문자열 표현으로 순서대로 바꿉니다.

반환 값 동작 유틸리티

반환 값 동작 유틸리티 목록

`$util.qr()` 및 `$util.quiet()`

반환된 값을 제한하면서 VTL 문을 실행합니다. 이는 맵에 항목 추가와 같이 임시 자리 표시자를 사용하지 않고 메서드를 실행하고자 하는 경우 유용합니다. 예:

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

Becomes:

```
#set ($myMap = {})
$util.qr($myMap.put("id", "first value"))
```

`$util.escapeJavaScript(String) : String`

입력 문자열을 JavaScript 이스케이프된 문자열로 반환합니다.

`$util.urlEncode(String) : String`

입력 문자열을 application/x-www-form-urlencoded 인코딩 문자열로 반환합니다.

`$util.urlDecode(String) : String`

application/x-www-form-urlencoded 인코딩 문자열을 인코딩되지 않은 형식으로 다시 디코딩합니다.

`$util.base64Encode( byte[] ) : String`

입력을 base64 인코딩 문자열로 인코딩합니다.

`$util.base64Decode(String) : byte[]`

데이터를 base64 인코딩 문자열에서 디코딩합니다.

$util.parseJson(String) : Object

"문자열화된" JSON을 가져와서 결과의 객체 표현을 반환합니다.

$util.toJson(Object) : String

객체를 받아 해당 객체의 "문자열화된" JSON 표현을 반환합니다.

$util.autoId() : String

임의로 생성된 128비트 UUID를 반환합니다.

$util.autoUlid() : String

무작위로 생성된 128비트 ULID(Universally Unique Lexicographically Sortable Identifier)를 반환합니다.

$util.autoKsuid() : String

길이가 27인 문자열로 인코딩된 무작위로 생성된 128비트 KSUID(K-Sortable Unique Identifier) base62를 반환합니다.

$util.unauthorized()

해석 중인 필드에 대해 Unauthorized를 발생시킵니다. 요청 또는 응답 매핑 템플릿에서 이를 사용하여 호출자가 필드를 확인하도록 허용할지 여부를 결정합니다.

$util.error(String)

사용자 지정 오류를 발생시킵니다. 요청 또는 응답 매핑 템플릿에서 사용하여 요청 또는 호출 결과와 관련된 오류를 감지할 수 있습니다.

$util.error(String, String)

사용자 지정 오류를 발생시킵니다. 요청 또는 응답 매핑 템플릿에서 사용하여 요청 또는 호출 결과와 관련된 오류를 감지할 수 있습니다. errorType도 지정할 수 있습니다.

$util.error(String, String, Object)

사용자 지정 오류를 발생시킵니다. 요청 또는 응답 매핑 템플릿에서 사용하여 요청 또는 호출 결과와 관련된 오류를 감지할 수 있습니다. errorType 및 data 필드를 지정할 수도 있습니다. GraphQL 응답에서 data 값은 error 내 해당 errors 블록에 추가됩니다. 참고: data는 쿼리 선택 세트를 기반으로 필터링됩니다.

$util.error(String, String, Object, Object)

사용자 지정 오류를 발생시킵니다. 이 필드는 요청 또는 응답 매핑 템플릿에서 템플릿이 요청 또는 호출 결과와 관련된 오류를 감지하는 경우 사용할 수 있습니다. 또한 errorType, data

및 errorInfo 필드를 지정할 수 있습니다. GraphQL 응답에서 data 값은 error 내 해당 errors 블록에 추가됩니다. 참고: data는 쿼리 선택 세트를 기반으로 필터링됩니다. GraphQL 응답에서 errorInfo 값은 error 내 해당 errors 블록에 추가됩니다. 참고: errorInfo는 쿼리 선택 세트를 기반으로 필터링되지 않습니다.

$util.appendError(String)

사용자 지정 오류를 추가합니다. 이 필드는 요청 또는 응답 매핑 템플릿에서 템플릿이 요청 또는 호출 결과와 관련된 오류를 감지하는 경우 사용할 수 있습니다. $util.error(String)와 달리 템플릿 평가가 중단되지 않기 때문에 데이터를 호출자에게 반환할 수 있습니다.

$util.appendError(String, String)

사용자 지정 오류를 추가합니다. 이 필드는 요청 또는 응답 매핑 템플릿에서 템플릿이 요청 또는 호출 결과와 관련된 오류를 감지하는 경우 사용할 수 있습니다. 또한 errorType을 지정할 수 있습니다. $util.error(String, String)와 달리 템플릿 평가가 중단되지 않기 때문에 데이터를 호출자에게 반환할 수 있습니다.

$util.appendError(String, String, Object)

사용자 지정 오류를 추가합니다. 이 필드는 요청 또는 응답 매핑 템플릿에서 템플릿이 요청 또는 호출 결과와 관련된 오류를 감지하는 경우 사용할 수 있습니다. 또한 errorType과 data 필드를 지정할 수 있습니다. $util.error(String, String, Object)와 달리 템플릿 평가가 중단되지 않기 때문에 데이터를 호출자에게 반환할 수 있습니다. GraphQL 응답에서 data 값은 error 내 해당 errors 블록에 추가됩니다. 참고: data는 쿼리 선택 세트를 기반으로 필터링됩니다.

$util.appendError(String, String, Object, Object)

사용자 지정 오류를 추가합니다. 이 필드는 요청 또는 응답 매핑 템플릿에서 템플릿이 요청 또는 호출 결과와 관련된 오류를 감지하는 경우 사용할 수 있습니다. 또한 errorType, data 및 errorInfo 필드를 지정할 수 있습니다. $util.error(String, String, Object, Object)와 달리 템플릿 평가가 중단되지 않기 때문에 데이터를 호출자에게 반환할 수 있습니다. GraphQL 응답에서 data 값은 error 내 해당 errors 블록에 추가됩니다. 참고: data는 쿼리 선택 세트를 기반으로 필터링됩니다. GraphQL 응답에서 errorInfo 값은 error 내 해당 errors 블록에 추가됩니다. 참고: errorInfo는 쿼리 선택 세트를 기반으로 필터링되지 않습니다.

$util.validate(Boolean, String) : void

조건이 거짓이면 지정된 메시지와 CustomTemplateException 함께 a를 던지십시오.

$util.validate(Boolean, String, String) : void

조건이 거짓이면 지정된 메시지 및 오류 유형과 CustomTemplateException 함께 a를 던지십시오.

$util.validate(Boolean, String, String, Object) : void

조건이 거짓이면 지정된 메시지 및 오류 유형과 CustomTemplateException 함께 응답에서 반환할 데이터를 포함하는 a를 던지십시오.

$util.isNull(Object) : Boolean

제공되는 객체가 null이면 true를 반환합니다.

$util.isNullOrEmpty(String) : Boolean

제공되는 데이터가 null이거나 빈 문자열이면 true를 반환합니다. 그렇지 않을 경우 false를 반환합니다.

$util.isNullOrBlank(String) : Boolean

제공되는 데이터가 null이거나 빈 문자열이면 true를 반환합니다. 그렇지 않을 경우 false를 반환합니다.

$util.defaultIfNull(Object, Object) : Object

첫 번째 객체가 null이 아니면 첫 번째 객체를 반환합니다. 그렇지 않은 경우 두 번째 객체를 "기본 객체"로 반환합니다.

$util.defaultIfNullOrEmpty(String, String) : String

첫 번째 문자열이 null이 아니거나 비어 있지 않으면 첫 번째 문자열을 반환합니다. 그렇지 않은 경우 두 번째 문자열을 "기본 문자열"로 반환합니다.

$util.defaultIfNullOrBlank(String, String) : String

첫 번째 문자열이 null이 아니거나 공백이 아니면 첫 번째 문자열을 반환합니다. 그렇지 않은 경우 두 번째 문자열을 "기본 문자열"로 반환합니다.

$util.isString(Object) : Boolean

객체가 문자열인 경우 true를 반환합니다.

$util.isNumber(Object) : Boolean

객체가 숫자인 경우 true를 반환합니다.

$util.isBoolean(Object) : Boolean

객체가 부울인 경우 true를 반환합니다.

$util.isList(Object) : Boolean

객체가 목록인 경우 true를 반환합니다.

$util.isMap(Object) : Boolean

객체가 맵인 경우 true를 반환합니다.

$util.typeOf(Object) : String

객체 유형을 설명하는 문자열을 반환합니다. 지원되는 유형 식별은 "Null", "숫자", "문자열", "맵", "목록", "부울"입니다. 유형을 식별할 수 없는 경우 반환 유형은 "객체"입니다.

$util.matches(String, String) : Boolean

첫 번째 인수의 지정된 패턴이 두 번째 인수에서 제공되는 데이터와 일치하는 경우 true를 반환합니다. 패턴은 `$util.matches("a*b", "aaaaab")` 등과 같은 정규식이어야 합니다. 이 기능은 Pattern을 기반으로 하며, 여기서 자세한 설명을 참조할 수 있습니다.

$util.authType() : String

요청에 사용되는 다중 인증 유형을 설명하는 문자열을 반환하고 'IAM 권한 부여', '사용자 풀 권한 부여', 'Open ID Connect 권한 부여' 또는 'API 키 인증'을 반환합니다.

$util.log.info(Object) : Void

API의 로그 수준에서 요청 수준 및 필드 수준 CloudWatch 로깅이 활성화된 경우 제공된 Object의 문자열 표현을 요청된 로그 스트림에 기록합니다. `ALL`

$util.log.info(String, Object...) : Void

API의 로그 수준에서 요청 수준 및 필드 수준 CloudWatch 로깅이 활성화된 경우 제공된 Object의 문자열 표현을 요청된 로그 스트림에 기록합니다. `ALL` 이 유틸리티는 첫 번째 입력 형식 문자열에서 '{}'로 표시된 모든 변수를 제공된 객체의 문자열 표현으로 순서대로 바꿉니다.

$util.log.error(Object) : Void

API의 로그 수준 또는 로그 수준에서 필드 수준 CloudWatch 로깅이 활성화된 경우 제공된 객체의 문자열 표현을 요청된 로그 스트림에 기록합니다. `ERROR ALL`

$util.log.error(String, Object...) : Void

API의 로그 수준 `ERROR` 또는 로그 수준에서 필드 수준 CloudWatch 로깅이 활성화된 경우 제공된 개체의 문자열 표현을 요청된 로그 스트림에 기록합니다. `ALL` 이 유틸리티는 첫 번째 입력 형식 문자열에서 '{}'로 표시된 모든 변수를 제공된 객체의 문자열 표현으로 순서대로 바꿉니다.

`$util.escapeJavaScript(String) : String`

입력 문자열을 JavaScript 이스케이프된 문자열로 반환합니다.

해석기 권한 부여

해석기 권한 부여 목록

`$util.unauthorized()`

해석 중인 필드에 대해 `Unauthorized`를 발생시킵니다. 요청 또는 응답 매핑 템플릿에서 이를 사용하여 호출자가 필드를 확인하도록 허용할지 여부를 결정합니다.

AWS AppSync 디렉티브

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync VTL로 작성할 때 개발자 생산성을 높이는 지침을 공개합니다.

지시문 유틸리티

`#return(Object)`

`#return(Object)`를 사용하면 모든 매핑 템플릿에서 조기에 돌아올 수 있습니다. `#return(Object)`는 가장 가까운 범위의 로직 블록에서 반환된다는 점에서 프로그래밍 언어의 return 키워드와 유사합니다. 해석기 매핑 템플릿 내에서 `#return(Object)`를 사용하면 해석기에서 반환함을 의미합니다. 또는 함수 매핑 템플릿의 `#return(Object)`를 사용하면 함수에서 반환하고, 파이프라인의 다음 함수 또는 해석기 응답 매핑 템플릿으로 실행합니다.

`#return`

`#return` 지시문은 `#return(Object)`와 동일한 동작을 나타내지만 `null`이 대신 반환됩니다.

$util.time의 시간 도우미

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

`$util.time` 변수에는 타임스탬프를 생성하고, 날짜/시간 형식 간에 변환하고, 날짜/시간 문자열을 구문 분석하는 데 도움이 되는 날짜/시간 메서드가 포함되어 있습니다. 날짜/시간 형식의 구문은 추가 설명서에서 참조할 수 DateTimeFormatter있는 구문을 기반으로 합니다. 아래에서는 몇 가지 예와 사용 가능한 메서드 목록 및 설명을 제공합니다.

시간 유틸리티

시간 유틸리티 목록

`$util.time.nowISO8601() : String`

UTC의 문자열 표현을 ISO8601 형식으로 반환합니다.

`$util.time.nowEpochSeconds() : long`

1970-01-01T00:00:00Z의 epoch부터 지금까지의 시간을 초로 반환합니다.

`$util.time.nowEpochMilliSeconds() : long`

1970-01-01T00:00:00Z의 epoch부터 지금까지의 시간을 밀리초로 반환합니다.

`$util.time.nowFormatted(String) : String`

문자열 입력 유형의 지정된 형식을 사용하여 현재 타임스탬프의 문자열을 UTC로 반환합니다.

`$util.time.nowFormatted(String, String) : String`

문자열 입력 유형의 지정된 형식 및 시간대를 사용하여 시간대의 현재 타임스탬프 문자열을 반환합니다.

`$util.time.parseFormattedToEpochMilliSeconds(String, String) : Long`

형식과 함께 문자열로 전달된 타임스탬프를 구문 분석하고 epoch 이후 타임스탬프를 밀리초로 반환합니다.

`$util.time.parseFormattedToEpochMilliSeconds(String, String, String) : Long`

형식 및 시간대와 함께 문자열로 전달된 타임스탬프를 구문 분석하고 epoch 이후 타임스탬프를 밀리초로 반환합니다.

`$util.time.parseISO8601ToEpochMilliSeconds(String) : Long`

문자열로 전달된 ISO8601 타임스탬프를 구문 분석하고 epoch 이후 타임스탬프를 밀리초로 반환합니다.

`$util.time.epochMilliSecondsToSeconds(long) : long`

epoch 밀리초 타임스탬프를 epoch 초 타임스탬프로 변환합니다.

`$util.time.epochMilliSecondsToISO8601(long) : String`

epoch 밀리초 타임스탬프를 ISO8601 타임스탬프로 변환합니다.

`$util.time.epochMilliSecondsToFormatted(long, String) : String`

long으로 전달된 epoch 밀리초 타임스탬프를 UTC의 제공된 형식에 따라 형식이 지정된 타임스탬프로 변환합니다.

`$util.time.epochMilliSecondsToFormatted(long, String, String) : String`

long으로 전달된 epoch 밀리초 타임스탬프를 제공된 시간대의 제공된 형식에 따라 형식이 지정된 타임스탬프로 변환합니다.

독립 실행형 함수 예제

```
$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
```

변환 예제

```
#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
```

구문 분석 예제

```
$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
```

AWS AppSync 정의된 스칼라와 함께 사용

다음 형식은 `AWSDate`, `AWSDateTime` 및 `AWSTime`과 호환됩니다.

```
$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
```

$util.list의 목록 도우미

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

`$util.list`에는 사용 사례 필터링을 위해 목록에서 항목을 제거하거나 유지하는 등과 같은 일반적인 목록 작업에 유용한 메서드가 포함되어 있습니다.

목록 유틸리티

`$util.list.copyAndRetainAll(List, List) : List`

첫 번째 인수에 제공된 목록의 단순 복사본을 생성하고 두 번째 인수에 지정된 항목(있는 경우)만 유지합니다. 기타 모든 항목은 이 복사본에서 제거됩니다.

`$util.list.copyAndRemoveAll(List, List) : List`

첫 번째 인수에 제공된 목록의 단순 복사본을 생성하고 두 번째 인수에 지정된 항목(있는 경우)만 제거합니다. 기타 모든 항목은 이 복사본에 보유됩니다.

`$util.list.sortList(List, Boolean, String) : List`

첫 번째 인수에 제공된 객체 목록을 정렬합니다. 두 번째 인수가 true인 경우 목록은 내림차순으로 정렬되고, 두 번째 인수가 false인 경우 목록은 오름차순으로 정렬됩니다. 세 번째 인수는 사용자 지정 객체 목록을 정렬하는 데 사용되는 속성의 문자열 이름입니다. 문자열, 정수, 부동 소수점 또는 배수인 경우 세 번째 인수는 임의의 문자열일 수 있습니다. 모든 객체가 같은 클래스에 속하지 않는 경우 원래 목록이 반환됩니다. 최대 1000개의 객체를 포함하는 목록만 지원됩니다. 다음은 이 유틸리티 사용 예제입니다.

```
 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}]
```

$util.map의 맵 도우미

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

`$util.map`에는 사용 사례 필터링을 위해 맵에서 항목을 제거하거나 유지하는 등과 같은 일반적인 맵 작업에 유용한 메서드가 포함되어 있습니다.

맵 유틸리티

`$util.map.copyAndRetainAllKeys(Map, List) : Map`

첫 번째 맵의 단순 복사본을 생성하고 목록에 지정된 키(있는 경우)만 유지합니다. 기타 모든 키는 이 복사본에서 제거됩니다.

`$util.map.copyAndRemoveAllKeys(Map, List) : Map`

첫 번째 맵의 단순 복사본을 생성하고 목록에서 키가 지정된 항목(있는 경우)을 제거합니다. 기타 모든 키는 이 복사본에 보유됩니다.

$util.dynamodb의 DynamoDB 헬퍼

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

`$util.dynamodb`에는 Amazon DynamoDB에 데이터 쓰기 및 읽기를 더 용이하게 하는 헬퍼 메서드가 포함되어 있습니다(예: 자동 유형 매핑 및 형식 지정). 이러한 메서드는 기본 유형 및 목록을 적절한 DynamoDB 입력 형식(`{ "TYPE" : VALUE }` 형식의 Map)에 자동으로 매핑하도록 설계되어 있습니다.

예를 들어, 이전에는 DynamoDB에서 새 항목을 생성하는 요청 매핑 템플릿은 다음과 같았습니다.

```
{
    "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) }
    }
}
```

객체에 필드를 추가하려는 경우 스키마에서 GraphQL 쿼리를 업데이트하고 요청 매핑 템플릿도 업데이트해야 했습니다. 그러나 이제는 스키마에 추가되는 새 필드를 자동으로 선택해 올바른 형식으로 DynamoDB에 추가하도록 요청 매핑 템플릿을 재구성할 수 있습니다.

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

이전 예에서는 `$util.dynamodb.toDynamoDBJson(...)` 헬퍼를 사용하여 생성된 ID를 자동으로 받아 문자열 속성의 DynamoDB 표현으로 변환했습니다. 그런 다음 모든 인수를 받아 DynamoDB 표현으로 변환하고 템플릿의 `attributeValues` 필드에 출력했습니다.

각 헬퍼에는 객체(예: `$util.dynamodb.toString(...)`)를 반환하는 버전과 객체를 JSON 문자열(예: `$util.dynamodb.toStringJson(...)`)로 반환하는 버전 두 가지가 있습니다. 이전 예에서는 데이터를 JSON 문자열로 반환하는 버전을 사용했습니다. 템플릿에 사용하기 전에 객체를 조작하려는 경우 다음과 같이 대신 객체를 반환하도록 선택할 수 있습니다.

```
{
    "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)
}
```

이전 예에서는 최종적으로 `version`을 사용하여 템플릿의 `timestamp` 필드에 출력하기 전에 JSON 문자열 대신 맵으로 변환된 인수를 반환한 다음 `attributeValues` 및 `$util.toJson(...)` 필드에 추가합니다.

각 헬퍼의 JSON 버전은 `$util.toJson(...)`에서 JSON 이외 버전을 래핑하는 것과 동일합니다. 예를 들어 다음 문은 정확하게 동일합니다.

```
$util.toStringJson("Hello, World!")
$util.toJson($util.toString("Hello, World!"))
```

toDynamoDB

toDynamoDB 유틸리티 목록

`$util.dynamodb.toDynamoDB(Object) : Map`

입력 객체를 적절한 DynamoDB 표현으로 변환하는 DynamoDB용 일반 객체 변환 도구입니다. 이 도구는 몇 가지 형식을 표현하는 방법에 대해 독자적인 방식을 가지고 있습니다. 예를 들어 집합("SS", "NS", "BS")보다는 목록("L")을 사용합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

문자열 예제

```
Input:      $util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

숫자 예제

```
Input:      $util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

부울 예제

```
Input:      $util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

록 예제

```
Input:      $util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
```

```
                {
                    "M" : {
                        "bar" : { "S" : "baz" }
                    }
                }
            ]
        }
```

맵 예제

```
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`

`$util.dynamodb.toDynamoDB(Object) :` Map과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

toString 유틸리티

toString 유틸리티 목록

`$util.dynamodb.toString(String) : String`

입력 문자열을 DynamoDB 문자열 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      $util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

`$util.dynamodb.toStringJson(String) : Map`

$util.dynamodb.toString(String) : String과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

`$util.dynamodb.toStringSet(List<String>) : Map`

문자열이 포함된 목록을 DynamoDB 문자열 집합 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      $util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

`$util.dynamodb.toStringSetJson(List<String>) : String`

$util.dynamodb.toStringSet(List<String>) : Map과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

toNumber 유틸리티

toNumber 유틸리티 목록

`$util.dynamodb.toNumber(Number) : Map`

숫자를 DynamoDB 숫자 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      $util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

`$util.dynamodb.toNumberJson(Number) : String`

$util.dynamodb.toNumber(Number) : Map과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

`$util.dynamodb.toNumberSet(List<Number>) : Map`

숫자 목록을 DynamoDB 숫자 집합 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      $util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
```

```
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

`$util.dynamodb.toNumberSetJson(List<Number>) : String`

`$util.dynamodb.toNumberSet(List<Number>) : Map`과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

toBinary 유틸리티

toBinary 유틸리티 목록

`$util.dynamodb.toBinary(String) : Map`

base64 문자열로 인코딩된 이진 데이터를 DynamoDB 이진 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      $util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

`$util.dynamodb.toBinaryJson(String) : String`

`$util.dynamodb.toBinary(String) : Map`과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

`$util.dynamodb.toBinarySet(List<String>) : Map`

base64 문자열로 인코딩된 이진 데이터 목록을 DynamoDB 이진수 집합 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      $util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

`$util.dynamodb.toBinarySetJson(List<String>) : String`

`$util.dynamodb.toBinarySet(List<String>) : Map`과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

toBoolean 유틸리티

toBoolean 유틸리티 목록

`$util.dynamodb.toBoolean(Boolean) : Map`

부울을 적절한 DynamoDB 부울 형식으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      $util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

`$util.dynamodb.toBooleanJson(Boolean) : String`

`$util.dynamodb.toBoolean(Boolean) : Map`과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

toNull 유틸리티

toNull 유틸리티 목록

`$util.dynamodb.toNull() : Map`

null을 DynamoDB null 형식으로 반환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      $util.dynamodb.toNull()
Output:     { "NULL" : null }
```

`$util.dynamodb.toNullJson() : String`

`$util.dynamodb.toNull() : Map`과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

toList 유틸리티

toList 유틸리티 목록

`$util.dynamodb.toList(List) : Map`

객체 목록을 DynamoDB 목록 형식으로 변환합니다. 목록의 각 항목 역시 적절한 DynamoDB 형식으로 변환됩니다. 이 도구는 몇 가지 중첩 객체를 표현하는 방법에 대해 독자적인 방식을 가지고 있

습니다. 예를 들어 집합("SS", "NS", "BS")보다는 목록("L")을 사용합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
Input:      $util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

`$util.dynamodb.toListJson(List) : String`

`$util.dynamodb.toList(List) : Map`과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

toMap 유틸리티

toMap 유틸리티 목록

`$util.dynamodb.toMap(Map) : Map`

맵을 DynamoDB 맵 형식으로 변환합니다. 맵의 각 값 역시 적절한 DynamoDB 형식으로 변환됩니다. 이 도구는 몇 가지 중첩 객체를 표현하는 방법에 대해 독자적인 방식을 가지고 있습니다. 예를 들어 집합("SS", "NS", "BS")보다는 목록("L")을 사용합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
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`

`$util.dynamodb.toMap(Map) :` Map과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

`$util.dynamodb.toMapValues(Map) : Map`

각 값이 적절한 DynamoDB 형식으로 변환된 탭의 사본을 생성합니다. 이 도구는 몇 가지 중첩 객체를 표현하는 방법에 대해 독자적인 방식을 가지고 있습니다. 예를 들어 집합("SS", "NS", "BS")보다는 목록("L")을 사용합니다.

```
Input:      $util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "foo"  : { "S" : "bar" },
               "baz"  : { "N" : 1234 },
               "beep" : {
                   "L" : [
                       { "S" : "boop" }
                   ]
               }
           }
```

> **Note**
>
> 이는 전체 속성 값 자체가 아니라 DynamoDB 속성 값의 내용만 반환하기 때문에 `$util.dynamodb.toMap(Map) :` Map과 약간 다릅니다. 예를 들어 다음 문은 정확하게 동일합니다.
>
> ```
> $util.dynamodb.toMapValues($map)
> $util.dynamodb.toMap($map).get("M")
> ```

`$util.dynamodb.toMapValuesJson(Map) : String`

`$util.dynamodb.toMapValues(Map) :` Map과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

S3Object 유틸리티

S3Object 유틸리티 목록

`$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`

키, 버킷 및 리전을 DynamoDB S3 객체 표현으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
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`

`$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

`$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map`

키, 버킷, 리전 및 선택적 버전을 DynamoDB S3 객체 표현으로 변환합니다. DynamoDB 속성 값을 설명하는 객체를 반환합니다.

```
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`

`$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map`과 동일하지만 DynamoDB 속성 값을 JSON 인코딩 문자열로 반환합니다.

`$util.dynamodb.fromS3ObjectJson(String) : Map`

DynamoDB S3 객체의 문자열 값을 수락하고 키, 버킷, 리전 및 선택적 버전을 포함하는 맵을 반환합니다.

```
Input:      $util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
 \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
```

```
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :
 "beep" }
```

$util.rds의 Amazon RDS 도우미

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

`$util.rds`에는 결과 출력에서 불필요한 데이터를 제거하여 Amazon RDS 작업의 형식을 지정하는 도우미 메서드가 포함되어 있습니다.

$util.rds 유틸리티 목록

`$util.rds.toJsonString(String serializedSQLResult): String`

문자열화된 원시 Amazon Relational Database Service(RDS) 데이터 API 작업 결과 형식을 보다 간결한 문자열로 변환하여 `String`을 반환합니다. 반환 문자열은 결과 집합의 SQL 레코드의 직렬화된 목록입니다. 모든 레코드는 키-값 페어 모음으로 표시됩니다. 키는 열 이름에 해당합니다.

입력의 해당 명령문이 변형을 수행하는 SQL 쿼리(예: INSERT, UPDATE, DELETE)인 경우 빈 목록이 반환됩니다. 예를 들어, 쿼리 `select * from Books limit 2`는 Amazon RDS 데이터 작업의 원시 결과를 제공합니다.

```
{
    "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.toJsonString은 다음과 같습니다.

```
[
  {
    "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

이는 util.rds.toJsonString과 같지만 결과는 JSON Object입니다.

$util.http의 HTTP 도우미

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

이 `$util.http` 유틸리티는 HTTP 요청 파라미터를 관리하고 응답 헤더를 추가하는 데 사용할 수 있는 도우미 메서드를 제공합니다.

$util.http 유틸리티 목록

`$util.http.copyHeaders(Map) : Map`

제한된 HTTP 헤더 세트 없이 맵에서 헤더를 복사합니다. 이를 사용하여 요청 헤더를 다운스트림 HTTP 엔드포인트로 전달할 수 있습니다.

```
{
    ...
    "params": {
        ...
        "headers": $util.http.copyHeaders($ctx.request.headers),
        ...
    },
    ...
}
```

`$util.http.addResponseHeader(String, Object)`

응답의 이름(`String`) 및 값(`Object`)이 포함된 단일 사용자 지정 헤더를 추가합니다. 다음과 같은 제한이 적용됩니다.

- 헤더 이름은 기존 또는 제한된 AWS 또는 AWS AppSync 헤더와 일치할 수 없습니다.
- 헤더 이름은 `x-amzn-` 또는 `x-amz-` 같은 제한된 접두사로 시작할 수 없습니다.
- 사용자 지정 응답 헤더의 크기는 4KB를 초과할 수 없습니다. 여기에는 헤더 이름과 값이 포함됩니다.
- GraphQL 작업당 각 응답 헤더를 한 번씩 정의해야 합니다. 하지만 이름이 같은 사용자 지정 헤더를 여러 번 정의하면 응답에 가장 최근의 정의가 나타납니다. 이름 지정과 상관없이 모든 헤더는 헤더 크기 제한에 포함됩니다.

```
...
$util.http.addResponseHeader("itemsCount", 7)
$util.http.addResponseHeader("render", $ctx.args.render)
...
```

`$util.http.addResponseHeaders(Map)`

지정된 이름 맵(String) 및 값(Object) 의 응답에 여러 응답 헤더를 추가합니다. `addResponseHeader(String, Object)` 메서드에 나열된 것과 동일한 제한 사항이 이 메서드에도 적용됩니다.

```
...
#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)
...
```

$util.xml의 XML 도우미

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

`$util.xml`에는 XML 응답을 JSON 또는 사전으로 쉽게 변환할 수 있도록 하는 헬퍼 메서드가 포함되어 있습니다.

$util.xml 유틸리티 목록

`$util.xml.toMap(String) : Map`

XML 문자열을 사전으로 변환합니다.

```
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`

XML 문자열을 JSON 문자열로 변환합니다. 출력이 문자열인 것을 제외하면 toMap과 비슷합니다. 이는 HTTP 객체에서 JSON으로 XML 응답을 직접 변환하고 반환하고자 하는 경우 유용합니다.

`$util.xml.toJsonString(String, Boolean) : String`

XML 문자열을 선택 사항인 부울 파라미터가 포함된 JSON 문자열로 변환하여 JSON으로 인코딩하고자 하는지 판단합니다.

$util.transform의 변환 도우미

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

`$util.transform`에는 Amazon DynamoDB 필터 작업과 같은 데이터 소스에 대해 복잡한 작업을 더 쉽게 수행할 수 있게 해주는 도우미 메서드가 포함되어 있습니다.

변환 도우미

변환 도우미 유틸리티 목록

`$util.transform.toDynamoDBFilterExpression(Map) : Map`

입력 문자열을 DynamoDB에 사용할 필터 표현식으로 변환합니다.

```
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

주어진 입력을 동등한 OpenSearch Query DSL 표현식으로 변환하여 JSON 문자열로 반환합니다.

```
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"
                    }
                  }
              ]
            }
          }
      ]
    }
}
```

기본 연산자는 AND로 가정합니다.

변환 도우미 구독 필터

변환 도우미 구독 필터 유틸리티 목록

`$util.transform.toSubscriptionFilter(Map) : Map`

Map 입력 객체를 SubscriptionFilter 표현식 객체로 변환합니다. `$util.transform.toSubscriptionFilter` 메서드는 `$extensions.setSubscriptionFilter()` 확장에 대한 입력으로 사용됩니다. 자세한 내용은 확장을 참조하세요.

`$util.transform.toSubscriptionFilter(Map, List) : Map`

Map 입력 객체를 SubscriptionFilter 표현식 객체로 변환합니다. `$util.transform.toSubscriptionFilter` 메서드는 `$extensions.setSubscriptionFilter()` 확장에 대한 입력으로 사용됩니다. 자세한 내용은 확장을 참조하세요.

첫 번째 인수는 SubscriptionFilter 표현식 객체로 변환되는 Map 입력 객체입니다. 두 번째 인수는 SubscriptionFilter 표현식 객체를 구성하는 동안 첫 번째 Map 입력 객체에서 무시되는 필드 이름의 List입니다.

`$util.transform.toSubscriptionFilter(Map, List, Map) : Map`

Map 입력 객체를 SubscriptionFilter 표현식 객체로 변환합니다. `$util.transform.toSubscriptionFilter` 메서드는 `$extensions.setSubscriptionFilter()` 확장에 대한 입력으로 사용됩니다. 자세한 내용은 확장을 참조하세요.

첫 번째 인수는 SubscriptionFilter 표현식 객체로 변환되는 Map 입력 객체, 두 번째 인수는 첫 번째 Map 입력 객체에서 무시될 필드 이름의 List, 세 번째 인수는 SubscriptionFilter 표현식 객체를 구성하는 동안 포함되는 엄격한 규칙의 Map 입력 객체입니다. 이러한 엄격한 규칙이 SubscriptionFilter 표현식 객체에 포함되므로 최소한 하나의 규칙이 충족되어 구독 필터를 통과할 수 있습니다.

구독 필터 인수

다음 표에서는 다음 유틸리티의 인수를 정의하는 방법을 설명합니다.

- `$util.transform.toSubscriptionFilter(Map) : Map`
- `$util.transform.toSubscriptionFilter(Map, List) : Map`

- `$util.transform.toSubscriptionFilter(Map, List, Map) : Map`

Argument 1: Map

인수 1은 다음과 같은 키 값을 가진 Map 객체입니다.

- 필드 이름
- "and"
- "or"

필드 이름을 키로 사용하는 경우 해당 필드 항목의 조건은 "operator" : "value" 형식입니다.

다음 예에서는 Map에 항목을 추가하는 방법을 보여줍니다.

```
"field_name" : {
                    "operator1" : value
                }

## We can have multiple conditions for the same field_name:

"field_name" : {
                    "operator1" : value
                    "operator2" : value
                    .
                    .
                    .
                }
```

필드에 두 개 이상의 조건이 있는 경우 이러한 모든 조건은 OR 연산을 사용하는 것으로 간주됩니다.

입력 Map에 "and"와 "or"를 키로 사용할 수도 있습니다. 즉, 이들 항목 내의 모든 항목은 키에 따라 AND 또는 OR 논리를 사용하여 결합되어야 합니다. 키 값 "and"와 "or"에는 일련의 조건이 필요합니다.

```
"and" : [

            {
                "field_name1" : {
                    "operator1" : value
```

```
            }
        },

        {
            "field_name2" : {
                "operator1" : value
            }
        },
        .
        .
    ].
```

참고로 "and"와 "or"는 중첩되지 않습니다. 즉, 다른 "and"/"or" 블록 내에 "and"/"or"를 중첩할 수 있습니다. 하지만 단순한 필드에서는 작동하지 않습니다.

```
"and" : [

        {
            "field_name1" : {
                "operator" : value
            }
        },

        {
            "or" : [
                        {
                            "field_name2" : {
                                "operator" : value
                            }
                        },

                        {
                            "field_name3" : {
                                "operator" : value
                            }
                        }

                    ].
```

다음 예제에서는 `$util.transform.toSubscriptionFilter(Map) : Map`를 사용하여 인수 1을 입력하는 방법을 보여줍니다.

입력

인수 1: 맵:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

출력

결과는 Map 객체입니다.

```
{
  "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

인수 2에는 SubscriptionFilter 표현식 객체를 생성하는 동안 입력 Map(인수 1)에서 고려하면 안 되는 필드 이름의 List가 포함되어 있습니다. List도 비어 있을 수 있습니다.

다음 예제에서는 $util.transform.toSubscriptionFilter(Map, List) : Map을 사용하여 인수 1 및 인수 2를 입력하는 방법을 보여줍니다.

입력

인수 1: 맵:

```
{

  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
```

```
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

인수 2: 목록:

```
["percentageUp", "author"]
```

출력

결과는 Map 객체입니다.

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 3: Map

인수 3은 필드 이름을 키 값으로 갖는 Map 객체입니다("and" 또는 "or"를 가질 수 없음). 필드 이름을 키로 사용하는 경우 해당 필드의 조건은 "operator" : "value" 형식의 항목입니다. 인수 1과

달리 인수 3은 동일한 키에 여러 조건을 포함할 수 없습니다. 또한 인수 3에는 "and" 또는 "or" 절이 없으므로 중첩도 필요하지 않습니다.

인수 3은 엄격한 규칙 목록을 나타내며, 이러한 규칙 목록은 필터를 통과하기 위해 이러한 조건 중 하나 이상이 충족되도록 `SubscriptionFilter` 표현식 객체에 추가됩니다.

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.
```

다음 예제에서는 `$util.transform.toSubscriptionFilter(Map, List, Map) : Map`을 사용하여 인수 1, 인수 2 및 인수 3을 입력하는 방법을 보여줍니다.

입력

인수 1: 맵:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
```

```
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

인수 2: 목록:

```
["percentageUp", "author"]
```

인수 3: 맵:

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
}
```

출력

결과는 Map 객체입니다.

```
{
  "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"
        }
      ]
    }
  ]
}
```

$util.math의 수학 연산 도우미

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

`$util.math`에는 일반적인 수학 연산 작업에 도움이 되는 메서드가 있습니다.

$util.math 유틸리티 목록

`$util.math.roundNum(Double) : Integer`

배수를 가져와서 가장 가까운 정수로 반올림합니다.

`$util.math.minVal(Double, Double) : Double`

두 개의 배수를 가져와서 두 배수의 최소값을 반환합니다.

`$util.math.maxVal(Double, Double) : Double`

두 개의 배수를 가져와서 두 배수의 최대값을 반환합니다.

`$util.math.randomDouble() : Double`

0과 1 사이의 임의의 배수를 반환합니다.

> **Important**
>
> 이 함수는 높은 엔트로피 무작위화가 필요한 경우(예: 암호화)에는 사용해서는 안 됩니다.

`$util.math.randomWithinRange(Integer, Integer) : Integer`

지정된 범위 내의 임의의 정수 값을 반환합니다. 첫 번째 인수는 범위의 하한값을 지정하고 두 번째 인수는 범위의 상한값을 지정합니다.

> **Important**
>
> 이 함수는 높은 엔트로피 무작위화가 필요한 경우(예: 암호화)에는 사용해서는 안 됩니다.

$util.str의 문자열 도우미

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

`$util.str`에는 일반적인 문자열 작업에 도움이 되는 메서드가 있습니다.

$util.str 유틸리티 목록

`$util.str.toUpper(String) : String`

문자열을 가져와서 전체 대문자로 변환합니다.

`$util.str.toLower(String) : String`

문자열을 가져와서 전체 소문자로 변환합니다.

`$util.str.toReplace(String, String, String) : String`

문자열 내의 하위 문자열을 다른 문자열로 바꿉니다. 첫 번째 인수는 대체 작업을 수행할 문자열을 지정합니다. 두 번째 인수는 대체할 하위 문자열을 지정합니다. 세 번째 인수는 두 번째 인수를 대체할 문자열을 지정합니다. 다음은 이 유틸리티 사용 예제입니다.

```
INPUT:      $util.str.toReplace("hello world", "hello", "mellow")
OUTPUT:     "mellow world"
```

`$util.str.normalize(String, String) : String`

NFC, NFD, NFKC 또는 NFKD의 네 가지 유니코드 정규화 형식 중 하나를 사용하여 문자열을 정규화합니다. 첫 번째 인수는 정규화할 문자열입니다. 두 번째 인수는 정규화 프로세스에 사용할 정규화 유형을 지정하는 'nfc', 'nfd', 'nfkc' 또는 'nfkd'입니다.

확장 프로그램

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

`$extensions`에는 해석기 내에서 추가 작업을 수행할 수 있는 일련의 메서드가 있습니다.

$확장자. evictFromApi캐시 (문자열, 문자열, 개체): 개체

AWS AppSync 서버 측 캐시에서 항목을 제거합니다. 첫 번째 인수는 형식 이름입니다. 두 번째 인수는 필드 이름입니다. 세 번째 인수는 캐싱 키 값을 지정하는 키-값 쌍 항목을 포함하는 객체입니다. 캐시된 해석기의 `cachingKey`에 있는 캐싱 키와 동일한 순서로 객체에 항목을 넣어야 합니다.

> **Note**
>
> 이 유틸리티는 뮤테이션에만 작동하고 쿼리에는 작동하지 않습니다.

$확장자. setSubscriptionFilter() filterJsonObject

향상된 구독 필터를 정의합니다. 각 구독 알림 이벤트는 제공된 구독 필터에 대해 평가되고 모든 필터가 `true`로 평가되면 클라이언트에 알림을 전달합니다. 인수는 다음에 설명된대로 `filterJsonObject`입니다.

> **Note**
>
> 이 확장 메서드는 구독 해석기의 응답 매핑 템플릿에서만 사용할 수 있습니다.

$확장명. setSubscriptionInvalidation필터 () filterJsonObject

구독 무효화 필터를 정의합니다. 구독 필터는 무효화 페이로드에 대해 평가된 후 필터가 `true`로 평가되면 지정된 구독을 무효화합니다. 인수는 다음에 설명된대로 `filterJsonObject`입니다.

> **Note**
>
> 이 확장 메서드는 구독 해석기의 응답 매핑 템플릿에서만 사용할 수 있습니다.

인수: filterJsonObject

JSON 객체는 구독 또는 무효화 필터를 정의합니다. `filterGroup`에 있는 필터 배열입니다. 각 필터는 개별 필터의 모음입니다.

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

각 필터에는 다음의 세 가지 속성이 있습니다.

- `fieldName` - GraphQL 스키마 필드
- `operator` - 연산자 유형
- `value` - 구독 알림 `fieldName` 값과 비교할 값.

다음은 이러한 속성에 대한 할당 예시입니다.

```
{
 "fieldName" : "severity",
 "operator" : "le",
 "value" : $context.result.severity
}
```

필드: fieldName

문자열 유형 `fieldName`은 구독 알림 페이로드에서 `fieldName`과 일치하는 GraphQL 스키마에 정의된 필드를 나타냅니다. 일치가 발견되면 GraphQL 스키마 필드의 `value`는 구독 알림 필터의 `value`와 비교됩니다. 다음 예제에서 `fieldName` 필터는 주어진 GraphQL 유형에 정의된 `service` 필드와 일치합니다. 알림 페이로드에 `AWS  AppSync`와 동등한 `value`가 있는 `service` 필드가 포함된 경우 필터는 `true`로 평가합니다.

```
{
 "fieldName" : "service",
 "operator" : "eq",
 "value" : "AWS AppSync"
}
```

필드: 값

값은 연산자에 따라 다른 유형일 수 있습니다.

- 단일 숫자 또는 부울
  - 문자열 예제: `"test"`, `"service"`
  - 숫자 예제: `1`, `2`, `45.75`
  - 부울 예제: `true`, `false`
- 숫자 또는 문자열 페어
  - 문자열 페어 예제:`["test1","test2"]`, `["start","end"]`
  - 숫자 페어 예제: `[1,4]`, `[67,89]`, `[12.45,  95.45]`
- 숫자 또는 문자열 배열
  - 문자열 배열 예제: `["test1","test2","test3","test4","test5"]`
  - 숫자 배열 예제: `[1,2,3,4,5]`, `[12.11,46.13,45.09,12.54,13.89]`

필드: 연산자

대소문자를 구분하는 문자열로, 가능한 값은 다음과 같습니다.

연산자	설명	가능한 값 유형
eq	같음	정수, 부동 소수점, 문자열, 부울
ne	같지 않음	정수, 플로트, 문자열, 부울
le	작거나 같음	정수, 플로트, 문자열
lt	보다 작음	정수, 플로트, 문자열
ge	크거나 같음	정수, 플로트, 문자열
gt	보다 큼	정수, 플로트, 문자열
다음을 포함함	집합의 하위 시퀀스 또는 값을 확인합니다.	정수, 부동 소수점, 문자열
포함되지 않음	세트에 하위 시퀀스가 없거나 값이 없는지 확인합니다.	정수, 부동 소수점, 문자열
로 시작	접두사를 확인합니다.	문자열
인	목록에 있는 일치하는 요소가 있는지 확인합니다.	정수, 부동 소수점 또는 문자열로 구성된 배열
notIn	목록에 없는 요소가 일치하는지 확인합니다.	정수, 부동 소수점 또는 문자열 배열
사이	두 값 사이	정수, 플로트, 문자열
모든 항목 포함	공통 요소 포함	정수, 플로트, 문자열

다음 테이블에는 구독 알림에서 각 연산자가 사용되는 방식이 설명되어 있습니다.

eq (equal)

eq 연산자는 구독 알림 필드 값이 일치하고 필터 값과 엄격하게 동일하면 true로 평가됩니다. 다음 예에서 구독 알림에 AWS AppSync와 동등한 값을 가진 service 필드가 있는 경우 필터는 true로 평가합니다.

가능한 값 유형: 정수, 부동 소수점, 문자열, 부울

```
{
 "fieldName" : "service",
 "operator" : "eq",
 "value" : "AWS AppSync"
}
```

ne (not equal)

ne 연산자는 구독 알림 필드 값이 필터 값과 다르면 true로 평가됩니다. 다음 예에서는 구독 알림에 AWS AppSync와 다른 값을 가진 service 필드가 있는 경우 필터는 true로 평가합니다.

가능한 값 유형: 정수, 부동 소수점, 문자열, 부울

```
{
 "fieldName" : "service",
 "operator" : "ne",
 "value" : "AWS AppSync"
}
```

le (less or equal)

le 연산자는 구독 알림 필드 값이 필터 값과 같거나 그보다 작으면 true로 평가됩니다. 다음 예에서는 구독 알림에 5와 같거나 그보다 작은 값을 가진 size 필드가 있는 경우 필터는 true로 평가합니다.

가능한 값 유형: 정수, 부동 소수점, 문자열

```
{
 "fieldName" : "size",
 "operator" : "le",
 "value" : 5
}
```

lt (less than)

`lt` 연산자는 구독 알림 필드 값이 필터 값보다 낮으면 `true`로 평가됩니다. 다음 예에서는 구독 알림에 5보다 낮은 값을 가진 `size` 필드가 있는 경우 필터는 `true`로 평가합니다.

가능한 값 유형: 정수, 부동 소수점, 문자열

```
{
 "fieldName" : "size",
 "operator" : "lt",
 "value" : 5
}
```

ge (greater or equal)

`ge` 연산자는 구독 알림 필드 값이 필터 값과 같거나 그보다 크면 `true`로 평가됩니다. 다음 예에서는 구독 알림에 5와 같거나 그보다 큰 값을 가진 `size` 필드가 있는 경우 필터는 `true`로 평가합니다.

가능한 값 유형: 정수, 부동 소수점, 문자열

```
{
 "fieldName" : "size",
 "operator" : "ge",
 "value" : 5
}
```

gt (greater than)

`gt` 연산자는 구독 알림 필드 값이 필터 값보다 크면 `true`로 평가됩니다. 다음 예에서는 구독 알림에 5보다 높은 값을 가진 `size` 필드가 있는 경우 필터는 `true`로 평가합니다.

가능한 값 유형: 정수, 부동 소수점, 문자열

```
{
 "fieldName" : "size",
 "operator" : "gt",
 "value" : 5
}
```

contains

contains 연산자는 집합이나 단일 항목의 하위 문자열, 하위 시퀀스 또는 값을 확인합니다. 구독 알림 필드 값에 필터 값이 포함된 경우 contains 연산자가 있는 필터는 true로 평가됩니다. 다음 예에서는 구독 알림에 값 10이 포함된 배열 값이 있는 seats 필드가 있는 경우 필터는 true로 평가합니다.

가능한 값 유형: 정수, 부동 소수점, 문자열

```
{
 "fieldName" : "seats",
 "operator" : "contains",
 "value" : 10
}
```

다른 예로, 필터는 구독 알림에 launch가 하위 문자열로 포함된 event 필드가 있는 경우 true로 평가합니다.

```
{
 "fieldName" : "event",
 "operator" : "contains",
 "value" : "launch"
}
```

notContains

notContains 연산자는 집합이나 단일 항목에 하위 문자열, 하위 시퀀스 또는 값이 없는지 확인합니다. 구독 알림 필드 값에 필터 값이 포함되지 않은 경우 notContains 연산자가 있는 필터는 true로 평가됩니다. 다음 예에서는 구독 알림에 값 10이 포함되지 않은 배열 값이 있는 seats 필드가 있는 경우 필터는 true로 평가합니다.

가능한 값 유형: 정수, 부동 소수점, 문자열

```
{
 "fieldName" : "seats",
 "operator" : "notContains",
 "value" : 10
}
```

다른 예로, 필터는 구독 알림에 launch가 하위 문자열로 포함되지 않은 event 필드 값이 있는 경우 true로 평가합니다.

```
{
 "fieldName" : "event",
 "operator" : "notContains",
 "value" : "launch"
}
```

beginsWith

beginsWith 연산자는 문자열에서 접두사를 확인합니다. 구독 알림 필드 값이 필터 값으로 시작하는 경우 beginsWith 연산자가 포함된 필터는 true로 평가됩니다. 다음 예에서는 구독 알림에 값이 AWS로 시작하는 service 필드가 있는 경우 필터는 true로 평가합니다.

가능한 값 유형: 문자열

```
{
 "fieldName" : "service",
 "operator" : "beginsWith",
 "value" : "AWS"
}
```

in

in 연산자는 배열에서 일치하는 요소가 있는지 확인합니다. in 연산자가 포함된 필터는 구독 알림 필드 값이 배열에 존재하는 경우 true로 평가됩니다. 다음 예에서는 구독 알림에 [1,2,3] 배열에 있는 값 중 하나가 포함된 severity 필드가 있는 경우 필터는 true로 평가합니다.

가능한 값 유형: 정수 배열, 부동 소수점 또는 문자열

```
{
 "fieldName" : "severity",
 "operator" : "in",
 "value" : [1,2,3]
}
```

notIn

notIn 연산자는 배열에서 누락된 요소가 있는지 확인합니다. notIn 연산자가 포함된 필터는 구독 알림 필드 값이 배열에 존재하지 않는 경우 true로 평가됩니다. 다음 예에서는 구독 알림에 [1,2,3] 배열에 없는 값 중 하나가 포함된 severity 필드가 있는 경우 필터는 true로 평가합니다.

가능한 값 유형: 정수 배열, 부동 소수점 또는 문자열

```
{
 "fieldName" : "severity",
 "operator" : "notIn",
 "value" : [1,2,3]
}
```

between

between 연산자는 두 숫자 또는 문자열 사이의 값을 확인합니다. 구독 알림 필드 값이 필터의 값 페어 사이에 있는 경우 between 연산자가 포함된 필터는 true로 평가됩니다. 다음 예에서는 구독 알림에 값이 2,3,4인 severity 필드가 있는 경우 필터는 true로 평가합니다.

가능한 값 유형: 정수 페어, 부동 소수점 또는 문자열

```
{
 "fieldName" : "severity",
 "operator" : "between",
 "value" : [1,5]
}
```

containsAny

containsAny 연산자는 배열에서 공통된 요소가 있는지 확인합니다. 구독 알림 필드 집합 값 및 필터 집합 값의 교차점이 비어 있지 않은 경우 containsAny 연산자가 있는 필터는 true로 평가됩니다. 다음 예에서는 구독 알림에 10 또는 15를 포함하는 배열 값을 가진 seats 필드가 있는 경우 필터는 true로 평가합니다. 이는 구독 알림에 [10,11] 또는 [15,20,30]라는 seats 필드 값이 있는 경우 필터가 true로 평가한다는 의미입니다.

가능한 값 유형: 정수, 부동 소수점 또는 문자열

```
{
 "fieldName" : "seats",
 "operator" : "containsAny",
 "value" : [10, 15]
}
```

AND 로직

filterGroup 배열의 filters 객체 내에 여러 항목을 정의하여 AND 논리를 사용하여 여러 필터를 결합할 수 있습니다. 다음 예에서는 구독 알림에 값이 1인 userId 필드가 있고(AND) Admin 또는 Developer인 group 필드 값이 있는 경우 필터는 true로 평가합니다.

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

OR 로직

filterGroup 배열 내에서 여러 필터 객체를 정의하면 OR 논리를 사용하여 여러 필터를 결합할 수 있습니다. 다음 예에서는 구독 알림에 값이 1인 userId 필드가 있거나(OR) Admin 또는 Developer인 group 필드 값이 있는 경우 필터는 true로 평가합니다.

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]
```

```
        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

예외

필터 사용에는 몇 가지 제한이 있다는 점에 유의하세요.

- `filters` 객체에는 필터당 최대 5개의 고유 `fieldName` 항목이 있을 수 있습니다. 즉, AND 로직을 사용하여 최대 5개의 개별 `fieldName` 객체를 결합할 수 있습니다.
- `containsAny` 연산자에는 최대 20개의 값이 있을 수 있습니다.
- `in` 및 `notIn` 연산자에는 최대 5개의 값이 있을 수 있습니다.
- 각 연결 문자열은 최대 256자입니다.
- 각 문자열 비교는 대/소문자를 구분합니다.
- 중첩된 객체 필터링은 최대 5개의 중첩 수준 필터링을 허용합니다.
- 각 `filterGroup`에는 최대 10개의 `filters`가 있을 수 있습니다. 즉, OR 로직을 사용하여 최대 10개의 개별 `filters`를 결합할 수 있습니다.
  - `in` 연산자는 OR 논리의 특별 사례입니다. 다음 예제에서는 두 가지 `filters`가 있습니다.

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
```

```
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
            ]
        }
    ]
}
```

이전 필터 그룹은 다음과 같이 평가되며 최대 필터 한도에 포함됩니다.

```
{
    "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"
                }
            ]
        }
    ]
}
```

$확장자. 구독 무효화 () invalidationJsonObject

뮤테이션으로부터 구독 무효화를 시작하는 데 사용됩니다. 인수는 다음에 설명된대로 `invalidationJsonObject`입니다.

> **Note**
>
> 이 확장은 뮤테이션 해석기의 응답 매핑 템플릿에서만 사용할 수 있습니다.
> 단일 요청에서 고유한 `$extensions.invalidateSubscriptions()` 메서드 호출을 최대 5개까지만 사용할 수 있습니다. 이 한도를 초과할 경우 GraphQL 오류가 발생합니다.

인수: invalidationJsonObject

`invalidationJsonObject`는 다음을 정의합니다.

- `subscriptionField` - 무효화할 GraphQL 스키마 구독입니다. `subscriptionField`에서 문자열로 정의된 단일 구독은 무효화 대상으로 간주됩니다.
- `payload` – 무효화 필터가 해당 값에 대해 `true`로 평가되는 경우 구독 무효화를 위한 입력으로 사용되는 키-값 쌍 목록입니다.

  다음 예에서는 구독 해석기에 정의된 무효화 필터가 `payload` 값에 대해 `true`로 평가될 때 `onUserDelete` 구독을 사용하여 구독 및 연결된 클라이언트를 무효화합니다.

```
$extensions.invalidateSubscriptions({
        "subscriptionField": "onUserDelete",
        "payload": {
                "group": "Developer"
                "type" : "Full-Time"
      }
    })
```

DynamoDB에 대한 해석기 매핑 템플릿 참조

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync DynamoDB 해석기에서는 GraphQL을 사용하여 자신의 계정에 있는 기존 Amazon DynamoDB 테이블의 데이터를 저장 및 검색할 수 있습니다. 이 해석기는 수신되는 GraphQL 요청을 DynamoDB 호출로 매핑한 다음 DynamoDB 응답을 다시 GraphQL로 매핑하도록 하여 작동합니다. 이 단원에서는 지원되는 DynamoDB 작업에 대한 매핑 템플릿에 대해 설명합니다.

GetItem

`GetItem` 요청 매핑 문서를 사용하면 AWS AppSync DynamoDB 해석기에 DynamoDB에 `GetItem` 요청을 하도록 지시하고 다음을 지정할 수 있습니다.

- DynamoDB의 항목 키
- 일관된 읽기를 사용할지 여부

`GetItem` 매핑 문서의 구조는 다음과 같습니다.

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true,
    "projection" : {
        ...
    }
}
```

필드는 다음과 같이 정의됩니다.

GetItem 필드

GetItem 필드 목록

`version`

템플릿 정의 버전. `2017-02-28` 및 `2018-05-29`만 현재 지원됩니다. 이 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `GetItem` DynamoDB 작업을 수행하려면 이 값을 `GetItem`으로 설정해야 합니다. 이 값은 필수입니다.

`key`

DynamoDB의 항목 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

`consistentRead`

DynamoDB에서 강력히 일관된 읽기를 수행할지 여부. 선택 사항으로, 기본값은 `false`입니다.

`projection`

DynamoDB 작업에서 반환할 속성을 지정하는 데 사용되는 프로젝션입니다. 프로젝션에 대한 자세한 내용은 프로젝션을 참조하세요. 이 필드는 선택 사항입니다.

DynamoDB에서 반환된 항목은 자동으로 GraphQL 및 JSON 기본 유형으로 변환되며 매핑 컨텍스트(`$context.result`)에서 사용할 수 있습니다.

DynamoDB 형식 변환에 대한 자세한 내용은 형식 시스템(응답 매핑)을 참조하세요.

응답 매핑 템플릿에 대한 자세한 내용은 해석기 매핑 템플릿 개요를 참조하세요.

예

다음 예제는 GraphQL 쿼리 `getThing(foo: String!, bar: String!)`의 매핑 템플릿입니다.

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
        "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
    },
    "consistentRead" : true
}
```

DynamoDB `GetItem` API에 대한 자세한 내용은 DynamoDB API 문서를 참조하십시오.

PutItem

`PutItem` 요청 매핑 문서를 사용하면 AWS AppSync DynamoDB 해석기에 DynamoDB에 `PutItem` 요청을 하도록 지시하고 다음을 지정할 수 있습니다.

- DynamoDB의 항목 키
- 항목의 전체 내용(key 및 attributeValues로 구성됨)
- 성공할 작업의 조건

PutItem 매핑 문서의 구조는 다음과 같습니다.

```
{
    "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
}
```

필드는 다음과 같이 정의됩니다.

PutItem 필드

PutItem 필드 목록

`version`

템플릿 정의 버전. `2017-02-28` 및 `2018-05-29`만 현재 지원됩니다. 이 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `PutItem` DynamoDB 작업을 수행하려면 이 값을 `PutItem`으로 설정해야 합니다. 이 값은 필수입니다.

`key`

DynamoDB의 항목 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

`attributeValues`

DynamoDB에 저장할 항목의 나머지 속성. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 필드는 선택 사항입니다.

`condition`

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건을 지정하지 않으면 `PutItem` 요청이 해당 항목에 대한 기존 입력을 덮어씁니다. 조건에 대한 자세한 내용은 조건 표현식을 참조하세요. 이 값은 선택 사항입니다.

`_version`

항목의 알려진 최신 버전을 나타내는 숫자 값입니다. 이 값은 선택 사항입니다. 이 필드는 충돌 감지에 사용되며 버전이 지정된 데이터 원본에만 지원됩니다.

`customPartitionKey`

활성화되면 이 문자열 값은 버전 관리가 활성화되었을 때 델타 동기화 테이블에서 사용되는 `ds_sk` 및 `ds_pk` 레코드의 형식을 수정합니다(자세한 내용은 AWS AppSync 개발자 안내서의 충돌 감지 및 동기화 참조). 활성화되면 `populateIndexFields` 항목 처리도 활성화됩니다. 이 필드는 선택 사항입니다.

`populateIndexFields`

`customPartitionKey`와 함께 활성화되면 델타 동기화 테이블, 특히 `gsi_ds_pk` 및 `gsi_ds_sk` 열의 각 레코드에 대해 새 항목을 생성하는 부울 값입니다. 자세한 내용은 AWS AppSync 개발자 안내서의 충돌 감지 및 동기화를 참조하세요. 이 필드는 선택 사항입니다.

DynamoDB에 기록된 항목은 자동으로 GraphQL 및 JSON 기본 형식으로 변환되며 매핑 컨텍스트(`$context.result`)에서 사용할 수 있습니다.

DynamoDB 형식 변환에 대한 자세한 내용은 형식 시스템(응답 매핑)을 참조하세요.

응답 매핑 템플릿에 대한 자세한 내용은 해석기 매핑 템플릿 개요를 참조하세요.

예 1

다음 예제는 GraphQL 변형 `updateThing(foo: String!, bar: String!, name: String!, version: Int!)`의 매핑 템플릿입니다.

지정된 키가 있는 항목이 없으면 생성됩니다. 지정된 키가 있는 항목이 있으면 덮어씁니다.

```
{
    "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)
    }
}
```

예 2

다음 예제는 GraphQL 변형 `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)`의 매핑 템플릿입니다.

이 예에서는 현재 DynamoDB에 있는 항목의 `version` 필드가 `expectedVersion`으로 설정되어 있는지 확인합니다.

```
{
    "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)
        }
    }
}
```

DynamoDB `PutItem` API에 대한 자세한 내용은 DynamoDB API 문서를 참조하십시오.

UpdateItem

`UpdateItem` 요청 매핑 문서를 사용하면 AWS AppSync DynamoDB 해석기에 DynamoDB에 `UpdateItem` 요청을 하도록 지시하고 다음을 지정할 수 있습니다.

- DynamoDB의 항목 키
- DynamoDB에서 항목 업데이트 방법을 설명하는 업데이트 표현식
- 성공할 작업의 조건

`UpdateItem` 매핑 문서의 구조는 다음과 같습니다.

```
{
    "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
}
```

필드는 다음과 같이 정의됩니다.

UpdateItem 필드

UpdateItem 필드 목록

`version`

템플릿 정의 버전. `2017-02-28` 및 `2018-05-29`만 현재 지원됩니다. 이 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `UpdateItem` DynamoDB 작업을 수행하려면 이 값을 `UpdateItem`으로 설정해야 합니다. 이 값은 필수입니다.

`key`

DynamoDB의 항목 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값' 지정에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

`update`

`update` 섹션에서는 DynamoDB의 항목 업데이트 방법을 설명하는 업데이트 표현식을 지정합니다. 업데이트 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB UpdateExpressions 문서를 참조하십시오. 이 섹션은 필수입니다.

`update` 섹션에는 다음 세 가지 구성 요소가 있습니다.

`expression`

업데이트 표현식. 이 값은 필수입니다.

`expressionNames`

표현식 속성 name 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 `expression`에 사용된 name 자리 표시자에 해당하고 값은 DynamoDB에 있는 항목의 속성 이름에 해당하는 문자열이어야 합니다. 이 필드는 선택 사항으로, `expression`에 사용된 표현식 속성인 name 자리 표시자의 대체 항목으로만 채워져야 합니다.

`expressionValues`

표현식 속성 value 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 `expression`에 사용되는 value 자리 표시자에 해당하고 값은 입력된 값이어야 합니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 입력된 값은

지정되어 있어야 합니다. 이 필드는 선택 사항으로, expression에 사용된 표현식 속성인 value 자리 표시자의 대체 항목으로만 채워져야 합니다.

`condition`

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건을 지정하지 않으면 UpdateItem 요청이 현재 상태와 상관 없이 기존 항목을 업데이트합니다. 조건에 대한 자세한 내용은 조건 표현식을 참조하세요. 이 값은 선택 사항입니다.

`_version`

항목의 알려진 최신 버전을 나타내는 숫자 값입니다. 이 값은 선택 사항입니다. 이 필드는 충돌 감지에 사용되며 버전이 지정된 데이터 원본에만 지원됩니다.

`customPartitionKey`

활성화되면 이 문자열 값은 버전 관리가 활성화되었을 때 델타 동기화 테이블에서 사용되는 ds_sk 및 ds_pk 레코드의 형식을 수정합니다(자세한 내용은 AWS AppSync 개발자 안내서의 충돌 감지 및 동기화 참조). 활성화되면 populateIndexFields 항목 처리도 활성화됩니다. 이 필드는 선택 사항입니다.

`populateIndexFields`

customPartitionKey와 함께 활성화되면 델타 동기화 테이블, 특히 gsi_ds_pk 및 gsi_ds_sk 열의 각 레코드에 대해 새 항목을 생성하는 부울 값입니다. 자세한 내용은 AWS AppSync 개발자 안내서의 충돌 감지 및 동기화를 참조하세요. 이 필드는 선택 사항입니다.

DynamoDB에서 업데이트된 항목은 자동으로 GraphQL 및 JSON 기본 형식으로 변환되며 매핑 컨텍스트($context.result)에서 사용할 수 있습니다.

DynamoDB 형식 변환에 대한 자세한 내용은 형식 시스템(응답 매핑)을 참조하세요.

응답 매핑 템플릿에 대한 자세한 내용은 해석기 매핑 템플릿 개요를 참조하세요.

예 1

다음 예제는 GraphQL 변형 upvote(id: ID!)의 매핑 템플릿입니다.

이 예에서 DynamoDB의 항목에는 1씩 증가하는 upvotes 및 version 필드가 있습니다.

```
{
    "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 }
        }
    }
}
```

예 2

다음 예제는 GraphQL 변형 `updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)`의 매핑 템플릿입니다.

다음은 인수를 조사해 클라이언트에서 제공한 인수만 포함하는 업데이트 표현식을 동적으로 생성하는 복잡한 예입니다. 예를 들어, `title` 및 `author`는 생략되면 업데이트되지 않습니다. 인수가 지정되어 있으나 그 값이 `null`이면 DynamoDB의 객체에서 해당 필드가 삭제됩니다. 마지막으로, 이 작업에는 DynamoDB에 현재 있는 항목에 `expectedVersion`으로 설정된 `version` 필드가 있는지 확인하는 조건이 있습니다.

```
{
    "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)
        }
    }
}
```

DynamoDB `UpdateItem` API에 대한 자세한 내용은 DynamoDB API 문서를 참조하십시오.

DeleteItem

`DeleteItem` 요청 매핑 문서를 사용하면 AWS AppSync DynamoDB 해석기에 DynamoDB에 `DeleteItem` 요청을 하도록 지시하고 다음을 지정할 수 있습니다.

- DynamoDB의 항목 키
- 성공할 작업의 조건

`DeleteItem` 매핑 문서의 구조는 다음과 같습니다.

```
{
    "version" : "2018-05-29",
    "operation" : "DeleteItem",
    "customPartitionKey" : "foo",
    "populateIndexFields" : boolean value,
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "condition" : {
        ...
    },
    "_version" : 1
}
```

필드는 다음과 같이 정의됩니다.

DeleteItem 필드

DeleteItem 필드 목록

`version`

템플릿 정의 버전. `2017-02-28` 및 `2018-05-29`만 현재 지원됩니다. 이 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `DeleteItem` DynamoDB 작업을 수행하려면 이 값을 `DeleteItem`으로 설정해야 합니다. 이 값은 필수입니다.

key

DynamoDB의 항목 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값' 지정에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

condition

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건을 지정하지 않으면 `DeleteItem` 요청이 현재 상태와 상관 없이 항목을 삭제합니다. 조건에 대한 자세한 내용은 조건 표현식을 참조하세요. 이 값은 선택 사항입니다.

_version

항목의 알려진 최신 버전을 나타내는 숫자 값입니다. 이 값은 선택 사항입니다. 이 필드는 충돌 감지에 사용되며 버전이 지정된 데이터 원본에만 지원됩니다.

customPartitionKey

활성화되면 이 문자열 값은 버전 관리가 활성화되었을 때 델타 동기화 테이블에서 사용되는 `ds_sk` 및 `ds_pk` 레코드의 형식을 수정합니다(자세한 내용은 AWS AppSync 개발자 안내서의 충돌 감지 및 동기화 참조). 활성화되면 `populateIndexFields` 항목 처리도 활성화됩니다. 이 필드는 선택 사항입니다.

populateIndexFields

customPartitionKey와 함께 활성화되면 델타 동기화 테이블, 특히 `gsi_ds_pk` 및 `gsi_ds_sk` 열의 각 레코드에 대해 새 항목을 생성하는 부울 값입니다. 자세한 내용은 AWS AppSync 개발자 안내서의 충돌 감지 및 동기화를 참조하세요. 이 필드는 선택 사항입니다.

DynamoDB에서 삭제된 항목은 자동으로 GraphQL 및 JSON 기본 형식으로 변환되며 매핑 컨텍스트(`$context.result`)에서 사용할 수 있습니다.

DynamoDB 형식 변환에 대한 자세한 내용은 형식 시스템(응답 매핑)을 참조하세요.

응답 매핑 템플릿에 대한 자세한 내용은 해석기 매핑 템플릿 개요를 참조하세요.

예 1

다음 예제는 GraphQL 변형 `deleteItem(id: ID!)`의 매핑 템플릿입니다. 이 ID를 가진 항목이 있으면 해당 항목은 삭제됩니다.

```
{
    "version" : "2017-02-28",
    "operation" : "DeleteItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

예 2

다음 예제는 GraphQL 변형 `deleteItem(id: ID!, expectedVersion: Int!)`의 매핑 템플릿입니다. 이 ID를 가진 항목이 있으면 해당 항목은 `version` 필드가 `expectedVersion`으로 설정된 경우에만 삭제됩니다.

```
{
    "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)
        }
    }
}
```

DynamoDB `DeleteItem` API에 대한 자세한 내용은 DynamoDB API 문서를 참조하십시오.

쿼리

`Query` 요청 매핑 문서를 사용하면 AWS AppSync DynamoDB 해석기에 DynamoDB에 `Query` 요청을 하도록 지시하고 다음을 지정할 수 있습니다.

- 키 표현식
- 사용할 인덱스
- 모든 추가 필터
- 반환할 항목 수

- 일관된 읽기를 사용할지 여부
- 쿼리 방향(앞으로 또는 뒤로)
- 페이지 매김 토큰입니다

Query 매핑 문서의 구조는 다음과 같습니다.

```
{
    "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" : {
        ...
    }
}
```

필드는 다음과 같이 정의됩니다.

쿼리 필드

쿼리 필드 목록

`version`

템플릿 정의 버전. `2017-02-28` 및 `2018-05-29`만 현재 지원됩니다. 이 값은 필수입니다.

operation

수행할 DynamoDB 작업입니다. `Query` DynamoDB 작업을 수행하려면 이 값을 `Query`으로 설정해야 합니다. 이 값은 필수입니다.

query

`query` 섹션에서는 DynamoDB에서 가져올 항목을 설명하는 키 조건 표현식을 지정할 수 있습니다. 키 조건 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB KeyConditions 문서를 참조하십시오. 이 섹션은 지정되어 있어야 합니다.

expression

쿼리 표현식. 이 필드는 지정되어 있어야 합니다.

expressionNames

표현식 속성 name 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 `expression`에 사용된 name 자리 표시자에 해당하고 값은 DynamoDB에 있는 항목의 속성 이름에 해당하는 문자열이어야 합니다. 이 필드는 선택 사항으로, `expression`에 사용된 표현식 속성인 name 자리 표시자의 대체 항목으로만 채워져야 합니다.

expressionValues

표현식 속성 value 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 `expression`에 사용되는 value 자리 표시자에 해당하고 값은 입력된 값이어야 합니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다. 이 필드는 선택 사항으로, `expression`에 사용된 표현식 속성인 value 자리 표시자의 대체 항목으로만 채워져야 합니다.

filter

반환되기 전 DynamoDB의 결과를 필터링하는 데 사용할 수 있는 추가 필터입니다. 필터에 대한 자세한 내용은 필터를 참조하십시오. 이 필드는 선택 사항입니다.

index

쿼리할 인덱스의 이름. DynamoDB 쿼리 작업을 사용하면 해시 키의 프라이머리 키 인덱스 이외에 로컬 보조 인덱스 및 글로벌 보조 인덱스를 검사할 수 있습니다. 지정하면 이 값은 DynamoDB에 지정한 인덱스를 쿼리하라고 지시합니다. 이 값을 생략하면 기본 키 인덱스가 쿼리됩니다.

nextToken

이전 쿼리를 지속하는 페이지 매김 토큰. 이 토큰은 이전 쿼리에서 얻습니다. 이 필드는 선택 사항입니다.

limit

평가할 최대 항목 수입니다(반드시 일치하는 항목 수는 아님). 이 필드는 선택 사항입니다.

scanIndexForward

앞으로 또는 뒤로 쿼리할지 여부를 나타내는 부울. 이 필드는 선택 사항으로, 기본값은 `true`입니다.

consistentRead

DynamoDB 쿼리 시 일관된 읽기를 사용할지 여부를 나타내는 부울. 이 필드는 선택 사항으로, 기본값은 `false`입니다.

select

기본적으로 AWS AppSync DynamoDB 해석기는 인덱스로 프로젝션된 속성만 반환합니다. 추가 속성이 필요한 경우 이 필드를 설정할 수 있습니다. 이 필드는 선택 사항입니다. 지원되는 값은 다음과 같습니다.

ALL_ATTRIBUTES

지정한 테이블 또는 인덱스에서 항목 속성을 모두 반환합니다. 로컬 보조 인덱스를 쿼리하는 경우 DynamoDB는 인덱스의 일치하는 각 항목에 대해 상위 테이블의 전체 항목을 가져옵니다. 인덱스가 모든 항목 속성을 프로젝션하도록 구성된 경우, 모든 데이터를 로컬의 보조 인덱스에서 얻을 수 있기 때문에 가져올 필요가 없습니다.

ALL_PROJECTED_ATTRIBUTES

인덱스를 쿼리하는 경우에만 허용됩니다. 인덱스로 프로젝션된 모든 속성을 가져옵니다. 모든 속성을 프로젝션하도록 인덱스가 구성된 경우 이 반환 값은 `ALL_ATTRIBUTES`를 지정하는 것과 동일합니다.

SPECIFIC_ATTRIBUTES

`projection`의 `expression`에 나열된 속성만 반환합니다. 이 반환 값은 `Select`에 대한 값을 지정하지 않고 `projection`의 `expression`을 지정하는 것과 같습니다.

projection

DynamoDB 작업에서 반환할 속성을 지정하는 데 사용되는 프로젝션입니다. 프로젝션에 대한 자세한 내용은 프로젝션을 참조하세요. 이 필드는 선택 사항입니다.

DynamoDB의 결과는 자동으로 GraphQL 및 JSON 기본 형식으로 변환되며 매핑 컨텍스트(`$context.result`)에서 사용할 수 있습니다.

DynamoDB 형식 변환에 대한 자세한 내용은 형식 시스템(응답 매핑)을 참조하세요.

응답 매핑 템플릿에 대한 자세한 내용은 해석기 매핑 템플릿 개요를 참조하세요.

결과의 구조는 다음과 같습니다.

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

필드는 다음과 같이 정의됩니다.

`items`

DynamoDB 쿼리에서 반환하는 항목이 포함된 목록.

`nextToken`

더 많은 결과가 있을 수 있는 경우 `nextToken`에는 다른 요청에 사용할 수 있는 페이지 매김 토큰이 포함됩니다. AWS AppSync 에서는 DynamoDB에서 반환된 페이지 매김 토큰을 암호화하고 난독화합니다. 따라서 테이블 데이터가 호출자에게 실수로 유출되는 일이 없습니다. 페이지 매김 토큰은 여러 해석기 간에 사용할 수 없습니다.

`scannedCount`

필터 표현식(있는 경우)을 적용하기 전에 쿼리 조건 표현식과 일치했던 항목 수

예

다음 예제는 GraphQL 쿼리 `getPosts(owner: ID!)`의 매핑 템플릿입니다.

이 예에서는 테이블의 글로벌 보조 인덱스를 쿼리해 지정한 ID가 소유한 게시물을 모두 반환합니다.

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "query" : {
        "expression" : "ownerId = :ownerId",
        "expressionValues" : {
            ":ownerId" : $util.dynamodb.toDynamoDBJson($context.arguments.owner)
```

```
        }
    },
    "index" : "owner-index"
}
```

DynamoDB Query API에 대한 자세한 내용은 DynamoDB API 문서를 참조하십시오.

스캔

Scan 요청 매핑 문서를 사용하면 AWS AppSync DynamoDB 해석기에 DynamoDB에 Scan 요청을 하도록 지시하고 다음을 지정할 수 있습니다.

- 결과를 제외하는 필터
- 사용할 인덱스
- 반환할 항목 수
- 일관된 읽기를 사용할지 여부
- 페이지 매김 토큰입니다
- 병렬 스캔

Scan 매핑 문서의 구조는 다음과 같습니다.

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "index" : "fooIndex",
    "limit" : 10,
    "consistentRead" : false,
    "nextToken" : "aPaginationToken",
    "totalSegments" : 10,
    "segment" : 1,
    "filter" : {
        ...
    },
    "projection" : {
        ...
    }
}
```

필드는 다음과 같이 정의됩니다.

Scan 필드

Scan 필드 목록

`version`

템플릿 정의 버전. `2017-02-28` 및 `2018-05-29`만 현재 지원됩니다. 이 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `Scan` DynamoDB 작업을 수행하려면 이 값을 `Scan`으로 설정해야 합니다. 이 값은 필수입니다.

`filter`

반환되기 전 DynamoDB의 결과를 필터링하는 데 사용할 수 있는 필터. 필터에 대한 자세한 내용은 필터를 참조하십시오. 이 필드는 선택 사항입니다.

`index`

쿼리할 인덱스의 이름. DynamoDB 쿼리 작업을 사용하면 해시 키의 프라이머리 키 인덱스 이외에 로컬 보조 인덱스 및 글로벌 보조 인덱스를 검사할 수 있습니다. 지정하면 이 값은 DynamoDB에 지정한 인덱스를 쿼리하라고 지시합니다. 이 값을 생략하면 기본 키 인덱스가 쿼리됩니다.

`limit`

한 번에 평가올 수 있는 최대 항목 수입니다. 이 필드는 선택 사항입니다.

`consistentRead`

DynamoDB 쿼리 시 일관된 읽기를 사용할지 여부를 나타내는 부울. 이 필드는 선택 사항으로, 기본값은 `false`입니다.

`nextToken`

이전 쿼리를 지속하는 페이지 매김 토큰. 이 토큰은 이전 쿼리에서 얻습니다. 이 필드는 선택 사항입니다.

`select`

기본적으로 AWS AppSync DynamoDB 해석기는 인덱스로 프로젝션된 속성만 반환합니다. 추가 속성이 필요한 경우 이 필드를 설정할 수 있습니다. 이 필드는 선택 사항입니다. 지원되는 값은 다음과 같습니다.

`ALL_ATTRIBUTES`

지정한 테이블 또는 인덱스에서 항목 속성을 모두 반환합니다. 로컬 보조 인덱스를 쿼리하는 경우 DynamoDB는 인덱스의 일치하는 각 항목에 대해 상위 테이블의 전체 항목을 가져옵니다. 인

덱스가 모든 항목 속성을 프로젝션하도록 구성된 경우, 모든 데이터를 로컬의 보조 인덱스에서 얻을 수 있기 때문에 가져올 필요가 없습니다.

ALL_PROJECTED_ATTRIBUTES

인덱스를 쿼리하는 경우에만 허용됩니다. 인덱스로 프로젝션된 모든 속성을 가져옵니다. 모든 속성을 프로젝션하도록 인덱스가 구성된 경우 이 반환 값은 ALL_ATTRIBUTES를 지정하는 것과 동일합니다.

SPECIFIC_ATTRIBUTES

projection의 expression에 나열된 속성만 반환합니다. 이 반환 값은 Select에 대한 값을 지정하지 않고 projection의 expression을 지정하는 것과 같습니다.

totalSegments

병렬 스캔을 수행할 때 테이블을 분할할 기준이 될 세그먼트 수. 이 필드는 선택 사항이지만 segment가 지정된 경우에는 반드시 지정해야 합니다.

segment

병렬 스캔을 수행할 때 작업의 테이블 세그먼트. 이 필드는 선택 사항이지만 totalSegments가 지정된 경우에는 반드시 지정해야 합니다.

projection

DynamoDB 작업에서 반환할 속성을 지정하는 데 사용되는 프로젝션입니다. 프로젝션에 대한 자세한 내용은 프로젝션을 참조하세요. 이 필드는 선택 사항입니다.

DynamoDB 스캔에서 반환하는 결과는 자동으로 GraphQL 및 JSON 기본 형식으로 변환되며 매핑 컨텍스트($context.result)에서 사용할 수 있습니다.

DynamoDB 형식 변환에 대한 자세한 내용은 형식 시스템(응답 매핑)을 참조하세요.

응답 매핑 템플릿에 대한 자세한 내용은 해석기 매핑 템플릿 개요를 참조하세요.

결과의 구조는 다음과 같습니다.

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

필드는 다음과 같이 정의됩니다.

`items`

DynamoDB 스캔에서 반환하는 항목이 포함된 목록

`nextToken`

더 많은 결과가 있을 수 있는 경우 `nextToken`에는 다른 요청에 사용할 수 있는 페이지 매김 토큰이 포함됩니다. AppSync에서는 DynamoDB에서 반환된 페이지 매김 토큰을 암호화하고 난독화합니다. 따라서 테이블 데이터가 호출자에게 실수로 유출되는 일이 없습니다. 또한 페이지 매김 토큰은 여러 해석기 간에 사용할 수 없습니다.

`scannedCount`

필터 표현식(있는 경우)을 적용하기 전에 DynamoDB에서 검색한 항목 수.

예 1

다음 예제는 GraphQL 쿼리 `allPosts`의 매핑 템플릿입니다.

이 예에서는 테이블의 항목을 모두 반환합니다.

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
}
```

예 2

다음 예제는 GraphQL 쿼리 `postsMatching(title: String!)`의 매핑 템플릿입니다.

이 예에서는 제목이 `title` 인수로 시작하는 경우 테이블의 모든 항목이 반환됩니다.

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "filter" : {
        "expression" : "begins_with(title, :title)",
        "expressionValues" : {
            ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
        },
```

```
    }
}
```

DynamoDB Scan API에 대한 자세한 내용은 DynamoDB API 문서를 참조하십시오.

동기화

Sync 요청 매핑 문서를 사용하면 DynamoDB 테이블에서 모든 결과를 가져온 다음, 마지막 쿼리(델타 업데이트) 이후에 변경된 데이터만 수신할 수 있습니다. Sync 요청은 버전이 지정된 DynamoDB 데이터 소스에만 수행할 수 있습니다. 지정할 수 있는 셰이프는 다음과 같습니다.

- 결과를 제외하는 필터
- 반환할 항목 수
- 페이지 매김 토큰
- 마지막 Sync 작업이 시작된 경우

Sync 매핑 문서의 구조는 다음과 같습니다.

```
{
    "version" : "2018-05-29",
    "operation" : "Sync",
    "basePartitionKey": "Base Tables PartitionKey",
    "deltaIndexName": "delta-index-name",
    "limit" : 10,
    "nextToken" : "aPaginationToken",
    "lastSync" :  1550000000000,
    "filter" : {
        ...
    }
}
```

필드는 다음과 같이 정의됩니다.

Sync 필드

Sync 필드 목록

`version`

템플릿 정의 버전. 현재 `2018-05-29`만 지원됩니다. 이 값은 필수입니다.

operation

수행할 DynamoDB 작업입니다. Sync 작업을 수행하려면 이 값을 Sync으로 설정해야 합니다. 이 값은 필수입니다.

filter

반환되기 전 DynamoDB의 결과를 필터링하는 데 사용할 수 있는 필터. 필터에 대한 자세한 내용은 필터를 참조하십시오. 이 필드는 선택 사항입니다.

limit

한 번에 평가올 수 있는 최대 항목 수입니다. 이 필드는 선택 사항입니다. 생략된 경우, 기본 제한이 `100`개 항목으로 설정됩니다. 이 필드의 최대값은 `1000`개 항목입니다.

nextToken

이전 쿼리를 지속하는 페이지 매김 토큰. 이 토큰은 이전 쿼리에서 얻습니다. 이 필드는 선택 사항입니다.

lastSync

마지막으로 성공한 Sync 작업이 시작되었던 시간(epoch 밀리초)입니다. 지정된 경우, `lastSync` 이후에 변경된 항목만 반환됩니다. 이 필드는 선택 사항이며 초기 Sync 작업에서 모든 페이지를 가져온 이후에만 채워져야 합니다. 생략된 경우, 기본 테이블의 결과가 반환되고, 그렇지 않으면 델타 테이블의 결과가 반환됩니다.

basePartitionKey

Sync 작업을 수행할 때 사용되는 기본 테이블의 파티션 키입니다. 이 필드를 사용하면 테이블에서 사용자 지정 파티션 키를 활용할 때 Sync 작업을 수행할 수 있습니다. 이 필드는 선택 사항입니다.

deltaIndexName

Sync 작업에 사용되는 인덱스. 이 인덱스는 테이블에서 사용자 지정 파티션 키를 사용할 때 전체 델타 저장소 테이블에서 Sync 작업을 활성화하는 데 필요합니다. Sync 작업은 GSI(`gsi_ds_pk` 및 `gsi_ds_sk`에서 생성)에서 수행됩니다. 이 필드는 선택 사항입니다.

DynamoDB 동기화에서 반환하는 결과는 자동으로 GraphQL 및 JSON 기본 형식으로 변환되며 매핑 컨텍스트(`$context.result`)에서 사용할 수 있습니다.

DynamoDB 형식 변환에 대한 자세한 내용은 형식 시스템(응답 매핑)을 참조하세요.

응답 매핑 템플릿에 대한 자세한 내용은 해석기 매핑 템플릿 개요를 참조하세요.

결과의 구조는 다음과 같습니다.

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10,
    startedAt = 1550000000000
}
```

필드는 다음과 같이 정의됩니다.

items

동기화에서 반환하는 항목이 포함된 목록.

nextToken

더 많은 결과가 있을 수 있는 경우 nextToken에는 다른 요청에 사용할 수 있는 페이지 매김 토큰이 포함됩니다. AppSync에서는 DynamoDB에서 반환된 페이지 매김 토큰을 암호화하고 난독화합니다. 따라서 테이블 데이터가 호출자에게 실수로 유출되는 일이 없습니다. 또한 페이지 매김 토큰은 여러 해석기 간에 사용할 수 없습니다.

scannedCount

필터 표현식(있는 경우)을 적용하기 전에 DynamoDB에서 검색한 항목 수.

startedAt

로컬로 저장하고 다른 요청에 lastSync 인수로 사용할 수 있는 동기화 작업이 시작된 시간(Epoch 밀리초)입니다. 페이지 매김 토큰이 요청에 포함된 경우, 이 값은 결과의 첫 페이지에 대한 요청에 의해 반환된 값과 동일합니다.

예 1

다음 예제는 GraphQL 쿼리 `syncPosts(nextToken: String, lastSync: AWSTimestamp)`의 매핑 템플릿입니다.

이 예에서 lastSync가 생략된 경우, 기본 테이블의 모든 항목이 반환됩니다. lastSync가 제공된 경우 lastSync 이후 변경된 델타 동기화 테이블의 항목만 반환됩니다.

```
{
    "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

BatchGetItem 요청 매핑 문서를 사용하면 AWS AppSync DynamoDB 해석기에 DynamoDB에 BatchGetItem 요청을 하여 잠재적으로 여러 테이블에 걸쳐 여러 항목을 검색하도록 지시할 수 있습니다. 이 요청 템플릿에 대해 다음을 지정해야 합니다.

- 항목을 가져올 테이블 이름
- 각 테이블에서 가져올 항목의 키

DynamoDB BatchGetItem 한도가 적용되고 표현식 없음이 제공될 수 있습니다.

BatchGetItem 매핑 문서의 구조는 다음과 같습니다.

```
{
    "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" : {
                  ...
             }
        }
    }
}
```

필드는 다음과 같이 정의됩니다.

BatchGetItem 필드

BatchGetItem 필드 목록

`version`

템플릿 정의 버전. `2018-05-29`만 지원됩니다. 이 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `BatchGetItem` DynamoDB 작업을 수행하려면 이 값을 `BatchGetItem`으로 설정해야 합니다. 이 값은 필수입니다.

`tables`

항목을 가져올 DynamoDB 테이블입니다. 이 값은 테이블 이름이 맵의 키로 지정되는 맵입니다. 테이블이 하나 이상 제공되어야 합니다. 이 `tables` 값은 필수입니다.

`keys`

검색할 항목의 프라이머리 키를 나타내는 DynamoDB 키 목록입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요.

consistentRead

GetItem 작업을 실행할 때 일관된 읽기를 사용할지 여부. 이 값은 선택 사항으로, 기본값은 false입니다.

projection

DynamoDB 작업에서 반환할 속성을 지정하는 데 사용되는 프로젝션입니다. 프로젝션에 대한 자세한 내용은 프로젝션을 참조하세요. 이 필드는 선택 사항입니다.

기억해야 할 내용:

- 테이블에서 항목을 가져오지 못하면 해당 테이블의 데이터 블록에 null 요소가 나타납니다.
- 호출 결과는 요청 매핑 템플릿 내에 제공된 순서에 따라 테이블별로 정렬됩니다.
- `BatchGetItem` 내부의 각 `Get` 명령은 원자성이지만 배치는 부분적으로 처리될 수 있습니다. 오류로 인해 배치가 부분적으로 처리된 경우 처리되지 않은 키는 unprocessedKeys 블록 내에 호출 결과의 일부로 반환됩니다.
- `BatchGetItem`은 100개의 키로 제한됩니다.

다음과 같은 요청 매핑 템플릿 예제가 있습니다.

```
{
  "version": "2018-05-29",
  "operation": "BatchGetItem",
  "tables": {
    "authors": [
        {
          "author_id": {
            "S": "a1"
          }
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          }
        }
```

```
    ],
  }
}
```

`$ctx.result`에서 사용할 수 있는 호출 결과는 다음과 같습니다.

```
{
   "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`에는 오류에 대한 세부 정보가 포함됩니다. 키 data, unprocessedKeys 및 요청 매핑 템플릿에서 제공된 각 테이블 키는 호출 결과에 표시됩니다. 삭제된 항목은 데이터 블록에 나타납니다. 처리되지 않은 항목은 데이터 블록 내에서 null로 표시되고 unprocessedKeys 블록 내에 배치됩니다.

보다 완벽한 예제를 찾아보려면 자습서: DynamoDB 배치 해석기에서 AppSync 관련 DynamoDB 배치 자습서를 따라해 보세요.

BatchDeleteItem

`BatchDeleteItem` 요청 매핑 문서를 사용하면 AWS AppSync DynamoDB 해석기에 DynamoDB에 `BatchWriteItem` 요청을 하여 잠재적으로 여러 테이블에 걸쳐 여러 항목을 삭제하도록 지시할 수 있습니다. 이 요청 템플릿에 대해 다음을 지정해야 합니다.

- 항목을 삭제할 테이블 이름
- 각 테이블에서 삭제할 항목의 키

DynamoDB `BatchWriteItem` 한도가 적용되고 표현식 없음이 제공될 수 있습니다.

`BatchDeleteItem` 매핑 문서의 구조는 다음과 같습니다.

```
{
    "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
        }],
    }
}
```

필드는 다음과 같이 정의됩니다.

BatchDeleteItem 필드

BatchDeleteItem 필드 목록

`version`

템플릿 정의 버전. `2018-05-29`만 지원됩니다. 이 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `BatchDeleteItem` DynamoDB 작업을 수행하려면 이 값을 `BatchDeleteItem`으로 설정해야 합니다. 이 값은 필수입니다.

`tables`

항목을 삭제할 DynamoDB 테이블입니다. 각 테이블은 삭제할 항목의 프라이머리 키를 나타내는 DynamoDB 키 목록입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 테이블이 하나 이상 제공되어야 합니다. `tables` 값이 필요합니다.

기억해야 할 내용:

- `DeleteItem` 작업과 달리 완전히 삭제된 항목은 응답에서 반환되지 않습니다. 전달된 키만 반환됩니다.
- 테이블에서 항목을 삭제하지 못하면 해당 테이블의 데이터 블록에 null 요소가 나타납니다.
- 호출 결과는 요청 매핑 템플릿 내에 제공된 순서에 따라 테이블별로 정렬됩니다.
- `BatchDeleteItem` 내부의 각 `Delete` 명령은 원자성입니다. 그러나 배치는 부분적으로 처리될 수 있습니다. 오류로 인해 배치가 부분적으로 처리된 경우 처리되지 않은 키는 unprocessedKeys 블록 내에 호출 결과의 일부로 반환됩니다.
- `BatchDeleteItem`은 25개의 키로 제한됩니다.

다음과 같은 요청 매핑 템플릿 예제가 있습니다.

```
{
  "version": "2018-05-29",
  "operation": "BatchDeleteItem",
  "tables": {
    "authors": [
        {
          "author_id": {
```

```
            "S": "a1"
          }
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          }
        }
    ],
  }
}
```

$ctx.result에서 사용할 수 있는 호출 결과는 다음과 같습니다.

```
{
   "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`에는 오류에 대한 세부 정보가 포함됩니다. 키 data, unprocessedKeys 및 요청 매핑 템플릿에서 제공된 각 테이블 키는 호출 결과에 표시됩니다. 삭제된 항목은 데이터 블록에 나타납니다. 처리되지 않은 항목은 데이터 블록 내에서 null로 표시되고 unprocessedKeys 블록 내에 배치됩니다.

보다 완벽한 예제를 찾아보려면 자습서: DynamoDB 배치 해석기에서 AppSync 관련 DynamoDB 배치 자습서를 따라해 보세요.

BatchPutItem

`BatchPutItem` 요청 매핑 문서를 사용하면 AWS AppSync DynamoDB 해석기에 DynamoDB에 `BatchWriteItem` 요청을 하여 잠재적으로 여러 테이블에 걸쳐 여러 항목을 입력하도록 지시할 수 있습니다. 이 요청 템플릿에 대해 다음을 지정해야 합니다.

- 항목을 저장할 테이블 이름
- 각 테이블에 저장할 전체 항목

DynamoDB `BatchWriteItem` 한도가 적용되고 표현식 없음이 제공될 수 있습니다.

`BatchPutItem` 매핑 문서의 구조는 다음과 같습니다.

```
{
    "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
        }],
    }
}
```

필드는 다음과 같이 정의됩니다.

BatchPutItem 필드

BatchPutItem 필드 목록

`version`

템플릿 정의 버전. `2018-05-29`만 지원됩니다. 이 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `BatchPutItem` DynamoDB 작업을 수행하려면 이 값을 `BatchPutItem`으로 설정해야 합니다. 이 값은 필수입니다.

`tables`

항목을 넣을 DynamoDB 테이블입니다. 각 테이블 항목은 이 특정 테이블에 삽입할 DynamoDB 항목 목록을 나타냅니다. 테이블이 하나 이상 제공되어야 합니다. 이 값은 필수입니다.

기억해야 할 내용:

- 성공하면, 완전히 삽입된 항목이 응답에서 반환됩니다.
- 테이블에 항목이 삽입되지 못하면 해당 테이블의 데이터 블록에 null 요소가 표시됩니다.
- 삽입된 항목은 요청 매핑 템플릿 내에 제공된 순서에 따라 테이블별로 정렬됩니다.
- `BatchPutItem` 내부의 각 `Put` 명령은 원자성이지만 배치는 부분적으로 처리될 수 있습니다. 오류로 인해 배치가 부분적으로 처리된 경우 처리되지 않은 키는 unprocessedKeys 블록 내에 호출 결과의 일부로 반환됩니다.
- `BatchPutItem`은 25개의 항목으로 제한됩니다.

다음과 같은 요청 매핑 템플릿 예제가 있습니다.

```
{
  "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"
          }
        }
    ],
  }
}
```

$ctx.result에서 사용할 수 있는 호출 결과는 다음과 같습니다.

```
{
   "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`에는 오류에 대한 세부 정보가 포함됩니다. 키 data, unprocessedItems 및 요청 매핑 템플릿에서 제공된 각 테이블 키는 호출 결과에 표시됩니다. 삽입된 항목은 데이터 블록에 나타납니다. 처리되지 않은 항목은 데이터 블록 내에서 null로 표시되고 unprocessedItems 블록 내에 배치됩니다.

보다 완벽한 예제를 찾아보려면 자습서: DynamoDB 배치 해석기에서 AppSync 관련 DynamoDB 배치 자습서를 따라해 보세요.

TransactGetItems

`TransactGetItems` 요청 매핑 문서를 사용하면 AWS AppSync DynamoDB 해석기에 DynamoDB에 `TransactGetItems` 요청을 하여 잠재적으로 여러 테이블에 걸쳐 여러 항목을 검색하도록 지시할 수 있습니다. 이 요청 템플릿에 대해 다음을 지정해야 합니다.

- 항목을 가져올 각 요청 항목의 테이블 이름
- 각 테이블에서 가져올 각 요청 항목의 키

DynamoDB `TransactGetItems` 한도가 적용되고 표현식 없음이 제공될 수 있습니다.

`TransactGetItems` 매핑 문서의 구조는 다음과 같습니다.

```
{
    "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" : {
                 ...
            }
        }
    ]
}
```

필드는 다음과 같이 정의됩니다.

TransactGetItems 필드

TransactGetItems 필드 목록

`version`

템플릿 정의 버전. `2018-05-29`만 지원됩니다. 이 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `TransactGetItems` DynamoDB 작업을 수행하려면 이 값을 `TransactGetItems`으로 설정해야 합니다. 이 값은 필수입니다.

`transactItems`

포함할 요청 항목입니다. 이 값은 요청 항목의 배열입니다. 하나 이상의 요청 항목이 제공되어야 합니다. 이 `transactItems` 값은 필수입니다.

`table`

항목을 가져올 DynamoDB 테이블입니다. 이 값은 테이블 이름의 문자열입니다. 이 `table` 값은 필수입니다.

`key`

검색할 항목의 프라이머리 키를 나타내는 DynamoDB 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요.

`projection`

DynamoDB 작업에서 반환할 속성을 지정하는 데 사용되는 프로젝션입니다. 프로젝션에 대한 자세한 내용은 프로젝션을 참조하세요. 이 필드는 선택 사항입니다.

기억해야 할 내용:

- 트랜잭션이 성공하면 `items` 블록에서 가져오는 항목의 순서는 요청 항목의 순서와 동일합니다.
- 트랜잭션은 전부 또는 전무 방식으로 수행됩니다. 요청 항목에 오류가 발생하면 전체 트랜잭션이 수행되지 않고 오류 세부 정보가 반환됩니다.
- 가져올 수 없는 요청 항목은 오류가 아닙니다. 대신 해당 위치의 항목 블록에 null 요소가 나타납니다.
- 트랜잭션의 오류가 TransactionCanceledException인 경우 `cancellationReasons` 블록이 채워집니다. `cancellationReasons` 블록 내의 취소 사유 순서는 요청 항목의 순서와 동일합니다.
- `TransactGetItems`는 25개의 요청 항목으로 제한됩니다.

다음과 같은 요청 매핑 템플릿 예제가 있습니다.

```
{
    "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
                }
            }
        }
    ]
}
```

트랜잭션이 성공하고 첫 번째 요청된 항목만 검색되는 경우 `$ctx.result`에서 사용할 수 있는 호출 결과는 다음과 같습니다.

```
{
    "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
}
```

첫 번째 요청 항목으로 인한 TransactionCanceledException 때문에 트랜잭션이 실패할 경우 `$ctx.result`에서 사용할 수 있는 호출 결과는 다음과 같습니다.

```
{
    "items": null,
    "cancellationReasons": [
        {
            "type":"Sample error type",
            "message":"Sample error message"
        },
        {
            "type":"None",
            "message":"None"
```

```
        }
    ]
}
```

$ctx.error에는 오류에 대한 세부 정보가 포함됩니다. 키 항목 과 cancellationReasons는 $ctx.result에 있습니다.

보다 완벽한 예제를 찾아보려면 자습서: DynamoDB 트랜잭션 해석기에서 AppSync 관련 DynamoDB 트랜잭션 자습서를 따라해 보세요.

TransactWriteItems

TransactWriteItems 요청 매핑 문서를 사용하면 AWS AppSync DynamoDB 해석기에 DynamoDB에 TransactWriteItems 요청을 하여 잠재적으로 여러 테이블에 걸쳐 여러 항목을 쓰도록 지시할 수 있습니다. 이 요청 템플릿에 대해 다음을 지정해야 합니다.

- 각 요청 항목의 대상 테이블 이름
- 수행할 각 요청 항목의 작업입니다. 지원되는 작업에는 PutItem, UpdateItem, DeleteItem, ConditionCheck의 네 가지 유형이 있습니다.
- 작성할 각 요청 항목의 키

DynamoDB TransactWriteItems 제한이 적용됩니다.

TransactWriteItems 매핑 문서의 구조는 다음과 같습니다.

```
{
    "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
            }
        }
    ]
}
```

TransactWriteItems 필드

TransactWriteItems 필드 목록

필드는 다음과 같이 정의됩니다.

`version`

템플릿 정의 버전. `2018-05-29`만 지원됩니다. 이 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `TransactWriteItems` DynamoDB 작업을 수행하려면 이 값을 `TransactWriteItems`으로 설정해야 합니다. 이 값은 필수입니다.

transactItems

포함할 요청 항목입니다. 이 값은 요청 항목의 배열입니다. 하나 이상의 요청 항목이 제공되어야 합니다. 이 `transactItems` 값은 필수입니다.

`PutItem`의 필드는 다음과 같이 정의됩니다.

table

대상 DynamoDB 테이블입니다. 이 값은 테이블 이름의 문자열입니다. 이 `table` 값은 필수입니다.

operation

수행할 DynamoDB 작업입니다. `PutItem` DynamoDB 작업을 수행하려면 이 값을 `PutItem`으로 설정해야 합니다. 이 값은 필수입니다.

key

입력할 항목의 프라이머리 키를 나타내는 DynamoDB 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

attributeValues

DynamoDB에 저장할 항목의 나머지 속성. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 필드는 선택 사항입니다.

condition

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건을 지정하지 않으면 `PutItem` 요청이 해당 항목에 대한 기존 입력을 덮어씁니다. 조건 검사에 실패할 경우 기존 항목을 다시 가져올지 여부를 지정할 수 있습니다. 트랜잭션 조건에 대한 자세한 내용은 트랜잭션 조건 표현식을 참조하세요. 이 값은 선택 사항입니다.

`UpdateItem`의 필드는 다음과 같이 정의됩니다.

table

업데이트할 DynamoDB 테이블입니다. 이 값은 테이블 이름의 문자열입니다. 이 `table` 값은 필수입니다.

operation

수행할 DynamoDB 작업입니다. `UpdateItem` DynamoDB 작업을 수행하려면 이 값을 `UpdateItem`으로 설정해야 합니다. 이 값은 필수입니다.

key

업데이트할 항목의 프라이머리 키를 나타내는 DynamoDB 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

update

`update` 섹션에서는 DynamoDB의 항목 업데이트 방법을 설명하는 업데이트 표현식을 지정합니다. 업데이트 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB UpdateExpressions 문서를 참조하십시오. 이 섹션은 필수입니다.

condition

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건을 지정하지 않으면 `UpdateItem` 요청이 현재 상태와 상관 없이 기존 항목을 업데이트합니다. 조건 검사에 실패할 경우 기존 항목을 다시 가져올지 여부를 지정할 수 있습니다. 트랜잭션 조건에 대한 자세한 내용은 트랜잭션 조건 표현식을 참조하세요. 이 값은 선택 사항입니다.

`DeleteItem`의 필드는 다음과 같이 정의됩니다.

table

항목을 삭제할 DynamoDB 테이블입니다. 이 값은 테이블 이름의 문자열입니다. 이 `table` 값은 필수입니다.

operation

수행할 DynamoDB 작업입니다. `DeleteItem` DynamoDB 작업을 수행하려면 이 값을 `DeleteItem`으로 설정해야 합니다. 이 값은 필수입니다.

key

삭제할 항목의 프라이머리 키를 나타내는 DynamoDB 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

condition

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건을 지정하지 않으면 `DeleteItem` 요청이 현재 상태와 상관 없이 항목을 삭제합니다. 조건 검사에 실패할 경우 기존 항목을 다시 가져올지 여부를 지정할 수 있습니다. 트랜잭션 조건에 대한 자세한 내용은 트랜잭션 조건 표현식을 참조하세요. 이 값은 선택 사항입니다.

`ConditionCheck`의 필드는 다음과 같이 정의됩니다.

`table`

조건을 검사할 DynamoDB 테이블입니다. 이 값은 테이블 이름의 문자열입니다. 이 `table` 값은 필수입니다.

`operation`

수행할 DynamoDB 작업입니다. `ConditionCheck` DynamoDB 작업을 수행하려면 이 값을 `ConditionCheck`으로 설정해야 합니다. 이 값은 필수입니다.

`key`

조건을 확인할 항목의 프라이머리 키를 나타내는 DynamoDB 키입니다. DynamoDB 항목은 테이블 구조에 따라 단일 해시 키 또는 해시 키와 정렬 키를 가질 수 있습니다. '입력된 값'을 지정하는 방법에 대한 자세한 내용은 유형 시스템(요청 매핑)을 참조하세요. 이 값은 필수입니다.

`condition`

DynamoDB에 이미 있는 객체의 상태를 기반으로 요청의 성공 여부를 결정하는 조건. 조건 검사에 실패할 경우 기존 항목을 다시 가져올지 여부를 지정할 수 있습니다. 트랜잭션 조건에 대한 자세한 내용은 트랜잭션 조건 표현식을 참조하세요. 이 값은 필수입니다.

기억해야 할 내용:

- 성공할 경우 요청 항목의 키만 응답에 반환됩니다. 키 순서는 요청 항목의 순서와 동일합니다.
- 트랜잭션은 전부 또는 전무 방식으로 수행됩니다. 요청 항목에 오류가 발생하면 전체 트랜잭션이 수행되지 않고 오류 세부 정보가 반환됩니다.
- 두 개의 요청 항목이 동일한 항목을 대상으로 할 수 없습니다. 그렇지 않으면 TransactionCanceledException 오류가 발생합니다.
- 트랜잭션의 오류가 TransactionCanceledException인 경우 `cancellationReasons` 블록이 채워집니다. 요청 항목의 조건 검사에 실패한 동시에 `returnValuesOnConditionCheckFailure`를 `false`로 지정하지 않은 경우, 테이블에 있는 항목이 검색되고 `item`에서 `cancellationReasons` 블록의 해당 위치에 저장됩니다.
- `TransactWriteItems`는 25개의 요청 항목으로 제한됩니다.

다음과 같은 요청 매핑 템플릿 예제가 있습니다.

```
{
```

```
    "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"
                   }
               }
           },
```

```
        }
    ]
}
```

트랜잭션이 성공하면 `$ctx.result`에서 사용할 수 있는 호출 결과는 다음과 같습니다.

```
{
    "keys": [
       // Key of the PutItem request
       {
           "post_id": "p1",
       },
       // Key of the UpdateItem request
       {
           "author_id": "a1"
       }
    ],
    "cancellationReasons": null
}
```

`PutItem` 요청의 조건 검사 실패로 인해 트랜잭션이 실패할 경우 `$ctx.result`에서 사용할 수 있는 간접 호출 결과는 다음과 같습니다.

```
{
    "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"
       }
    ]
}
```

`$ctx.error`에는 오류에 대한 세부 정보가 포함됩니다. 키 및 cancellationReasons 키는 `$ctx.result`에 있습니다.

보다 완벽한 예제를 찾아보려면 자습서: DynamoDB 트랜잭션 해석기에서 AppSync 관련 DynamoDB 트랜잭션 자습서를 따라해 보세요.

형식 시스템(요청 매핑)

AWS AppSync DynamoDB 해석기를 사용하여 DynamoDB 테이블을 호출하는 경우 AWS AppSync에서는 해당 호출에 사용된 각 값의 형식을 알고 있어야 합니다. 이는 DynamoDB가 GraphQL 또는 JSON보다 더 많은 유형 기본 형식(예: 세트 및 이진 데이터)을 지원하기 때문입니다. AWS AppSync는 GraphQL과 DynamoDB 간에 변환할 때 몇 가지 힌트가 필요합니다. 그렇지 않으면 테이블에서 데이터가 구성되는 방식에 대해 몇 가지 가정을 해야 합니다.

DynamoDB 데이터 유형에 대한 자세한 내용은 DynamoDB 데이터 유형 설명자 및 데이터 유형 설명서를 참조하세요.

DynamoDB 값은 단일 키-값 페어가 포함된 JSON 객체로 나타납니다. 여기서 키는 DynamoDB 형식을 지정하고 값은 값 자체를 지정합니다. 다음 예에서 키 `S`는 값이 문자열임을 나타내고 값 `identifier`는 문자열 값 자체를 나타냅니다.

```
{ "S" : "identifier" }
```

JSON 객체는 키-값 페어를 두 개 이상 포함할 수 없습니다. 키-값 페어를 두 개 이상 지정하면 요청 매핑 문서가 구문 분석되지 않습니다.

요청 매핑 문서에서 값을 지정해야 하는 모든 곳에 DynamoDB 값을 사용할 수 있습니다. 이렇게 해야 하는 위치로는 `key` 및 `attributeValue` 섹션과 표현식 섹션 중 `expressionValues` 섹션이 있습니다. 다음 예에서는 DynamoDB 문자열 값 `identifier`를 `key` 섹션(`GetItem` 요청 매핑 문서)의 `id` 필드에 할당합니다.

```
"key" : {
   "id" : { "S" : "identifier" }
}
```

지원되는 유형

AWS AppSync는 다음 DynamoDB 스칼라, 문서 및 집합 형식을 지원합니다.

String 형식 **S**

단일 문자열 값. DynamoDB String 값은 다음과 같이 표시됩니다.

```
{ "S" : "some string" }
```

사용 예제는 다음과 같습니다.

```
"key" : {
   "id" : { "S" : "some string" }
}
```

String set 형식 **SS**

문자열 값 집합. DynamoDB String Set 값은 다음과 같이 표시됩니다.

```
{ "SS" : [ "first value", "second value", ... ] }
```

사용 예제는 다음과 같습니다.

```
"attributeValues" : {
   "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
}
```

Number 형식 **N**

단일 숫자 값. DynamoDB Number 값은 다음과 같이 표시됩니다.

```
{ "N" : 1234 }
```

사용 예제는 다음과 같습니다.

```
"expressionValues" : {
   ":expectedVersion" : { "N" : 1 }
}
```

Number set 형식 **NS**

숫자 값 집합. DynamoDB Number Set 값은 다음과 같이 표시됩니다.

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

사용 예제는 다음과 같습니다.

```
"attributeValues" : {
   "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }
}
```

Binary 형식 **B**

이진 값. DynamoDB Binary 값은 다음과 같이 표시됩니다.

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

값은 실제로 문자열이며, 여기서 문자열은 이진 데이터의 base64 인코딩 표현입니다. AWS AppSync는 이 문자열을 DynamoDB로 보내기 전에 이 문자열을 이진 값으로 다시 디코딩합니다. AWS AppSync는 RFC 2045에 정의된 base64 디코딩 체계를 사용합니다. base64 알파벳에 없는 모든 문자는 무시됩니다.

사용 예제는 다음과 같습니다.

```
"attributeValues" : {
   "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

Binary set 형식 **BS**

이진 값 집합. DynamoDB Binary Set 값은 다음과 같이 표시됩니다.

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

값은 실제로 문자열이며, 여기서 문자열은 이진 데이터의 base64 인코딩 표현입니다. AWS AppSync는 이 문자열을 DynamoDB로 보내기 전에 이 문자열을 이진 값으로 다시 디코딩합니다. AWS AppSync는 RFC 2045에 정의된 base64 디코딩 체계를 사용합니다. base64 알파벳에 없는 모든 문자는 무시됩니다.

사용 예제는 다음과 같습니다.

```
"attributeValues" : {
   "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }
```

```
}
```

Boolean 형식 **BOOL**

부울 값. DynamoDB Boolean 값은 다음과 같이 표시됩니다.

```
{ "BOOL" : true }
```

`true` 및 `false`만 유효한 값입니다.

사용 예제는 다음과 같습니다.

```
"attributeValues" : {
   "orderComplete" : { "BOOL" : false }
}
```

List 형식 **L**

기타 지원되는 DynamoDB 값 목록입니다. DynamoDB List 값은 다음과 같이 표시됩니다.

```
{ "L" : [ ... ] }
```

값은 복합 값으로, 목록에는 지원되는 DynamoDB 값(다른 목록 포함)이 0개 이상 포함될 수 있습니다. 또한 목록에는 여러 형식이 혼합되어 포함될 수 있습니다.

사용 예제는 다음과 같습니다.

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

Map 형식 **M**

지원되는 다른 DynamoDB 값의 키-값 페어로 구성된 순서 없는 모음을 나타냅니다. DynamoDB Map 값은 다음과 같이 표시됩니다.

```
{ "M" : { ... } }
```

맵에는 키-값 페어가 0개 이상 포함될 수 있습니다. 키는 문자열이어야 하며 값은 지원되는 모든 DynamoDB 값(다른 맵 포함)일 수 있습니다. 또한 맵에는 여러 형식이 혼합되어 포함될 수 있습니다.

사용 예제는 다음과 같습니다.

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

Null 형식 **NULL**

null 값. DynamoDB Null 값은 다음과 같이 표시됩니다.

```
{ "NULL" : null }
```

사용 예제는 다음과 같습니다.

```
"attributeValues" : {
   "phoneNumbers" : { "NULL" : null }
}
```

각 형식에 대한 자세한 내용은 DynamoDB 문서를 참조하십시오.

형식 시스템(응답 매핑)

DynamoDB의 응답을 수신하면 AWS AppSync에서는 수신한 응답을 GraphQL 및 JSON 기본 형식으로 자동 변환합니다. DynamoDB의 각 속성은 디코딩되어 응답 매핑 컨텍스트에서 반환됩니다.

예를 들어 DynamoDB에서 다음을 반환한 경우:

```
{
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

AWS AppSync DynamoDB 해석기에서는 반환된 값을 다음과 같이 GraphQL 및 JSON 형식으로 변환합니다.

```
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

이 단원에서는 AWS AppSync에서 다음 DynamoDB 스칼라, 문서 및 집합 형식을 변환하는 방법을 설명합니다.

String 형식 **S**

단일 문자열 값. DynamoDB String 값은 문자열로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB String 값을 반환한 경우:

```
{ "S" : "some string" }
```

AWS AppSync는 반환된 값을 문자열로 변환합니다.

```
"some string"
```

String set 형식 **SS**

문자열 값 집합. DynamoDB String Set 값은 문자열 목록으로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB String Set 값을 반환한 경우:

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync는 반환된 값을 문자열 목록으로 변환합니다.

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

Number 형식 **N**

단일 숫자 값. DynamoDB Number 값은 숫자로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB Number 값을 반환한 경우:

```
{ "N" : 1234 }
```

AWS AppSync는 반환된 값을 숫자로 변환합니다.

```
1234
```

Number set 형식 **NS**

숫자 값 집합. DynamoDB Number Set 값은 숫자 목록으로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB Number Set 값을 반환한 경우:

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync는 반환된 값을 숫자 목록으로 변환합니다.

```
[ 67.8, 12.2, 70 ]
```

Binary 형식 **B**

이진 값. DynamoDB Binary 값은 base64로 표시된 값이 포함된 문자열로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB Binary 값을 반환한 경우:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync는 반환된 값을 base64로 표시된 값이 포함된 문자열로 변환합니다.

```
"SGVsbG8sIFdvcmxkIQo="
```

이진 데이터는 RFC 4648 및 RFC 2045에 지정된 base64 인코딩 체계로 인코딩됩니다.

Binary set 형식 **BS**

이진 값 집합. DynamoDB Binary Set 값은 base64로 표시된 값이 포함된 문자열 목록으로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB Binary Set 값을 반환한 경우:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync는 반환된 값을 base64로 표시된 값이 포함된 문자열 목록으로 변환합니다.

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

이진 데이터는 RFC 4648 및 RFC 2045에 지정된 base64 인코딩 체계로 인코딩됩니다.

Boolean 형식 **BOOL**

부울 값. DynamoDB Boolean 값은 부울로 반환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB Boolean 값을 반환한 경우:

```
{ "BOOL" : true }
```

AWS AppSync는 반환된 값을 부울로 변환합니다.

```
true
```

List 형식 **L**

기타 지원되는 DynamoDB 값 목록입니다. DynamoDB List 값은 값 목록으로 반환되며, 여기서 각 내부 값 역시 변환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB List 값을 반환한 경우:

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

AWS AppSync는 반환된 값을 변환된 값 목록으로 변환합니다.

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

Map 형식 **M**

지원되는 기타 모든 DynamoDB 값의 키/값 모음입니다. DynamoDB Map 값은 JSON 객체로 반환되며, 여기서 각 키/값 역시 변환됩니다.

예를 들어 DynamoDB에서 다음 DynamoDB Map 값을 반환한 경우:

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

AWS AppSync는 반환된 값을 JSON 객체로 변환합니다.

```
{
   "someString" : "A string value",
   "someNumber" : 1,
   "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

Null 형식 **NULL**

null 값.

예를 들어 DynamoDB에서 다음 DynamoDB Null 값을 반환한 경우:

```
{ "NULL" : null }
```

AWS AppSync는 반환된 값을 null로 변환합니다.

```
null
```

Filters

`Query` 및 `Scan` 작업을 사용하여 DynamoDB에서 객체를 쿼리하는 경우, 결과를 평가해 원하는 값만 반환하는 `filter`를 선택적으로 지정할 수 있습니다.

Query 또는 Scan 매핑 문서의 필터 매핑 섹션의 구조는 다음과 같습니다.

```
"filter" : {
    "expression" : "filter expression"
    "expressionNames" : {
        "#name" : "name",
    },
    "expressionValues" : {
        ":value" : ... typed value
    },
}
```

필드는 다음과 같이 정의됩니다.

expression

쿼리 표현식. 필터 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB QueryFilter 및 DynamoDB ScanFilter 문서를 참조하십시오. 이 필드는 지정되어 있어야 합니다.

expressionNames

표현식 속성 name 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 expression에 사용된 이름 자리 표시자에 해당합니다. 이 값은 DynamoDB에서 항목의 속성 이름에 해당하는 문자열이어야 합니다. 이 필드는 선택 사항으로, expression에 사용된 표현식 속성인 name 자리 표시자의 대체 항목으로만 채워져야 합니다.

expressionValues

표현식 속성 value 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 expression에 사용되는 value 자리 표시자에 해당하고 값은 입력된 값이어야 합니다. "입력된 값"을 지정하는 방법에 대한 자세한 내용은 형식 시스템(요청 매핑)을 참조하십시오. 입력된 값은 지정되어 있어야 합니다. 이 필드는 선택 사항으로, expression에 사용된 표현식 속성인 value 자리 표시자의 대체 항목으로만 채워져야 합니다.

예

다음 예제는 매핑 템플릿의 필터 섹션으로, 여기서 DynamoDB에서 가져온 항목은 제목이 title 인수로 시작되는 경우에만 반환됩니다.

```
"filter" : {
    "expression" : "begins_with(#title, :title)",
```

```
    "expressionNames" : {
        "#title" : "title"
    },
    "expressionValues" : {
        ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
    }
}
```

조건 표현식

DynamoDB에서 `PutItem`, `UpdateItem` 및 `DeleteItem` DynamoDB 작업을 사용하여 객체를 변경하는 경우, 작업을 수행하기 전에 DynamoDB에 이미 있는 객체의 상태를 기준으로 요청에 성공할지 여부를 제어하는 조건 표현식을 선택적으로 지정할 수 있습니다.

AWS AppSync DynamoDB 해석기에서는 `PutItem`, `UpdateItem` 및 `DeleteItem` 요청 매핑 문서에 조건 표현식을 지정할 수 있고 조건에 실패해 객체가 업데이트되지 않은 경우 따를 전략을 지정할 수 있습니다.

예 1

다음 `PutItem` 매핑 문서에는 조건식은 포함되어 있지 않습니다. 따라서 키가 같은 항목이 이미 존재하더라도 DynamoDB에 항목을 저장하기 때문에 기존 항목을 덮어 씁니다.

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   }
}
```

예 2

다음 `PutItem` 매핑 문서에는 동일한 키를 가진 항목이 DynamoDB에 없는 경우에만 작업을 성공시킬 수 있는 조건 표현식이 있습니다.

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
```

```
      "id" : { "S" : "1" }
   },
   "condition" : {
      "expression" : "attribute_not_exists(id)"
   }
}
```

기본적으로 조건 확인이 실패하면 AWS AppSync DynamoDB 해석기는 GraphQL 응답의 `error` 섹션에 있는 `data` 필드에 DynamoDB 객체의 현재 값과 변형에 대한 오류를 반환합니다. 그러나 AWS AppSync DynamoDB 해석기에서는 개발자가 몇 가지 일반 엣지 케이스를 처리할 수 있도록 일부 추가 기능을 제공합니다.

- AWS AppSync DynamoDB 해석기가 DynamoDB의 현재 값이 원하는 결과와 일치하는지 확인할 수 있는 경우 작업을 성공한 것으로 간주합니다.
- 따라서 오류를 반환하는 대신 AWS AppSync DynamoDB 해석기가 실패를 처리해야 하는 방식을 결정하는 사용자 지정 Lambda 함수를 호출하도록 해석기를 구성할 수 있습니다.

이러한 내용은 조건 확인 실패 처리 단원에 자세히 설명되어 있습니다.

DynamoDB 조건 표현식에 대한 자세한 내용은 DynamoDB ConditionExpressions 문서를 참조하세요.

조건 지정

`PutItem`, `UpdateItem` 및 `DeleteItem` 요청 매핑 문서 모두에서 선택적 `condition` 섹션을 지정할 수 있습니다. 이 섹션을 지정하지 않으면 조건 검사가 수행되지 않습니다. 지정한 경우, 해당 조건을 충족해야 작업이 성공합니다.

`condition` 섹션의 구조는 다음과 같습니다.

```
"condition" : {
    "expression" : "someExpression"
    "expressionNames" : {
        "#foo" : "foo"
    },
    "expressionValues" : {
        ":bar" : ... typed value
    },
    "equalsIgnore" : [ "version" ],
    "consistentRead" : true,
    "conditionalCheckFailedHandler" : {
        "strategy" : "Custom",
```

```
        "lambdaArn" : "arn:..."
    }
}
```

조건을 지정하는 필드는 다음과 같습니다.

expression

업데이트 표현식 자체. 조건 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB ConditionExpressions 문서를 참조하십시오. 이 필드는 지정되어 있어야 합니다.

expressionNames

표현식 속성 name 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 expression에 사용된 name 자리 표시자에 해당하고 값은 DynamoDB에 있는 항목의 속성 이름에 해당하는 문자열이어야 합니다. 이 필드는 선택 사항으로, expression에 사용된 표현식 속성인 name 자리 표시자의 대체 항목으로만 채워져야 합니다.

expressionValues

표현식 속성 value 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 expression에 사용되는 value 자리 표시자에 해당하고 값은 입력된 값이어야 합니다. "입력된 값"을 지정하는 방법에 대한 자세한 내용은 형식 시스템(요청 매핑)을 참조하십시오. 입력된 값은 지정되어 있어야 합니다. 이 필드는 선택 사항으로, expression에 사용된 표현식 속성인 value 자리 표시자의 대체 항목으로만 채워져야 합니다.

나머지 필드는 AWS AppSync DynamoDB 해석기에 조건 검사 실패를 처리하는 방법을 지시합니다.

equalsIgnore

`PutItem` 작업 사용 시 조건 검사에 실패하면 AWS AppSync DynamoDB 해석기는 DynamoDB에 현재 있는 항목을 쓰려고 하는 항목과 비교합니다. 이러한 두 항목이 동일하면 해석기에서는 해당 작업을 마치 성공한 것처럼 간주합니다. `equalsIgnore` 필드를 사용하여 비교 수행 시 AWS AppSync에서 무시해야 하는 속성 목록을 지정할 수 있습니다. 예를 들어, `version` 속성만 차이가 나는 경우에는 해당 작업을 성공한 작업으로 취급합니다. 이 필드는 선택 사항입니다.

consistentRead

조건 검사에 실패하면 AWS AppSync에서는 강력히 일관된 읽기를 사용하여 DynamoDB에서 항목의 현재 값을 가져옵니다. 이 필드를 사용하여 AWS AppSync DynamoDB 해석기에 최종 읽기 일관성을 대신 사용하도록 지시할 수 있습니다. 이 필드는 선택 사항으로, 기본값은 `true`입니다.

conditionalCheckFailedHandler

이 섹션에서는 AWS AppSync DynamoDB 해석기가 DynamoDB의 현재 값을 예상 결과와 비교한 후 조건 검사를 처리하는 방법을 지정할 수 있습니다. 이 섹션은 선택 사항으로, 방법을 지정하지 않으면 `Reject` 전략이 기본값으로 지정됩니다.

strategy

AWS AppSync DynamoDB 해석기가 DynamoDB의 현재 값을 예상 결과와 비교한 후 따르는 전략. 이 필드는 필수 필드로, 다음과 같은 두 가지 값을 가질 수 있습니다.

Reject

변형에 실패하고, GraphQL 응답의 `error` 섹션에 있는 `data` 필드에 DynamoDB에 있는 객체의 현재 값과 변형에 대한 오류를 반환합니다.

Custom

AWS AppSync DynamoDB 해석기는 조건 검사 실패를 처리하는 방법을 결정하는 사용자 지정 Lambda 함수를 호출합니다. `strategy`를 `Custom`으로 설정하면 `lambdaArn` 필드에 호출할 Lambda 함수의 ARN이 포함되어 있어야 합니다.

lambdaArn

AWS AppSync DynamoDB 해석기가 조건 검사 실패를 처리하는 방법을 결정하기 위해 호출하는 Lambda 함수의 ARN. 이 필드는 `strategy`를 `Custom`으로 설정한 경우에만 지정해야 합니다. 이 기능을 사용하는 방법에 대한 자세한 내용은 조건 검사 실패 처리를 참조하십시오.

조건 검사 실패 처리

기본적으로 조건 확인이 실패하면 AWS AppSync DynamoDB 해석기는 GraphQL 응답의 `error` 섹션에 있는 `data` 필드에 DynamoDB 객체의 현재 값과 변형에 대한 오류를 반환합니다. 그러나 AWS AppSync DynamoDB 해석기에서는 개발자가 몇 가지 일반 엣지 케이스를 처리할 수 있도록 일부 추가 기능을 제공합니다.

- AWS AppSync DynamoDB 해석기가 DynamoDB의 현재 값이 원하는 결과와 일치하는지 확인할 수 있는 경우 작업을 성공한 것으로 간주합니다.
- 따라서 오류를 반환하는 대신 AWS AppSync DynamoDB 해석기가 실패를 처리해야 하는 방식을 결정하는 사용자 지정 Lambda 함수를 호출하도록 해석기를 구성할 수 있습니다.

이 프로세스의 흐름 차트입니다.

원하는 결과 확인

조건 검사에 실패하면 AWS AppSync DynamoDB 해석기는 `GetItem` DynamoDB 요청을 수행하여 DynamoDB에서 항목의 현재 값을 가져옵니다. 기본적으로 이 해석기는 강력히 일관된 읽기를 사용하지만, `condition` 블록의 `consistentRead` 필드를 사용하여 예상 결과와 비교하도록 구성할 수 있습니다.

- `PutItem` 작업의 경우 AWS AppSync DynamoDB 해석기는 현재 값을 쓰려는 값과 비교하여 비교 결과에서 `equalsIgnore`에 나열된 모든 속성을 제외합니다. 항목이 동일한 경우 해당 작업을 성공한 작업으로 간주하고 DynamoDB에서 가져온 항목을 반환합니다. 그렇지 않으면 구성된 전략을 따릅니다.

  예를 들어, `PutItem` 요청 매핑 문서가 다음과 같은 경우:

```
{
   "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" ]
   }
}
```

  그리고 DynamoDB에 현재 있는 항목이 다음과 같은 경우:

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
```

```
}
```

AWS AppSync DynamoDB 해석기는 쓰려는 항목을 현재 값과 비교해 `version` 필드만 차이가 난다는 사실을 파악합니다. 그러나 `version` 필드를 무시하도록 구성되어 있기 때문에 해당 작업을 성공으로 간주하고 DynamoDB에서 가져온 항목을 반환합니다.

- `DeleteItem` 작업의 경우 AWS AppSync DynamoDB 해석기는 항목이 DynamoDB에서 반환되었는지 확인합니다. 반환되는 항목이 없는 경우 해당 작업을 성공으로 간주합니다. 그렇지 않으면 구성된 전략을 따릅니다.
- `UpdateItem` 작업의 경우 AWS AppSync DynamoDB 해석기는 현재 DynamoDB에 있는 항목이 예상 결과와 일치하는지 확인하는데 필요한 충분한 정보가 없기 때문에 구성된 전략을 따릅니다.

DynamoDB에 있는 객체의 현재 상태가 예상 결과와 다르면 AWS AppSync DynamoDB 해석기는 구성된 전략에 따라 변경을 거부하거나 다음에 수행할 작업을 결정하기 위해 Lambda 함수를 호출합니다.

'Reject' 전략 따르기

`Reject` 전략을 따르는 경우 AWS AppSync DynamoDB 해석기는 변형에 대한 오류를 반환하고 GraphQL 응답의 `error` 섹션에 있는 `data` 필드에 DynamoDB에 있는 객체의 현재 값을 반환합니다. DynamoDB에서 반환된 항목은 클라이언트에서 기대하는 형식으로 항목을 변환하는 응답 매핑 템플릿을 통해 입력되며 선택 세트를 기준으로 필터링됩니다.

예를 들어 다음과 같은 변형 요청이 있습니다.

```
mutation {
    updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
        Name
        theVersion
    }
}
```

DynamoDB에서 반환된 항목이 다음과 같은 경우:

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

그리고 응답 매핑 템플릿이 다음과 같은 경우:

```
{
   "id" : $util.toJson($context.result.id),
   "Name" : $util.toJson($context.result.name),
   "theVersion" : $util.toJson($context.result.version)
}
```

GraphQL 응답은 다음과 같습니다.

```
{
  "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
      },
      ...
    }
  ]
}
```

반환된 객체의 필드를 다른 해석기가 채우고 변형에 성공했는데 이 객체가 `error` 섹션에서 반환되면 반환된 객체의 필드가 확인되지 않습니다.

'Custom' 전략 따르기

`Custom` 전략을 따르면 AWS AppSync DynamoDB 해석기는 다음에 수행해야 할 작업을 결정하는 Lambda 함수를 호출합니다. Lambda 함수는 다음 옵션 중 하나를 선택합니다.

- 변형 `reject`. 이 옵션은 AWS AppSync DynamoDB 해석기에 구성된 전략이 `Reject`인 것처럼 동작하도록 지시해 이전의 단원에서 설명한 것처럼 DynamoDB에 있는 객체의 현재 값과 변형에 대한 오류를 반환하도록 합니다.
- 변형 `discard`. 이 옵션은 AWS AppSync DynamoDB 해석기에 조건 검사 실패를 자동으로 무시하고 DynamoDB의 값을 반환하도록 지시합니다.
- 변형 `retry`. 이 옵션은 AWS AppSync DynamoDB 해석기에 새 요청 매핑 문서를 사용하여 변형을 다시 시도하도록 지시합니다.

Lambda 호출 요청

AWS AppSync DynamoDB 해석기가 `lambdaArn`에 지정된 Lambda 함수를 호출합니다. 데이터 원본에 대해 구성된 동일한 `service-role-arn`을 사용합니다. 호출의 페이로드 구조는 다음과 같습니다.

```
{
    "arguments": { ... },
    "requestMapping": {... },
    "currentValue": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

필드는 다음과 같이 정의됩니다.

`arguments`

GraphQL 변형의 인수. `$context.arguments`에서 요청 매핑 문서에 사용할 수 있는 인수와 동일합니다.

`requestMapping`

작업에 대한 요청 매핑 문서

`currentValue`

DynamoDB에 있는 객체의 현재 값

`resolver`

AWS AppSync 해석기에 대한 정보입니다.

`identity`

호출자에 대한 정보. `$context.identity`에서 요청 매핑 문서에 사용할 수 있는 자격 증명 정보와 동일합니다.

페이로드 전체를 보여주는 예:

```
{
    "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"
    }
}
```

Lambda 호출 응답

Lambda 함수는 호출 페이로드를 검사해 AWS AppSync DynamoDB 해석기에서 실패를 처리해야 하는 방법을 결정하는 비즈니스 로직을 어떠한 것이든 적용할 수 있습니다. 조건 검사 실패를 처리하기 위한 옵션에는 다음 3가지가 있습니다.

- 변형 `reject`. 이 옵션에 대한 응답 페이로드의 구조는 다음과 같아야 합니다.

```
{
    "action": "reject"
}
```

  이 옵션은 AWS AppSync DynamoDB 해석기에 구성된 전략이 `Reject`인 것처럼 동작하도록 지시해 위의 섹션에서 설명한 것처럼 DynamoDB에 있는 객체의 현재 값과 변형에 대한 오류를 반환하도록 합니다.

- 변형 `discard`. 이 옵션에 대한 응답 페이로드의 구조는 다음과 같아야 합니다.

```
{
    "action": "discard"
}
```

  이 옵션은 AWS AppSync DynamoDB 해석기에 조건 검사 실패를 자동으로 무시하고 DynamoDB의 값을 반환하도록 지시합니다.

- 변형 `retry`. 이 옵션에 대한 응답 페이로드의 구조는 다음과 같아야 합니다.

```
{
    "action": "retry",
    "retryMapping": { ... }
}
```

  이 옵션은 AWS AppSync DynamoDB 해석기에 새 요청 매핑 문서를 사용하여 변형을 다시 시도하도록 지시합니다. `retryMapping` 섹션의 구조는 DynamoDB 작업에 따라 달라지며 해당 작업에 대한 전체 요청 매핑 문서의 하위 집합입니다.

  `PutItem`의 경우 `retryMapping` 섹션의 구조는 다음과 같습니다. `attributeValues` 필드에 대한 설명은 PutItem을 참조하십시오.

```
{
    "attributeValues": { ... },
    "condition": {
        "equalsIgnore" = [ ... ],
        "consistentRead" = true
    }
}
```

UpdateItem의 경우 retryMapping 섹션의 구조는 다음과 같습니다. update 섹션에 대한 설명은 UpdateItem을 참조하십시오.

```
{
    "update" : {
        "expression" : "someExpression"
        "expressionNames" : {
            "#foo" : "foo"
        },
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "condition": {
        "consistentRead" = true
    }
}
```

DeleteItem의 경우 retryMapping 섹션의 구조는 다음과 같습니다.

```
{
    "condition": {
        "consistentRead" = true
    }
}
```

작동할 수 있는 다른 작업 또는 키를 지정할 수 있는 방법은 없습니다. AWS AppSync DynamoDB 해석기는 동일한 객체에 대해 동일한 작업만 재시도하도록 허용합니다. condition 섹션에서는 conditionalCheckFailedHandler를 지정할 수 없습니다. 재시도에 실패하면 AWS AppSync DynamoDB 해석기는 Reject 전략을 따릅니다.

다음은 실패한 PutItem 요청을 처리하는 Lambda 함수의 예입니다. 이 비즈니스 로직은 호출한 사람이 누구인지 확인합니다. jeffTheAdmin이 호출한 경우 요청을 재시도해 현재 DynamoDB에 있는 항목에서 version 및 expectedVersion을 업데이트합니다. 그렇지 않으면 변형을 거부합니다.

```
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)
};
```

트랜잭션 조건 표현식

트랜잭션 조건식은 `TransactWriteItems`에서 네 가지 작업 유형 모두(`PutItem`, `DeleteItem`, `UpdateItem`, `ConditionCheck`)의 요청 매핑 템플릿에 사용할 수 있습니다.

`PutItem`, `DeleteItem`, `UpdateItem`의 경우 트랜잭션 조건 표현식은 선택 사항입니다. `ConditionCheck`의 경우 트랜잭션 조건 표현식이 필수입니다.

예 1

다음 트랜잭션 `DeleteItem` 매핑 문서에는 조건식이 없습니다. 따라서 DynamoDB에서 항목을 삭제합니다.

```
{
   "version": "2018-05-29",
```

```
   "operation": "TransactWriteItems",
   "transactItems": [
      {
         "table": "posts",
         "operation": "DeleteItem",
         "key": {
            "id": { "S" : "1" }
         }
      }
   ]
}
```

예 2

다음 트랜잭션 DeleteItem 매핑 문서에는 해당 게시물의 작성자가 특정 이름과 같은 경우에만 작업이 성공할 수 있는 트랜잭션 조건식이 있습니다.

```
{
   "version": "2018-05-29",
   "operation": "TransactWriteItems",
   "transactItems": [
      {
         "table": "posts",
         "operation": "DeleteItem",
         "key": {
            "id": { "S" : "1" }
         }
         "condition": {
            "expression": "author = :author",
            "expressionValues": {
               ":author": { "S" : "Chunyan" }
            }
         }
      }
   ]
}
```

조건 검사에 실패하면 TransactionCanceledException이 발생하고, $ctx.result.cancellationReasons에 오류 세부 정보가 반환됩니다. 기본적으로, 조건 검사가 실패하게 된 원인인 DynamoDB의 이전 항목이 $ctx.result.cancellationReasons에 반환됩니다.

조건 지정

PutItem, UpdateItem 및 DeleteItem 요청 매핑 문서 모두에서 선택적 condition 섹션을 지정할 수 있습니다. 이 섹션을 지정하지 않으면 조건 검사가 수행되지 않습니다. 지정한 경우, 해당 조건을 충족해야 작업이 성공합니다. ConditionCheck에는 지정할 condition 섹션이 있어야 합니다. 전체 트랜잭션이 성공하려면 조건이 true여야 합니다.

condition 섹션의 구조는 다음과 같습니다.

```
"condition": {
    "expression": "someExpression",
    "expressionNames": {
        "#foo": "foo"
    },
    "expressionValues": {
        ":bar": ... typed value
    },
    "returnValuesOnConditionCheckFailure": false
}
```

조건을 지정하는 필드는 다음과 같습니다.

expression

업데이트 표현식 자체. 조건 표현식을 작성하는 방법에 대한 자세한 내용은 DynamoDB ConditionExpressions 문서를 참조하십시오. 이 필드는 지정되어 있어야 합니다.

expressionNames

표현식 속성 name 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 expression에 사용된 name 자리 표시자에 해당하고 값은 DynamoDB에 있는 항목의 속성 이름에 해당하는 문자열이어야 합니다. 이 필드는 선택 사항으로, expression에 사용된 표현식 속성인 name 자리 표시자의 대체 항목으로만 채워져야 합니다.

expressionValues

표현식 속성 value 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 expression에 사용되는 value 자리 표시자에 해당하고 값은 입력된 값이어야 합니다. "입력된 값"을 지정하는 방법에 대한 자세한 내용은 형식 시스템(요청 매핑)을 참조하십시오. 입력된 값은 지정되어 있어야 합니다. 이 필드는 선택 사항으로, expression에 사용된 표현식 속성인 value 자리 표시자의 대체 항목으로만 채워져야 합니다.

`returnValuesOnConditionCheckFailure`

조건 검사에 실패할 경우 DynamoDB의 항목을 다시 가져올지 여부를 지정합니다. 가져온 항목은 `$ctx.result.cancellationReasons[$index].item`에 있습니다. 여기서 `$index`는 조건 검사에 실패한 요청 항목의 인덱스입니다. 기본값은 true입니다.

프로젝션

`GetItem`, `Scan`, `Query`, `BatchGetItem` 및 `TransactGetItems` 작업을 사용하여 DynamoDB에서 객체를 읽을 때 원하는 속성을 식별하는 프로젝션을 선택적으로 지정할 수 있습니다. 프로젝션의 구조는 필터와 비슷하며 다음과 같습니다.

```
"projection" : {
    "expression" : "projection expression"
    "expressionNames" : {
        "#name" : "name",
    }
}
```

필드는 다음과 같이 정의됩니다.

`expression`

프로젝션 표현식은 문자열입니다. 단일 속성을 가져오려면 속성의 이름을 지정합니다. 여러 속성의 경우 이름은 쉼표로 구분된 값이어야 합니다. 프로젝션 표현식 작성에 대한 자세한 내용은 DynamoDB 프로젝션 표현식 설명서를 참조하세요. 이 필드는 필수 항목입니다.

`expressionNames`

표현식 속성 name 자리 표시자의 대체 항목으로, 키-값 페어의 형식으로 표시됩니다. 키는 `expression`에 사용된 이름 자리 표시자에 해당합니다. 이 값은 DynamoDB에서 항목의 속성 이름에 해당하는 문자열이어야 합니다. 이 필드는 선택 사항으로, `expression`에 사용된 표현식 속성인 name 자리 표시자의 대체 항목으로만 채워져야 합니다. `expressionNames`에 대한 자세한 내용은 DynamoDB 설명서를 참조하세요.

예 1

다음 예는 DynamoDB에서 `author` 및 `id` 속성만 반환되는 VTL 매핑 템플릿의 프로젝션 섹션입니다.

```
"projection" : {
```

```
    "expression" : "#author, id",
    "expressionNames" : {
        "#author" : "author"
    }
}
```

> **Tip**
>
> $context.info.SelectionSetList를 사용하여 GraphQL 요청 설정에 액세스할 수 있습니다. 이 필드를 사용하면 요구 사항에 따라 프로젝션 표현식을 동적으로 프레이밍할 수 있습니다.

> **Note**
>
> `Query` 및 `Scan` 작업과 함께 프로젝션 표현식을 사용하는 경우 `select`의 값은 반드시 `SPECIFIC_ATTRIBUTES`이어야 합니다. 자세한 내용은 DynamoDB 설명서를 참조하세요.

RDS에 대한 해석기 매핑 템플릿 참조

AWS AppSync RDS 해석기 매핑 템플릿을 사용해 개발자는 Amazon Aurora Serverless를 위한 데이터 API로 SQL 쿼리를 전송하고 이러한 쿼리의 결과를 다시 가져올 수 있습니다.

요청 매핑 템플릿

RDS 요청 매핑 템플릿은 꽤 간단합니다.

```
{
    "version": "2018-05-29",
    "statements": [],
    "variableMap": {},
    "variableTypeHintMap": {}
}
```

다음은 해석되고 난 RDS 요청 매핑 템플릿의 JSON 스키마 표현입니다.

```
{
    "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"
        }
    }
}
```

다음은 정적 쿼리를 사용하는 요청 매핑 템플릿의 예입니다.

```
{
    "version": "2018-05-29",
    "statements": [
        "select title, isbn13 from BOOKS where author = 'Mark Twain'"
    ]
}
```

버전

모든 요청 매핑 템플릿에 공통적으로 적용되는 버전 필드는 템플릿이 사용하는 버전을 정의합니다. 버전 필드는 필수입니다. 값 '2018-05-29'는 Amazon RDS 매핑 템플릿에 지원되는 유일한 버전입니다.

```
"version": "2018-05-29"
```

명령문 및 VariableMap

명령문 배열은 개발자 제공 쿼리를 위한 자리 표시자입니다. 현재, 요청 매핑 템플릿당 쿼리를 최대 2개까지 지원합니다. variableMap은 SQL 문을 더 간단하고 쉽게 읽을 수 있도록 만드는 데 사용할 수 있는 별칭을 포함하고 있는 선택적 필드입니다. 예를 들어, 가능한 예제는 다음과 같습니다.

```
{
"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)
    }
}
```

는 변수 맵 값을 사용하여 Amazon Aurora Serverless Data API로 전송될 SqlParameterized 쿼리를 구성합니다. SQL 명령문은 변수 맵에 제공된 파라미터를 사용하여 실행되므로 SQL 삽입 위험이 없습니다.

VariableTypeHintMap

`variableTypeHintMap`은 SQL 파라미터 유형 힌트를 보내는 데 사용할 수 있는 별칭 유형을 포함하는 선택적 필드입니다. 이러한 유형 힌트는 SQL 명령문에서의 명시적 캐스팅을 방지하므로 길이가 짧아집니다. 예를 들어, 가능한 예제는 다음과 같습니다.

```
{
    "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 AppSync는 이 변수 맵 값을 사용해 Amazon Aurora Serverless Data API로 전송되는 쿼리를 구성합니다. 또한 `variableTypeHintMap` 데이터를 사용하고 유형 정보를 RDS로 보냅니다. RDS가 지원되는 `typeHints`는 여기에서 확인할 수 있습니다.

OpenSearch에 대한 해석기 매핑 템플릿 참조입니다.

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

Amazon OpenSearch Service용 AWS AppSync 해석기에서는 GraphQL을 사용하여 계정에 있는 기존 OpenSearch Service 도메인의 데이터를 저장 및 검색할 수 있습니다. 이 해석기는 수신되는 GraphQL

요청을 OpenSearch Service 요청으로 매핑한 다음 OpenSearch Service 응답을 다시 GraphQL로 매핑하도록 하여 작동합니다. 이 섹션에서는 지원되는 OpenSearch Service 작업에 대한 매핑 템플릿에 대해 설명합니다.

요청 매핑 템플릿

대부분의 OpenSearch Service 요청 매핑 템플릿에는 몇 부분만 변경되는 공통 구조가 있습니다. 다음 예제에서는 OpenSearch Service 도메인을 대상으로 검색을 실행합니다. 이 도메인에서는 `post`라는 인덱스로 문서가 구성되어 있습니다. 검색 파라미터는 `body` 섹션에 정의되어 있고, 공통 쿼리 절은 대부분 `query` 필드에서 정의합니다. 이 예에서는 문서의 `"Nadia"` 필드에서 `"Bailey"`, `author` 또는 둘 다가 포함된 문서를 검색합니다.

```
{
    "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" }}
                    ]
                }
            }
        }
    }
}
```

응답 매핑 템플릿

다른 데이터 소스와 마찬가지로, OpenSearch Service는 GraphQL로 변환해야 하는 응답을 AWS AppSync에 전송합니다.

대부분의 GraphQL 쿼리는 OpenSearch Service 응답에서 `_source` 필드를 검색합니다. 개별 문서 또는 문서 목록을 반환하도록 검색을 수행할 수 있기 때문에 OpenSearch Service에서는 두 가지 공통 응답 매핑 템플릿이 사용됩니다.

결과 목록

```
[
    #foreach($entry in $context.result.hits.hits)
      #if( $velocityCount > 1 ) , #end
        $utils.toJson($entry.get("_source"))
    #end
]
```

개별 항목

```
$utils.toJson($context.result.get("_source"))
```

`operation` field

(요청 매핑 템플릿만)

AWS AppSync에서 OpenSearch Service 도메인에 보내는 HTTP 메서드 또는 동사(GET, POST, PUT, HEAD 또는 DELETE). 키와 값은 둘 다 문자열이어야 합니다.

```
"operation" : "PUT"
```

`path` field

(요청 매핑 템플릿만)

AWS AppSync에서 보낸 OpenSearch Service 요청에 대한 검색 경로입니다. 이 경로는 작업의 HTTP 동사에 대한 URL을 구성합니다. 키와 값은 둘 다 문자열이어야 합니다.

```
"path" : "/<indexname>/_doc/<_id>"
"path" : "/<indexname>/_doc"
"path" : "/<indexname>/_search"
"path" : "/<indexname>/_update/<_id>
```

매핑 템플릿이 평가되면 이 경로는 OpenSearch Service 도메인을 포함해 HTTP 요청의 일부로 전송됩니다. 예를 들어, 이전 예제는 다음과 같이 변환될 수 있습니다.

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

params field

(요청 매핑 템플릿만)

검색에서 수행할 작업을 지정하는 데 사용되며, 가장 일반적으로 본문 내에 쿼리 값을 설정하여 지정합니다. 그러나 응답 형식 지정 등과 같이 구성할 수 있는 기타 여러 가지 기능이 있습니다.

- headers

  키-값 페어로 표시되는 헤더 정보. 키와 값은 둘 다 문자열이어야 합니다. 예제:

  ```
  "headers" : {
      "Content-Type" : "application/json"
  }
  ```

  > **Note**
  >
  > AWS AppSync는 현재 JSON만 `Content-Type`으로 지원합니다.

- queryString

  JSON 응답에 대한 코드 형식 지정 등과 같이 공통 옵션을 지정하는 키-값 페어. 키와 값은 둘 다 문자열이어야 합니다. 예를 들어, 형식이 잘 지정된 JSON을 가져오려면 다음을 사용합니다.

  ```
  "queryString" : {
      "pretty" : "true"
  }
  ```

- body

  요청의 주요 부분으로, AWS AppSync에서 OpenSearch Service 도메인에 대해 잘 구성된 검색 요청을 작성하도록 합니다. 키는 객체로 구성된 문자열이어야 합니다. 아래에 몇 가지 경우가 나와 있습니다.

예 1

"시애틀"과 일치하는 도시가 포함된 모든 문서를 반환합니다.

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "match" : {
            "city" : "seattle"
        }
    }
}
```

예제 2

도시 또는 주가 "워싱턴"과 일치하는 모든 문서를 반환합니다.

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "multi_match" : {
            "query" : "washington",
            "fields" : ["city", "state"]
        }
    }
}
```

변수 전달

(요청 매핑 템플릿만)

또한 VTL 문에서 평가의 일부로 변수를 전달할 수도 있습니다. 예를 들어, 다음과 같은 GraphQL 쿼리가 있다고 가정해 보겠습니다.

```
query {
    searchForState(state: "washington"){
        ...
    }
}
```

이 매핑 템플릿은 인수로 주를 받을 수 있습니다.

```
"body":{
    "from":0,
```

```
    "size":50,
    "query" : {
        "multi_match" : {
            "query" : "$context.arguments.state",
            "fields" : ["city", "state"]
        }
    }
}
```

VTL에 포함할 수 있는 유틸리티 목록은 요청 헤더에 액세스를 참조하십시오.

Lambda용 해석기 매핑 템플릿 참조

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

함수와 해석기를 사용하여 계정에 있는 Lambda AWS AppSync 함수를 호출할 수 있습니다. Lambda 함수를 클라이언트에 반환하기 전에 요청 페이로드와 Lambda 함수의 응답을 조정할 수 있습니다. 또한 매핑 템플릿을 사용하여 호출할 작업의 특성에 AWS AppSync 대한 힌트를 제공할 수 있습니다. 이 단원에서는 지원되는 Lambda 작업에 대한 다양한 매핑 템플릿에 대해 설명합니다.

요청 매핑 템플릿

Lambda 요청 매핑 템플릿은 Lambda 함수와 관련된 필드를 처리합니다.

```
{
  "version": string,
  "operation": Invoke|BatchInvoke,
  "payload": any type,
  "invocationType": RequestResponse|Event
}
```

다음은 Lambda 요청 매핑 템플릿이 해결되었을 때의 JSON 스키마 표현입니다.

```
{
  "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
}
```

다음은 GraphQL 스키마의 getPost 필드로 페이로드 데이터를 컨텍스트의 인수와 함께 사용하는 invoke 작업을 사용하는 예제입니다.

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $util.toJson($context.arguments)
  }
}
```

전체 매핑 문서가 Lambda 함수에 입력으로 전달되므로 이전 예제는 이제 다음과 같습니다.

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "id": "postId1"
    }
  }
}
```

버전

모든 요청 매핑 템플릿에 공통으로, 템플릿이 사용하는 버전을 version 정의합니다. version는 필수이며 정적 값입니다.

```
"version": "2018-05-29"
```

Operation

Lambda 데이터 소스를 사용하면 필드에 두 가지 작업 (및) 을 정의할 수 있습니다operation. Invoke BatchInvoke 이 Invoke 작업은 모든 GraphQL 필드 리졸버에 대해 Lambda 함수를 호출하도록 AWS AppSync 알려줍니다. BatchInvoke현재 GraphQL 필드에 대한 요청을 일괄 AWS AppSync 처리하도록 지시합니다. operation 필드는 필수 사항입니다.

의 경우Invoke, 해결된 요청 매핑 템플릿은 Lambda 함수의 입력 페이로드와 일치합니다. 위 예제를 수정해 보겠습니다.

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
    "payload": {
      "arguments": $util.toJson($context.arguments)
    }
}
```

이 문제는 해결되어 Lambda 함수로 전달되며, 다음과 같이 보일 수 있습니다.

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
    "payload": {
      "arguments": {
        "id": "postId1"
      }
    }
}
```

의 경우BatchInvoke, 매핑 템플릿은 배치의 모든 필드 리졸버에 적용됩니다. 간결하게 정리하기 위해 모든 확인된 매핑 템플릿 payload 값을 매핑 템플릿과 일치하는 단일 객체 아래의 목록으로 AWS AppSync 병합합니다. 다음 예제 템플릿을 이러한 병합을 보여줍니다.

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": $util.toJson($context)
}
```

이 템플릿은 다음 매핑 문서로 해석됩니다.

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

payload목록의 각 요소는 단일 배치 항목에 해당합니다. 또한 Lambda 함수는 요청에 전송된 항목의 순서와 일치하는 목록 형태의 응답을 반환할 것으로 예상됩니다.

```
[
  { "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
]
```

페이로드

payload필드는 올바른 형식의 JSON을 Lambda 함수로 전달하는 데 사용되는 컨테이너입니다. operation필드가 로 설정된 경우 기존 payload 값을 BatchInvoke 목록으로 AWS AppSync 래핑합니다. payload 필드는 선택 사항입니다.

호출 유형

Lambda 데이터 소스를 사용하면 두 가지 호출 유형인 및 을 정의할 수 있습니다. RequestResponse Event 호출 유형은 Lambda API에 정의된 호출 유형과 동일합니다. RequestResponse호출 유형을 사용하면 AWS AppSync Lambda 함수를 동기적으로 호출하여 응답을 기다릴 수 있습니다. Event호출을 통해 Lambda 함수를 비동기적으로 호출할 수 있습니다. Lambda가 호출 유형 요청을 Event 처리하는 방법에 대한 자세한 내용은 비동기 호출을 참조하십시오. invocationType 필드는 선택 사항입니다. 이 필드가 요청에 포함되지 않은 경우 기본적으로 호출 유형이 사용됩니다. AWS AppSync RequestResponse

모든 invocationType 필드에 대해 해결된 요청은 Lambda 함수의 입력 페이로드와 일치합니다. 위 예제를 수정해 보겠습니다.

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event"
  "payload": {
    "arguments": $util.toJson($context.arguments)
  }
}
```

이 문제는 해결되어 Lambda 함수로 전달되며, 다음과 같이 보일 수 있습니다.

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

BatchInvoke작업이 Event 호출 유형 필드와 함께 사용될 경우 위에서 언급한 것과 동일한 방식으로 필드 확인자를 AWS AppSync 병합하고, 요청은 값 목록이 되는 비동기 이벤트로 Lambda 함수에 전달됩니다. payload 캐시 적중이 발생할 경우 Lambda로 전송되지 않으므로 Event 호출 유형 확인자에 대해서는 확인자 캐싱을 비활성화하는 것이 좋습니다.

응답 매핑 템플릿

다른 데이터 소스와 마찬가지로 Lambda 함수는 GraphQL 유형으로 변환되어야 AWS AppSync 하는 응답을 보냅니다.

Lambda 함수의 결과가 Velocity Template Language(VTL) context 속성을 통해 사용할 수 있는 $context.result 객체에 설정됩니다.

Lambda 함수 응답의 모양이 GraphQL 유형의 모양과 정확히 일치하는 경우 다음 응답 매핑 템플릿을 사용하여 응답을 전달할 수 있습니다.

```
$util.toJson($context.result)
```

응답 매핑 템플릿에 적용되는 필수 필드 또는 모양 제한이 없습니다. 하지만 GraphQL은 강력한 형식이므로 해석된 매핑 템플릿이 필요한 GraphQL 유형과 일치해야 합니다.

Lambda 함수 일괄 처리 응답

operation필드가 로 BatchInvoke 설정된 경우 Lambda 함수에서 항목 목록이 반환될 AWS AppSync 것으로 예상됩니다. 각 결과를 원래 요청 항목에 다시 AWS AppSync 매핑하려면 응답 목록의 크기와 순서가 일치해야 합니다. 응답 목록에 null 항목이 있어도 유효하며 이에 따라 null로 *$ctx.result* 설정됩니다.

Direct Lambda 해석기

매핑 템플릿을 완전히 사용하지 않으려면 Lambda 함수에 기본 페이로드를 제공하고 GraphQL 유형에 대한 기본 Lambda 함수 응답을 제공할 AWS AppSync 수 있습니다. 요청 템플릿을 제공할지, 응답 템플릿을 제공하지 않을지 선택할 수 있으며 그에 따라 처리할 수 있습니다. AWS AppSync

Direct Lambda 요청 매핑 템플릿

요청 매핑 템플릿이 제공되지 않은 경우 Lambda 함수에 작업으로서 `Context` 객체를 직접 전송합니다. AWS AppSync `Invoke` `Context` 객체의 구조에 대한 자세한 내용은 해석기 매핑 템플릿 컨텍스트 참조 단원을 참조하세요.

Direct Lambda 응답 매핑 템플릿

응답 매핑 템플릿이 제공되지 않은 경우, Lambda 함수의 응답을 수신하면 두 가지 작업 중 하나를 AWS AppSync 수행합니다. 요청 매핑 템플릿을 제공하지 않았거나 해당 `2018-05-29` 버전과 함께 요청 매핑 템플릿을 제공한 경우 응답은 다음 응답 매핑 템플릿과 동일합니다.

```
#if($ctx.error)
     $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
 #end
 $util.toJson($ctx.result)
```

`2017-02-28`버전과 함께 템플릿을 제공한 경우 응답 로직은 다음 응답 매핑 템플릿과 동일하게 작동합니다.

```
$util.toJson($ctx.result)
```

표면적으로 보면 매핑 템플릿 바이패스는 이전 예와 같이 특정 매핑 템플릿을 사용하는 것과 비슷하게 작동합니다. 그러나 내부에서는 매핑 템플릿에 대한 평가가 완전히 우회됩니다. 템플릿 평가 단계를 우회하기 때문에 일부 시나리오에서는 평가가 필요한 응답 매핑 템플릿이 있는 Lambda 함수에 비해 애플리케이션에서 응답 중에 발생하는 오버헤드와 지연 시간이 줄어들 수 있습니다.

Direct Lambda 해석기 응답에서의 사용자 지정 오류 처리

사용자 지정 예외를 발생시켜 Direct Lambda 해석기가 호출하는 Lambda 함수의 오류 응답을 사용자 지정할 수 있습니다. 다음 예제는 다음을 사용하여 사용자 지정 예외를 생성하는 방법을 보여줍니다. JavaScript

```
class CustomException extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomException";
  }
}

throw new CustomException("Custom message");
```

예외가 발생하는 경우 `errorType` 및 `errorMessage`는 각각 발생한 사용자 지정 오류의 `name` 및 `message`입니다.

`errorType`인 경우`UnauthorizedException`, 사용자 지정 메시지 대신 기본 메시지 (`"You are not authorized to make this call."`) 를 AWS AppSync 반환합니다.

다음 스니펫은 사용자 정의를 보여주는 GraphQL 응답의 예시입니다. `errorType`

```
{
  "data": {
    "query": null
  },
  "errors": [
    {
      "path": [
        "query"
      ],
      "data": null,
      "errorType": "CustomException",
      "errorInfo": null,
      "locations": [
        {
          "line": 5,
          "column": 10,
          "sourceName": null
        }
      ],
      "message": "Custom Message"
    }
  ]
}
```

Direct Lambda 해석기: 일괄 처리 활성화

해석기에 `maxBatchSize`를 구성하여 Direct Lambda 해석기에 대한 일괄 처리를 활성화할 수 있습니다. `maxBatchSize`가 Direct Lambda AWS AppSync 해석기보다 큰 값으로 설정된 `0` 경우 최대 크기의 Lambda 함수에 요청을 일괄 전송합니다. `maxBatchSize`

Direct Lambda 리졸버를 `maxBatchSize` `0` 켜기로 설정하면 일괄 처리가 해제됩니다.

Lambda 해석기를 사용한 일괄 처리 작동 방식에 대한 자세한 내용은 고급 사용 사례: 일괄 처리 단원을 참조하세요.

요청 매핑 템플릿

일괄 처리가 활성화되고 요청 매핑 템플릿이 제공되지 않은 경우, `Context` 객체 목록을 `BatchInvoke` 작업으로 Lambda 함수에 직접 AWS AppSync 전송합니다.

응답 매핑 템플릿

일괄 처리가 활성화되고 응답 매핑 템플릿이 제공되지 않은 경우, 응답 로직은 다음 응답 매핑 템플릿과 동일합니다.

```
#if( $context.result && $context.result.errorMessage )
      $utils.error($context.result.errorMessage, $context.result.errorType,
      $context.result.data)
#else
      $utils.toJson($context.result.data)
#end
```

Lambda 함수는 전송된 `Context` 객체 목록과 동일한 순서로 결과 목록을 반환해야 합니다. 특정 결과에 대해 `errorMessage` 및 `errorType`을 제공하여 개별 오류를 반환할 수 있습니다. 목록에서 각 결과는 다음 형식이어야 합니다.

```
{
   "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**
>
> 결과 객체의 다른 필드는 현재 무시됩니다.

Lambda의 오류 처리

Lambda 함수에서 예외 또는 오류를 발생시켜 모든 결과에 대해 오류를 반환할 수 있습니다. 배치 요청의 페이로드 요청 또는 응답 크기가 너무 크면 Lambda는 오류를 반환합니다. 이 경우 `maxBatchSize`를 줄이거나 응답 페이로드의 크기를 줄이는 것을 고려해야 합니다.

개별 오류 처리에 대한 자세한 내용은 개별 오류 반환을 참조하세요.

샘플 Lambda 함수

아래 스키마를 사용하여 필드 리졸버용 Direct Lambda 리졸버를 생성하고 위에서 `Post.relatedPosts` 설정하여 일괄 처리를 활성화할 수 있습니다. `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]
```

```
}
```

다음 쿼리에서는 `relatedPosts` 해결 요청을 일괄 처리하여 Lambda 함수를 호출합니다.

```
query getAllPosts {
  allPosts {
    id
    relatedPosts {
      id
    }
  }
}
```

Lambda 함수의 간단한 구현은 다음과 같습니다.

```
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' }
    }
  })
}
```

에 대한 리졸버 매핑 템플릿 참조 EventBridge

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

EventBridge 데이터 소스와 함께 사용되는 AWS AppSync 리졸버 매핑 템플릿을 사용하면 Amazon EventBridge 버스에 사용자 지정 이벤트를 보낼 수 있습니다.

요청 매핑 템플릿

`PutEvents`요청 매핑 템플릿을 사용하면 여러 사용자 지정 이벤트를 이벤트 버스로 보낼 수 있습니다 EventBridge. 매핑 문서의 구조는 다음과 같습니다.

```
{
    "version" : "2018-05-29",
    "operation" : "PutEvents",
    "events" : [{}]
}
```

다음은 요청 매핑 템플릿의 EventBridge 예입니다.

```
{
    "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"
    }

    ]
}
```

응답 매핑 템플릿

PutEvents작업이 성공하면 응답 EventBridge 양식이 다음에 포함됩니다$ctx.result.

```
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end
  $util.toJson($ctx.result)
```

InternalExceptions 또는 Timeouts와 같은 PutEvents 작업을 수행하는 동안 발생하는 오류는 $ctx.error에 표시됩니다. EventBridge의 일반적인 오류 목록은 일반 오류 참조를 EventBridge 참조하십시오.

result는 다음 형식을 따릅니다.

```
{
    "Entries" [
        {
            "ErrorCode" : String,
            "ErrorMessage" : String,
            "EventId" : String
        }
    ],
    "FailedEntryCount" : number
}
```

- 항목

  수집된 이벤트 결과(성공 및 실패 모두) 수집이 성공하면 항목에 EventID가 포함됩니다. 그렇지 않으면 ErrorCode 및 ErrorMessage를 사용하여 항목의 문제를 식별할 수 있습니다.

  각 레코드의 응답 요소의 인덱스는 요청 배열의 인덱스와 동일합니다.

- FailedEntryCount

  실패한 항목 수입니다. 이 값은 정수로 표시됩니다.

의 응답에 대한 자세한 내용은 PutEvents 을 참조하십시오 PutEvents.

예제 샘플 응답 1

다음 예는 두 개의 성공적인 이벤트가 있는 PutEvents 작업입니다.

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ],
    "FailedEntryCount" : 0
}
```

예제 샘플 응답 2

다음 예는 3개의 이벤트(성공 2개, 실패 1개)가 있는 PutEvents 작업입니다.

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        },
        {
            "ErrorCode" : "SampleErrorCode",
            "ErrorMessage" : "Sample Error Message"
        }
    ],
    "FailedEntryCount" : 1
}
```

`PutEvents` field

- 버전

  모든 요청 매핑 템플릿에 공통적으로 적용되는 `version` 필드는 템플릿이 사용하는 버전을 정의합니다. 이 필드는 필수 항목입니다. `2018-05-29`값은 EventBridge 매핑 템플릿에 지원되는 유일한 버전입니다.

- Operation

  지원되는 유일한 작업은 `PutEvents`입니다. 이 작업을 통해 이벤트 버스에 사용자 지정 이벤트를 추가할 수 있습니다.

- 이벤트

  이벤트 버스에 추가될 이벤트 배열입니다. 이 배열에는 1~10개의 항목이 할당되어야 합니다.

  `Event` 객체는 다음 필드를 포함하는 유효한 JSON 객체입니다.

  - `"source"`: 이벤트 소스를 정의하는 문자열입니다.
  - `"detail"`: 이벤트에 대한 정보를 첨부하는 데 사용할 수 있는 JSON 객체입니다. 이 필드는 빈 맵(`{ }`)일 수 있습니다.
  - `"detailType`: 스토리지의 유형을 식별하는 문자열입니다.
  - `"resources"`: 이벤트와 관련된 리소스를 식별하는 문자열의 JSON 배열입니다. 이 필드는 빈 배열일 수 있습니다.
  - `"time"`: 이벤트 타임스탬프는 문자열로 제공됩니다. 이는 RFC3339 타임스탬프 형식을 따라야 합니다.

다음은 유효한 `Event` 객체의 몇 가지 예시입니다.

예 1

```
{
    "source" : "source1",
    "detail" : {
        "key1" : [1,2,3,4],
        "key2" : "strval"
    },
    "detailType" : "sampleDetailType",
    "resources" : ["Resouce1", "Resource2"],
    "time" : "2022-01-10T05:00:10Z"
```

```
}
```

예제 2

```
{
    "source" : "source1",
    "detail" : {},
    "detailType" : "sampleDetailType"
}
```

예 3

```
{
    "source" : "source1",
    "detail" : {
        "key1" : 1200
    },
    "detailType" : "sampleDetailType",
    "resources" : []
}
```

None 데이터 소스에 대한 해석기 매핑 템플릿 참조

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

None 유형의 데이터 소스와 함께 사용되는 AWS AppSync 해석기 매핑 템플릿을 사용하면 AWS AppSync 로컬 작업 요청의 형태를 만들 수 있습니다.

요청 매핑 템플릿

매핑 템플릿은 단순하며 `payload` 필드를 통해 가능한 많은 컨텍스트 정보를 전달할 수 있습니다.

```
{
   "version": string,
   "payload": any type
```

```
}
```

다음은 해석되고 난 요청 매핑 템플릿의 JSON 스키마 표현입니다.

```
{
    "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
}
```

다음은 VTL 컨텍스트 속성 `$context.arguments`를 통한 필드 인수 전달의 예입니다.

```
{
    "version": "2018-05-29",
    "payload": $util.toJson($context.arguments)
}
```

`payload` 필드의 값이 응답 매핑 템플릿으로 전달되어 VTL 컨텍스트 속성(`$context.result`)에서 사용할 수 있습니다.

다음은 `payload` 필드의 보간 값을 나타내는 예제입니다.

```
{
    "id": "postId1"
}
```

버전

모든 요청 매핑 템플릿에 공통된 `version` 필드는 템플릿에 사용되는 버전을 정의합니다.

`version` 필드는 필수입니다.

예제:

```
"version": "2018-05-29"
```

Payload

`payload` 필드는 올바른 형식의 JSON을 응답 매핑 템플릿으로 전달하는 데 사용할 수 있는 컨테이너입니다.

`payload` 필드는 선택 사항입니다.

응답 매핑 템플릿

데이터 원본이 없으므로 `payload` 필드의 값이 응답 매핑 템플릿으로 전달되고 VTL `context` 속성을 통해 제공되는 `$context.result` 객체에서 설정됩니다.

`payload` 필드 값의 모양이 GraphQL 유형의 모양과 정확히 일치하는 경우 다음 응답 매핑 템플릿을 사용하여 응답을 전달할 수 있습니다.

```
$util.toJson($context.result)
```

응답 매핑 템플릿에 적용되는 필수 필드 또는 모양 제한이 없습니다. 하지만 GraphQL은 강력한 형식이므로 해석된 매핑 템플릿이 필요한 GraphQL 유형과 일치해야 합니다.

HTTP에 대한 해석기 매핑 템플릿 참조

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

AWS AppSync HTTP 해석기 매핑 템플릿을 사용하면 요청을 AWS AppSync 에서 HTTP 엔드포인트로 전송하고 HTTP 엔드포인트의 응답을 다시 AWS AppSync로 전송할 수 있습니다. 매핑 템플릿을 사

용하면 호출할 작업의 특성에 대해 AWS AppSync 에 알려줄 수도 있습니다. 이 단원에서는 지원되는 HTTP 해석기에 대한 다양한 매핑 템플릿에 대해 설명합니다.

요청 매핑 템플릿

```
{
    "version": "2018-05-29",
    "method": "PUT|POST|GET|DELETE|PATCH",
    "params": {
        "query": Map,
        "headers": Map,
        "body": any
    },
    "resourcePath": string
}
```

HTTP 요청 매핑 템플릿을 해석한 후 요청 매핑 템플릿의 JSON 스키마 표현은 아래와 같습니다.

```
{
    "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
    "type": "object",
    "properties": {
        "version": {
        "$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"
```

```
    ]
}
```

다음은 text/plain 본문을 포함한 HTTP POST 요청의 예입니다.

```
{
    "version": "2018-05-29",
    "method": "POST",
    "params": {
        "headers":{
        "Content-Type":"text/plain"
        },
        "body":"this is an example of text body"
    },
    "resourcePath": "/"
}
```

버전

요청 매핑 템플릿만 해당

템플릿에 사용되는 버전을 정의합니다. version은 모든 요청 매핑 템플릿에 공통되며 필수입니다.

```
"version": "2018-05-29"
```

메서드

요청 매핑 템플릿만 해당

AWS AppSync 에서 HTTP 엔드포인트에 보내는 HTTP 메서드 또는 동사(GET, POST, PUT, PATCH 또는 DELETE).

```
"method": "PUT"
```

ResourcePath

요청 매핑 템플릿만 해당

액세스하고자 하는 리소스 경로입니다. 리소스 경로는 HTTP 데이터 원본의 엔드포인트와 함께 AWS AppSync 서비스에서 요청을 수행하는 URL을 형성합니다.

```
"resourcePath": "/v1/users"
```

매핑 템플릿이 평가되면 이 경로는 HTTP 엔드포인트를 포함해 HTTP 요청의 일부로 전송됩니다. 예를 들어, 이전 예제는 다음과 같이 변환될 수 있습니다.

```
PUT <endpoint>/v1/users
```

Params Field

요청 매핑 템플릿만 해당

검색에서 수행할 작업을 지정하는 데 사용되며, 가장 일반적으로 본문 내에 쿼리 값을 설정하여 지정합니다. 그러나 응답 형식 지정 등과 같이 구성할 수 있는 기타 여러 가지 기능이 있습니다.

headers

키-값 페어로 표시되는 헤더 정보. 키와 값은 둘 다 문자열이어야 합니다.

예:

```
"headers" : {
    "Content-Type" : "application/json"
}
```

현재 지원되는 `Content-Type` 헤더는 다음과 같습니다.

```
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
```

참고: 다음 HTTP 헤더를 설정할 수 없습니다.

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
```

```
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

JSON 응답에 대한 코드 형식 지정 등과 같이 공통 옵션을 지정하는 키-값 페어. 키와 값은 둘 다 문자열이어야 합니다. 다음 예제는 쿼리 문자열을 `?type=json`으로 보낼 방법을 보여줍니다.

```
"query" : {
    "type" : "json"
}
```

body

본문에는 설정을 선택하는 HTTP 요청 본문이 포함됩니다. 콘텐츠 유형이 charset을 지정하지 않는 한 요청 본문은 항상 UTF-8 인코딩된 문자열입니다.

```
"body":"body string"
```

HTTPS 엔드포인트에 대해 AWS AppSync가 인정하는 인증 기관(CA)

> **Note**
>
> Let's Encrypt는 identrust 및 isrgrootx1 인증서를 통해 승인됩니다. Let's Encrypt를 사용하는 경우 별도의 조치가 필요하지 않습니다.

현재 HTTPS를 사용하는 경우 HTTP 해석기는 자체 서명된 인증서를 지원하지 않습니다. AWS AppSync는 HTTPS용 SSL/TLS 인증서를 해석할 때 다음과 같은 인증 기관을 인식합니다.

AWS AppSync의 알려진 루트 인증서

이름	날짜	SHA1 지문
digicertassuredidrootca	2018년 4월 21일	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
trustcenterclass2caii	2018년 4월 21일	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E

이름	날짜	SHA1 지문
thawtepremiumserverca	2018년 4월 21일	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	2016년 11월 23일	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
swisssignplatinumg2ca	2018년 4월 21일	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66
swisssignsilverg2ca	2018년 4월 21일	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
thawteserverca	2018년 4월 21일	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
equifaxsecureebusinessca1	2018년 4월 21일	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
securetrustca	2018년 4월 21일	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
utnuserfirstclientauthemailca	2018년 4월 21일	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
thawtepersonalfreemailca	2018년 4월 21일	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
affirmtrustnetworkingca	2018년 4월 21일	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
entrustevca	2018년 4월 21일	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
utnuserfirsthardwareca	2018년 4월 21일	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
certumca	2018년 4월 21일	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18

이름	날짜	SHA1 지문
addtrustclass1ca	2018년 4월 21일	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
entrustrootcag2	2018년 4월 21일	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
equifaxsecureca	2018년 4월 21일	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
quovadisrootca3	2018년 4월 21일	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	2018년 4월 21일	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
digicertglobalrootg2	2018년 4월 21일	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
digicerthighassurancceevrootca	2018년 4월 21일	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
secomvalicertclass1ca	2018년 4월 21일	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
equifaxsecureglobalebusinessca1	2018년 4월 21일	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
geotrustuniversalca	2018년 4월 21일	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
deprecateditsecca	2012년 1월 27일	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
verisignclass3ca	2018년 4월 21일	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
thawteprimaryrootcag3	2018년 4월 21일	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2

이름	날짜	SHA1 지문
thawteprimaryrootcag2	2018년 4월 21일	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
deutschetelekomroottca2	2018년 4월 21일	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
buypassclass3ca	2018년 4월 21일	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
utnuserfirstobjectca	2018년 4월 21일	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46
geotrustprimaryca	2018년 4월 21일	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
buypassclass2ca	2018년 4월 21일	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
baltimorecodesigningca	2018년 4월 21일	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D
verisignclass1ca	2018년 4월 21일	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1
baltimorecybertrustca	2018년 4월 21일	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
starfieldclass2ca	2018년 4월 21일	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
camerfirmachamberscommerceca	2018년 4월 21일	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
ttelesecglobalrootclass3ca	2018년 4월 21일	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
verisignclass3g5ca	2018년 4월 21일	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5

이름	날짜	SHA1 지문
ttelesecglobalrootclass2ca	2018년 4월 21일	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
trustcenteruniversalcai	2018년 4월 21일	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
verisignclass3g4ca	2018년 4월 21일	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3g3ca	2018년 4월 21일	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
xrampglobalca	2018년 4월 21일	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
amzninternalrootca	2008년 12월 12일	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06
certplusclass3pprimaryca	2018년 4월 21일	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
certumtrustednetworkca	2018년 4월 21일	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
verisignclass3g2ca	2018년 4월 21일	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
globalsignr3ca	2018년 4월 21일	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
utndatacorpsgcca	2018년 4월 21일	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
secomscrootca2	2018년 4월 21일	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
gtecybertrustgloballca	2018년 4월 21일	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74

이름	날짜	SHA1 지문
secomscrootca1	2018년 4월 21일	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
affirmtrustcommercialca	2018년 4월 21일	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
trustcenterclass4caii	2018년 4월 21일	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
verisignuniversalrootca	2018년 4월 21일	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
globalsignr2ca	2018년 4월 21일	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
certplusclass2primaryca	2018년 4월 21일	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
digicertglobalrootca	2018년 4월 21일	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
globalsignca	2018년 4월 21일	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
thawteprimaryrootca	2018년 4월 21일	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
starfieldrootg2ca	2018년 4월 21일	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
geotrustglobalca	2018년 4월 21일	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
soneraclass2ca	2018년 4월 21일	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
verisigntsaca	2018년 4월 21일	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01

이름	날짜	SHA1 지문
soneraclass1ca	2018년 4월 21일	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
quovadisrootca	2018년 4월 21일	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
affirmtrustpremiumeccca	2018년 4월 21일	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
starfieldservicesrootg2ca	2018년 4월 21일	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
valicertclass2ca	2018년 4월 21일	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
comodoaaaca	2018년 4월 21일	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
aolrootca2	2018년 4월 21일	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
keynectisrootca	2018년 4월 21일	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
addtrustqualifiedca	2018년 4월 21일	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
aolrootca1	2018년 4월 21일	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignclass2g3ca	2018년 4월 21일	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
addtrustexternalca	2018년 4월 21일	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
verisignclass2g2ca	2018년 4월 21일	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D

이름	날짜	SHA1 지문
geotrustprimarycag3	2018년 4월 21일	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycag2	2018년 4월 21일	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
swisssigngoldg2ca	2018년 4월 21일	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
entrust2048ca	2018년 4월 21일	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
chunghwaepkirootca	2018년 4월 21일	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
camerfirmachambersignca	2018년 4월 21일	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
camerfirmachambersca	2018년 4월 21일	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
godaddyclass2ca	2018년 4월 21일	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
affirmtrustpremiumca	2018년 4월 21일	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
verisignclass1g3ca	2018년 4월 21일	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
secomevrootca1	2018년 4월 21일	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
verisignclass1g2ca	2018년 4월 21일	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
amzninternalinfoseccag3	2015년 2월 27일	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6

이름	날짜	SHA1 지문
cia-crt-g3-01-ca	2016년 11월 23일	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
godaddyrootg2ca	2018년 4월 21일	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
digicertassuredidrootca	2018년 4월 21일	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
microseceszignorootca2009	2018년 4월 21일	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
affirmtrustcommercial	2018년 4월 21일	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
comodoecccertificationauthority	2018년 4월 21일	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
cadisigrootr2	2018년 4월 21일	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
swisssignsilvercag2	2018년 4월 21일	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
securetrustca	2018년 4월 21일	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
cadisigrootr1	2018년 4월 21일	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
accvraiz1	2018년 4월 21일	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
entrustrootcertificationauthority	2018년 4월 21일	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
camerfirmaglobalchambersignroot	2018년 4월 21일	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9

이름	날짜	SHA1 지문
dstacescax6	2018년 4월 21일	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
identrustpublicsectorrootca1	2018년 4월 21일	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
starfieldrootcertificateauthorityg2	2018년 4월 21일	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
secureglobalca	2018년 4월 21일	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
eecertificationcentrerootca	2018년 4월 21일	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
opentrustrootcag3	2018년 4월 21일	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8A:A6
teliasonerarootcav1	2018년 4월 21일	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
autoridaddecertificacionfirmaprofesionalcifa62634068	2018년 4월 21일	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
opentrustrootcag2	2018년 4월 21일	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	2018년 4월 21일	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:D5:7E
globalsigneccrootcar5	2018년 4월 21일	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
globalsigneccrootcar4	2018년 4월 21일	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB

이름	날짜	SHA1 지문
izenpecom	2018년 4월 21일	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
turktrustelektronikserti fikahizmetsaglayicisih5	2018년 4월 21일	C4:18:F6:4D:46:D1:DF:00:3D:27:30:13:72:43:A9:12:11:C6:75:FB
gdcatrustauthr5root	2018년 4월 21일	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
dtrustrootclass3ca22009	2018년 4월 21일	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
quovadisrootca3	2018년 4월 21일	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	2018년 4월 21일	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
geotrustprimarycertificationauthorityg3	2018년 4월 21일	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycertificationauthorityg2	2018년 4월 21일	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
oistewisekeyglobalrootgbca	2018년 4월 21일	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
addtrustexternalroot	2018년 4월 21일	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
chambersofcommerceroot2008	2018년 4월 21일	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C

이름	날짜	SHA1 지문
digicertglobalroot g3	2018년 4월 21일	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3: 3F:FA:D9:3B:E8:3D:34:9E
comodoaaaservicesr oot	2018년 4월 21일	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2: F1:F1:60:17:64:D8:E3:49
digicertglobalroot g2	2018년 4월 21일	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73: FE:06:D1:CC:8D:4F:82:A4
certinomisrootca	2018년 4월 21일	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C: 01:B9:32:C5:34:E7:88:A8
oistewisekeyglobal rootgaca	2018년 4월 21일	59:22:A1:E1:5A:EA:16:35:21:F8:98:39: 6A:46:46:B0:44:1B:0F:A9
dstrootcax3	2018년 4월 21일	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1: 73:26:38:CA:6A:D7:7C:13
certigna	2018년 4월 21일	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68: 37:58:2D:C4:AC:FD:94:97
digicerthighassura nceevrootca	2018년 4월 21일	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A: E6:D3:8F:1A:61:C7:DC:25
soneraclass2rootca	2018년 4월 21일	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A: B7:41:10:B4:F2:E4:9A:27
trustcorrootcertca 2	2018년 4월 21일	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA: 4E:06:34:C7:94:B2:1C:C0
usertrustrsacertif icationauthority	2018년 4월 21일	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51: F7:0E:E9:0D:DA:B9:AD:8E
trustcorrootcertca 1	2018년 4월 21일	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86: 5C:CA:A8:3A:45:5B:C3:0A
geotrustuniversalc a	2018년 4월 21일	E6:21:F3:35:43:79:05:9A:4B:68:30:9D: 8A:2F:74:22:15:87:EC:79

이름	날짜	SHA1 지문
certsignrootca	2018년 4월 21일	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
amazonrootca4	2018년 4월 21일	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
amazonrootca3	2018년 4월 21일	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
amazonrootca2	2018년 4월 21일	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
verisignuniversalrootcertificationauthority	2018년 4월 21일	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
amazonrootca1	2018년 4월 21일	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
networksolutionscertificateauthority	2018년 4월 21일	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
thawteprimaryrootcag3	2018년 4월 21일	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
affirmtrustnetworking	2018년 4월 21일	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
thawteprimaryrootcag2	2018년 4월 21일	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
trustcoreca1	2018년 4월 21일	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
deutschetelekomrootca2	2018년 4월 21일	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF

이름	날짜	SHA1 지문
godaddyrootcertificateauthorityg2	2018년 4월 21일	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
entrustrootcertificationauthorityec1	2018년 4월 21일	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
szafirrootca2	2018년 4월 21일	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE
tubitakkamusmsslkoksertifikasisurum1	2018년 4월 21일	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
buypassclass3rootca	2018년 4월 21일	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
comodorsacertificationauthority	2018년 4월 21일	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
netlockaranyclassgoldfotanusitvany	2018년 4월 21일	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
securitycommunicationrootca2	2018년 4월 21일	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
dtrustrootclass3ca2ev2009	2018년 4월 21일	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
starfieldclass2ca	2018년 4월 21일	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
pscprocert	2018년 4월 21일	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
actalisauthenticationrootca	2018년 4월 21일	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC

이름	날짜	SHA1 지문
staatdernederlandenrootcag3	2018년 4월 21일	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
cfcaevroot	2018년 4월 21일	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
digicerttrustedrootg4	2018년 4월 21일	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
staatdernederlandenrootcag2	2018년 4월 21일	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
securitycommunicationevrootca1	2018년 4월 21일	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
globalsignrootcar3	2018년 4월 21일	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
globalsignrootcar2	2018년 4월 21일	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
certumtrustednetworkca2	2018년 4월 21일	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
acraizfnmtrcm	2018년 4월 21일	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
hellenicacademicandresearchinstitutionseccrootca2015	2018년 4월 21일	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
certplusrootcag2	2018년 4월 21일	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:CC:1A
twcarootcertificationauthority	2018년 4월 21일	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48

이름	날짜	SHA1 지문
twcaglobalrootca	2018년 4월 21일	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
certplusrootcag1	2018년 4월 21일	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:F7:66
geotrustuniversalca2	2018년 4월 21일	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
baltimorecybertrustroot	2018년 4월 21일	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
buypassclass2rootca	2018년 4월 21일	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
certumtrustednetworkca	2018년 4월 21일	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
digicertassuredidrootg3	2018년 4월 21일	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
digicertassuredidrootg2	2018년 4월 21일	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F
isrgrootx1	2018년 4월 21일	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
entrustnetpremium2048secureserverca	2018년 4월 21일	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
certplusclass2primaryca	2018년 4월 21일	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
digicertglobalrootca	2018년 4월 21일	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
entrustrootcertificationauthorityg2	2018년 4월 21일	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4

이름	날짜	SHA1 지문
starfieldservicesrootcertificateauthorityg2	2018년 4월 21일	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
thawteprimaryrootca	2018년 4월 21일	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
atostrustedroot2011	2018년 4월 21일	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
geotrustglobalca	2018년 4월 21일	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
luxtrustglobalroot2	2018년 4월 21일	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5F:3F
etugracertificationauthority	2018년 4월 21일	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
visaecommerceroot	2018년 4월 21일	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
quovadisrootca	2018년 4월 21일	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
identrustcommercialrootca1	2018년 4월 21일	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
staatdernederlandenevrootca	2018년 4월 21일	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
ttelesecglobalrootclass3	2018년 4월 21일	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
ttelesecglobalrootclass2	2018년 4월 21일	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9

이름	날짜	SHA1 지문
comodocertificationauthority	2018년 4월 21일	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
securitycommunicationrootca	2018년 4월 21일	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
quovadisrootca3g3	2018년 4월 21일	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D
xrampglobalcaroot	2018년 4월 21일	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
securesignrootca11	2018년 4월 21일	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
affirmtrustpremium	2018년 4월 21일	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
globalsignrootca	2018년 4월 21일	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
swisssigngoldcag2	2018년 4월 21일	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
quovadisrootca2g3	2018년 4월 21일	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
affirmtrustpremiumecc	2018년 4월 21일	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
geotrustprimarycertificationauthority	2018년 4월 21일	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
quovadisrootca1g3	2018년 4월 21일	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67

이름	날짜	SHA1 지문
hongkongpostrootca1	2018년 4월 21일	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
usertrustecccertificationauthority	2018년 4월 21일	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0
cybertrustglobalroot	2018년 4월 21일	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
godaddyclass2ca	2018년 4월 21일	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
hellenicacademicandresearchinstitutionsrootca2015	2018년 4월 21일	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
ecacc	2018년 4월 21일	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:F2:E8
hellenicacademicandresearchinstitutionsrootca2011	2018년 4월 21일	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
verisignclass3publicprimarycertificationauthorityg5	2018년 4월 21일	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
verisignclass3publicprimarycertificationauthorityg4	2018년 4월 21일	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A

이름	날짜	SHA1 지문
verisignclass3publicprimarycertificationauthorityg3	2018년 4월 21일	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
trustisfpsrootca	2018년 4월 21일	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
epkirootcertificationauthority	2018년 4월 21일	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
globalchambersignroot2008	2018년 4월 21일	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
camerfirmachambersofcommerceroot	2018년 4월 21일	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert81.pem	2014년 3월 13일	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
mozillacert99.pem	2014년 3월 13일	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
mozillacert145.pem	2014년 3월 13일	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:3A:04
mozillacert37.pem	2014년 3월 13일	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert4.pem	2014년 3월 13일	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B
mozillacert70.pem	2014년 3월 13일	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
mozillacert88.pem	2014년 3월 13일	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D

이름	날짜	SHA1 지문
mozillacert134.pem	2014년 3월 13일	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
mozillacert26.pem	2014년 3월 13일	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
mozillacert77.pem	2014년 3월 13일	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
mozillacert123.pem	2014년 3월 13일	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10:DD:6B:DF:99:72:2C:96:E5
mozillacert15.pem	2014년 3월 13일	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
mozillacert66.pem	2014년 3월 13일	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34
mozillacert112.pem	2014년 3월 13일	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
mozillacert55.pem	2014년 3월 13일	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
mozillacert101.pem	2014년 3월 13일	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
mozillacert119.pem	2014년 3월 13일	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
mozillacert44.pem	2014년 3월 13일	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
mozillacert108.pem	2014년 3월 13일	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
mozillacert95.pem	2014년 3월 13일	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57

이름	날짜	SHA1 지문
mozillacert141.pem	2014년 3월 13일	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
mozillacert33.pem	2014년 3월 13일	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
mozillacert0.pem	2014년 3월 13일	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
mozillacert84.pem	2014년 3월 13일	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2
mozillacert130.pem	2014년 3월 13일	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert148.pem	2014년 3월 13일	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
mozillacert22.pem	2014년 3월 13일	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
mozillacert7.pem	2014년 3월 13일	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
mozillacert73.pem	2014년 3월 13일	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
mozillacert137.pem	2014년 3월 13일	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A1:D1
mozillacert11.pem	2014년 3월 13일	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
mozillacert29.pem	2014년 3월 13일	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
mozillacert62.pem	2014년 3월 13일	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B

이름	날짜	SHA1 지문
mozillacert126.pem	2014년 3월 13일	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
mozillacert18.pem	2014년 3월 13일	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
mozillacert51.pem	2014년 3월 13일	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
mozillacert69.pem	2014년 3월 13일	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
mozillacert115.pem	2014년 3월 13일	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert40.pem	2014년 3월 13일	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
mozillacert58.pem	2014년 3월 13일	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
mozillacert104.pem	2014년 3월 13일	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E
mozillacert91.pem	2014년 3월 13일	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
mozillacert47.pem	2014년 3월 13일	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
mozillacert80.pem	2014년 3월 13일	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert98.pem	2014년 3월 13일	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
mozillacert144.pem	2014년 3월 13일	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27

이름	날짜	SHA1 지문
mozillacert36.pem	2014년 3월 13일	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D
mozillacert3.pem	2014년 3월 13일	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
mozillacert87.pem	2014년 3월 13일	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert133.pem	2014년 3월 13일	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
mozillacert25.pem	2014년 3월 13일	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
mozillacert76.pem	2014년 3월 13일	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert122.pem	2014년 3월 13일	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
mozillacert14.pem	2014년 3월 13일	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
mozillacert65.pem	2014년 3월 13일	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:35:FB
mozillacert111.pem	2014년 3월 13일	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
mozillacert129.pem	2014년 3월 13일	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
mozillacert54.pem	2014년 3월 13일	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
mozillacert100.pem	2014년 3월 13일	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0

이름	날짜	SHA1 지문
mozillacert118.pem	2014년 3월 13일	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45
mozillacert151.pem	2014년 3월 13일	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
mozillacert43.pem	2014년 3월 13일	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
mozillacert107.pem	2014년 3월 13일	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
mozillacert94.pem	2014년 3월 13일	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
mozillacert140.pem	2014년 3월 13일	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert32.pem	2014년 3월 13일	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C
mozillacert83.pem	2014년 3월 13일	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:EA:46
mozillacert147.pem	2014년 3월 13일	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
mozillacert21.pem	2014년 3월 13일	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
mozillacert39.pem	2014년 3월 13일	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
mozillacert6.pem	2014년 3월 13일	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
mozillacert72.pem	2014년 3월 13일	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B

이름	날짜	SHA1 지문
mozillacert136.pem	2014년 3월 13일	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert10.pem	2014년 3월 13일	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert28.pem	2014년 3월 13일	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
mozillacert61.pem	2014년 3월 13일	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
mozillacert79.pem	2014년 3월 13일	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert125.pem	2014년 3월 13일	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert17.pem	2014년 3월 13일	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert50.pem	2014년 3월 13일	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
mozillacert68.pem	2014년 3월 13일	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
mozillacert114.pem	2014년 3월 13일	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert57.pem	2014년 3월 13일	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
mozillacert103.pem	2014년 3월 13일	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert90.pem	2014년 3월 13일	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC

이름	날짜	SHA1 지문
mozillacert46.pem	2014년 3월 13일	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
mozillacert97.pem	2014년 3월 13일	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
mozillacert143.pem	2014년 3월 13일	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
mozillacert35.pem	2014년 3월 13일	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
mozillacert2.pem	2014년 3월 13일	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
mozillacert86.pem	2014년 3월 13일	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
mozillacert132.pem	2014년 3월 13일	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert24.pem	2014년 3월 13일	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
mozillacert9.pem	2014년 3월 13일	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
mozillacert75.pem	2014년 3월 13일	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
mozillacert121.pem	2014년 3월 13일	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
mozillacert139.pem	2014년 3월 13일	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert13.pem	2014년 3월 13일	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91

이름	날짜	SHA1 지문
mozillacert64.pem	2014년 3월 13일	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
mozillacert110.pem	2014년 3월 13일	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert128.pem	2014년 3월 13일	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D
mozillacert53.pem	2014년 3월 13일	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
mozillacert117.pem	2014년 3월 13일	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
mozillacert150.pem	2014년 3월 13일	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
mozillacert42.pem	2014년 3월 13일	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
mozillacert106.pem	2014년 3월 13일	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
mozillacert93.pem	2014년 3월 13일	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17
mozillacert31.pem	2014년 3월 13일	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
mozillacert49.pem	2014년 3월 13일	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71
mozillacert82.pem	2014년 3월 13일	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
mozillacert146.pem	2014년 3월 13일	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A

이름	날짜	SHA1 지문
mozillacert20.pem	2014년 3월 13일	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert38.pem	2014년 3월 13일	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36
mozillacert5.pem	2014년 3월 13일	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
mozillacert71.pem	2014년 3월 13일	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert89.pem	2014년 3월 13일	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
mozillacert135.pem	2014년 3월 13일	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert27.pem	2014년 3월 13일	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
mozillacert60.pem	2014년 3월 13일	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert78.pem	2014년 3월 13일	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert124.pem	2014년 3월 13일	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert16.pem	2014년 3월 13일	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
mozillacert67.pem	2014년 3월 13일	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
mozillacert113.pem	2014년 3월 13일	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31

이름	날짜	SHA1 지문
mozillacert56.pem	2014년 3월 13일	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
mozillacert102.pem	2014년 3월 13일	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
mozillacert45.pem	2014년 3월 13일	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert109.pem	2014년 3월 13일	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
mozillacert96.pem	2014년 3월 13일	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
mozillacert142.pem	2014년 3월 13일	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
mozillacert34.pem	2014년 3월 13일	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
mozillacert1.pem	2014년 3월 13일	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
mozillacert85.pem	2014년 3월 13일	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
mozillacert131.pem	2014년 3월 13일	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
mozillacert149.pem	2014년 3월 13일	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert23.pem	2014년 3월 13일	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
mozillacert8.pem	2014년 3월 13일	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:6A:0F

이름	날짜	SHA1 지문
mozillacert74.pem	2014년 3월 13일	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert120.pem	2014년 3월 13일	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
mozillacert138.pem	2014년 3월 13일	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
mozillacert12.pem	2014년 3월 13일	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
mozillacert63.pem	2014년 3월 13일	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
mozillacert127.pem	2014년 3월 13일	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert19.pem	2014년 3월 13일	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
mozillacert52.pem	2014년 3월 13일	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F
mozillacert116.pem	2014년 3월 13일	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert41.pem	2014년 3월 13일	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
mozillacert59.pem	2014년 3월 13일	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
mozillacert105.pem	2014년 3월 13일	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC
mozillacert92.pem	2014년 3월 13일	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0

이름	날짜	SHA1 지문
mozillacert30.pem	2014년 3월 13일	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
mozillacert48.pem	2014년 3월 13일	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC
verisignc4g2.pem	2014년 3월 20일	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52
verisignc2g3.pem	2014년 3월 20일	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
verisignc1g6.pem	2014년 12월 31일	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
verisignc2g2.pem	2014년 3월 20일	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
verisignroot.pem	2014년 3월 20일	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
verisignc2g1.pem	2014년 3월 20일	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:14:2A
verisignc3g5.pem	2014년 3월 20일	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
verisignc1g3.pem	2014년 3월 20일	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
verisignc3g4.pem	2014년 3월 20일	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignc1g2.pem	2014년 3월 20일	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
verisignc3g3.pem	2014년 3월 20일	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6

이름	날짜	SHA1 지문
verisignc1g1.pem	2014년 3월 20일	90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:55:6F
verisignc3g2.pem	2014년 3월 20일	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
verisignc3g1.pem	2014년 3월 20일	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
verisignc2g6.pem	2014년 12월 31일	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F
verisignc4g3.pem	2014년 3월 20일	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
gdroot-g2.pem	2014년 12월 31일	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
gd-class2-root.pem	2014년 12월 31일	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
gd_bundle-g2.pem	2014년 12월 31일	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8
dstacescax6	2018년 6월 18일	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
gd_bundle-g2.pem	2018년 6월 18일	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8
verisignc4g3.pem	2018년 6월 18일	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
swisssignplatinumg2ca	2018년 4월 21일	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66

이름	날짜	SHA1 지문
geotrustprimarycertificationauthorityg3	2018년 6월 18일	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycertificationauthorityg2	2018년 6월 18일	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
buypassclass2rootca	2018년 6월 18일	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
camerfirmachambersofcommerceroot	2018년 6월 18일	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert20.pem	2018년 6월 18일	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert12.pem	2018년 6월 18일	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
mozillacert90.pem	2018년 6월 18일	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
mozillacert82.pem	2018년 6월 18일	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
mozillacert140.pem	2018년 6월 18일	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert74.pem	2018년 6월 18일	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert132.pem	2018년 6월 18일	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert66.pem	2018년 6월 18일	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34

이름	날짜	SHA1 지문
mozillacert124.pem	2018년 6월 18일	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert58.pem	2018년 6월 18일	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
securitycommunicationrootca2	2018년 6월 18일	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert116.pem	2018년 6월 18일	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert108.pem	2018년 6월 18일	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
certigna	2018년 6월 18일	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert3.pem	2018년 6월 18일	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
verisignc1g1.pem	2018년 6월 18일	90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:55:6F
verisignc4g2.pem	2018년 6월 18일	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52
deutschetelekomrootca2	2018년 6월 18일	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
starfieldrootg2ca	2018년 4월 21일	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
comodoecccertificationauthority	2018년 6월 18일	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
digicertglobalrootg3	2018년 6월 18일	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E

이름	날짜	SHA1 지문
digicertglobalroot g2	2018년 6월 18일	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73: FE:06:D1:CC:8D:4F:82:A4
mozillacert11.pem	2018년 6월 18일	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E: 4B:DF:B5:A8:99:B2:4D:43
mozillacert81.pem	2018년 6월 18일	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28: A2:59:3A:19:A7:0F:06:9E
mozillacert73.pem	2018년 6월 18일	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D: 92:F4:FE:39:D4:E7:0F:0E
szafirrootca2	2018년 6월 18일	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28: F3:9C:CC:CF:5E:B3:3F:DE
mozillacert131.pem	2018년 6월 18일	37:9A:19:7B:41:85:45:35:0C:A6:03:69: F3:3C:2E:AF:47:4F:20:79
ecacc	2018년 6월 18일	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B: 6D:A7:D6:BA:A6:4A:F2:E8
mozillacert65.pem	2018년 6월 18일	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93: CA:55:6A:F3:EC:AA:35:FB
turktrustelektroni ksertifik ahizmetsa glayicisih5	2018년 6월 18일	C4:18:F6:4D:46:D1:DF:00:3D:27:30:13: 72:43:A9:12:11:C6:75:FB
mozillacert123.pem	2018년 6월 18일	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10: DD:6B:DF:99:72:2C:96:E5
mozillacert57.pem	2018년 6월 18일	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F: CB:34:6E:B2:58:B2:8A:58
mozillacert115.pem	2018년 6월 18일	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62: 32:17:65:CF:17:D8:94:E9

이름	날짜	SHA1 지문
mozillacert49.pem	2018년 6월 18일	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71
mozillacert107.pem	2018년 6월 18일	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
verisignclass3g4ca	2018년 4월 21일	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
securetrustca	2018년 6월 18일	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
mozillacert2.pem	2018년 6월 18일	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
buypassclass2ca	2018년 4월 21일	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
secomscrootca2	2018년 4월 21일	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
secomscrootca1	2018년 4월 21일	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
trustisfpsrootca	2018년 6월 18일	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
hongkongpostrootca1	2018년 6월 18일	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
certsignrootca	2018년 6월 18일	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
geotrustprimaryca	2018년 4월 21일	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
twcaglobalrootca	2018년 6월 18일	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65

이름	날짜	SHA1 지문
camerfirmachambersca	2018년 4월 21일	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
mozillacert10.pem	2018년 6월 18일	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert80.pem	2018년 6월 18일	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert72.pem	2018년 6월 18일	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
comodoaaaca	2018년 4월 21일	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert130.pem	2018년 6월 18일	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert64.pem	2018년 6월 18일	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
mozillacert122.pem	2018년 6월 18일	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
mozillacert56.pem	2018년 6월 18일	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
equifaxsecureebusinessca1	2018년 4월 21일	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
camerfirmachambersignca	2018년 4월 21일	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert114.pem	2018년 6월 18일	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert48.pem	2018년 6월 18일	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC

이름	날짜	SHA1 지문
pscprocert	2018년 6월 18일	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert106.pem	2018년 6월 18일	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
mozillacert1.pem	2018년 6월 18일	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
eecertificationcentrerootca	2018년 6월 18일	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
digicertglobalrootca	2018년 6월 18일	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
thawteprimaryrootcag3	2018년 6월 18일	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
thawteprimaryrootcag2	2018년 6월 18일	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
entrustrootcertificationauthorityec1	2018년 6월 18일	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
valicertclass2ca	2018년 4월 21일	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
globalchambersignroot2008	2018년 6월 18일	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
amazonrootca4	2018년 6월 18일	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
gd-class2-root.pem	2018년 6월 18일	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4

이름	날짜	SHA1 지문
amazonrootca3	2018년 6월 18일	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
amazonrootca2	2018년 6월 18일	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
securitycommunicationrootca	2018년 6월 18일	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
amazonrootca1	2018년 6월 18일	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
acraizfnmtrcm	2018년 6월 18일	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
quovadisrootca3g3	2018년 6월 18일	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D
certplusrootcag2	2018년 6월 18일	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:CC:1A
certplusrootcag1	2018년 6월 18일	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:F7:66
mozillacert71.pem	2018년 6월 18일	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert63.pem	2018년 6월 18일	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
mozillacert121.pem	2018년 6월 18일	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
ttelesecglobalrootclass3ca	2018년 4월 21일	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
mozillacert55.pem	2018년 6월 18일	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12

이름	날짜	SHA1 지문
mozillacert113.pem	2018년 6월 18일	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
baltimorecybertrustca	2018년 4월 21일	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
mozillacert47.pem	2018년 6월 18일	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
mozillacert105.pem	2018년 6월 18일	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC
mozillacert39.pem	2018년 6월 18일	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
usertrustecccertificationauthority	2018년 6월 18일	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0
mozillacert0.pem	2018년 6월 18일	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
securitycommunicationevrootca1	2018년 6월 18일	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
verisignc3g5.pem	2018년 6월 18일	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
globalsignr3ca	2018년 4월 21일	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
trustcoreca1	2018년 6월 18일	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
equifaxsecureglobalebusinessca1	2018년 4월 21일	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
geotrustuniversalca	2018년 6월 18일	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79

이름	날짜	SHA1 지문
affirmtrustpremiumca	2018년 4월 21일	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
staatdernederlandenrootcag3	2018년 6월 18일	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
staatdernederlandenrootcag2	2018년 6월 18일	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
mozillacert70.pem	2018년 6월 18일	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
secomevrootca1	2018년 4월 21일	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
geotrustglobalca	2018년 6월 18일	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert62.pem	2018년 6월 18일	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert120.pem	2018년 6월 18일	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
mozillacert54.pem	2018년 6월 18일	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
mozillacert112.pem	2018년 6월 18일	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
mozillacert46.pem	2018년 6월 18일	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
swisssigngoldcag2	2018년 6월 18일	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert104.pem	2018년 6월 18일	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E

이름	날짜	SHA1 지문
mozillacert38.pem	2018년 6월 18일	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36
certplusclass3pprimaryca	2018년 4월 21일	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
entrustrootcertificationauthorityg2	2018년 6월 18일	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
godaddyrootg2ca	2018년 4월 21일	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
cfcaevroot	2018년 6월 18일	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
verisignc3g4.pem	2018년 6월 18일	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
geotrustuniversalca2	2018년 6월 18일	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
starfieldservicesrootg2ca	2018년 4월 21일	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
digicerthighassuranceevrootca	2018년 6월 18일	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
entrustnetpremium2048secureserverca	2018년 6월 18일	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
camerfirmaglobalchambersignroot	2018년 6월 18일	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
verisignclass3g3ca	2018년 4월 21일	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
godaddyclass2ca	2018년 6월 18일	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4

이름	날짜	SHA1 지문
mozillacert61.pem	2018년 6월 18일	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
mozillacert53.pem	2018년 6월 18일	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
atostrustedroot2011	2018년 6월 18일	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert111.pem	2018년 6월 18일	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
staatdernederlandenevrootca	2018년 6월 18일	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
mozillacert45.pem	2018년 6월 18일	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert103.pem	2018년 6월 18일	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert37.pem	2018년 6월 18일	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert29.pem	2018년 6월 18일	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
izenpecom	2018년 6월 18일	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
comodorsacertificationauthority	2018년 6월 18일	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
mozillacert99.pem	2018년 6월 18일	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
mozillacert149.pem	2018년 6월 18일	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1

이름	날짜	SHA1 지문
utnuserfirstobjectca	2018년 4월 21일	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46
verisignc3g3.pem	2018년 6월 18일	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
dstrootcax3	2018년 6월 18일	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
addtrustexternalroot	2018년 6월 18일	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
certumtrustednetworkca	2018년 6월 18일	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
affirmtrustpremiumecc	2018년 6월 18일	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
starfieldclass2ca	2018년 6월 18일	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
actalisauthenticationrootca	2018년 6월 18일	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
verisignclass2g3ca	2018년 4월 21일	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
isrgrootx1	2018년 6월 18일	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
godaddyrootcertificateauthorityg2	2018년 6월 18일	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
mozillacert60.pem	2018년 6월 18일	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
chunghwaepkirootca	2018년 4월 21일	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0

이름	날짜	SHA1 지문
mozillacert52.pem	2018년 6월 18일	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F
microseceszignorootca2009	2018년 6월 18일	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
securesignrootca11	2018년 6월 18일	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert110.pem	2018년 6월 18일	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert44.pem	2018년 6월 18일	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
mozillacert102.pem	2018년 6월 18일	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
mozillacert36.pem	2018년 6월 18일	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D
mozillacert28.pem	2018년 6월 18일	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
baltimorecybertrustroot	2018년 6월 18일	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
amzninternalrootca	2008년 12월 12일	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06
mozillacert98.pem	2018년 6월 18일	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
mozillacert148.pem	2018년 6월 18일	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
verisignc3g2.pem	2018년 6월 18일	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F

이름	날짜	SHA1 지문
quovadisrootca2g3	2018년 6월 18일	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
geotrustprimarycertificatio nauthority	2018년 6월 18일	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
opentrustrootcag3	2018년 6월 18일	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8A:A6
opentrustrootcag2	2018년 6월 18일	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	2018년 6월 18일	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:D5:7E
verisignclass3ca	2018년 4월 21일	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
globalsignca	2018년 4월 21일	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
ttelesecglobalrootclass2ca	2018년 4월 21일	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
verisignclass1g3ca	2018년 4월 21일	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
verisignuniversalrootca	2018년 4월 21일	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
soneraclass2ca	2018년 4월 21일	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
starfieldservicesrootcertificateauthorityg2	2018년 6월 18일	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F

이름	날짜	SHA1 지문
mozillacert51.pem	2018년 6월 18일	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
mozillacert43.pem	2018년 6월 18일	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
mozillacert101.pem	2018년 6월 18일	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
mozillacert35.pem	2018년 6월 18일	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
globalsignr2ca	2018년 4월 21일	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
mozillacert27.pem	2018년 6월 18일	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
affirmtrustpremium	2018년 6월 18일	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert19.pem	2018년 6월 18일	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
mozillacert97.pem	2018년 6월 18일	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
netlockaranyclassgoldfotanusitvany	2018년 6월 18일	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
mozillacert89.pem	2018년 6월 18일	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
verisignroot.pem	2018년 6월 18일	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
mozillacert147.pem	2018년 6월 18일	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4

이름	날짜	SHA1 지문
aolrootca2	2018년 4월 21일	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	2016년 11월 23일	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
aolrootca1	2018년 4월 21일	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignc3g1.pem	2018년 6월 18일	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert139.pem	2018년 6월 18일	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
soneraclass2rootca	2018년 6월 18일	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
swisssignsilverg2ca	2018년 4월 21일	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
thawteprimaryrootca	2018년 6월 18일	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
gdcatrustauthr5root	2018년 6월 18일	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
trustcenterclass4caii	2018년 4월 21일	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
usertrustrsacertificationauthority	2018년 6월 18일	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
digicertassuredidrootg3	2018년 6월 18일	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
digicertassuredidrootg2	2018년 6월 18일	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F

이름	날짜	SHA1 지문
mozillacert50.pem	2018년 6월 18일	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
mozillacert42.pem	2018년 6월 18일	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
mozillacert100.pem	2018년 6월 18일	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
mozillacert34.pem	2018년 6월 18일	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
affirmtrustcommercialca	2018년 4월 21일	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert26.pem	2018년 6월 18일	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
globalsigneccrootcar5	2018년 6월 18일	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
globalsigneccrootcar4	2018년 6월 18일	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
buypassclass3rootca	2018년 6월 18일	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert18.pem	2018년 6월 18일	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
mozillacert96.pem	2018년 6월 18일	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
verisignc2g6.pem	2018년 6월 18일	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F
secomvalicertclass1ca	2018년 4월 21일	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E

이름	날짜	SHA1 지문
mozillacert88.pem	2018년 6월 18일	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
accvraiz1	2018년 6월 18일	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert146.pem	2018년 6월 18일	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A
mozillacert138.pem	2018년 6월 18일	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
verisignclass3g2ca	2018년 4월 21일	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
dtrustrootclass3ca2ev2009	2018년 6월 18일	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
xrampglobalca	2018년 4월 21일	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
mozillacert9.pem	2018년 6월 18일	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
verisignuniversalrootcertificationauthority	2018년 6월 18일	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
tubitakkamusmsslkoksertifikasisurum1	2018년 6월 18일	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
mozillacert41.pem	2018년 6월 18일	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
mozillacert33.pem	2018년 6월 18일	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D

이름	날짜	SHA1 지문
mozillacert25.pem	2018년 6월 18일	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
mozillacert17.pem	2018년 6월 18일	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert95.pem	2018년 6월 18일	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
affirmtrustpremiumeccca	2018년 4월 21일	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert87.pem	2018년 6월 18일	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert145.pem	2018년 6월 18일	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:3A:04
mozillacert79.pem	2018년 6월 18일	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert137.pem	2018년 6월 18일	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A1:D1
digicertassuredidrootca	2018년 6월 18일	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
addtrustqualifiedca	2018년 4월 21일	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert129.pem	2018년 6월 18일	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
verisignclass2g2ca	2018년 4월 21일	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
baltimorecodesigningca	2018년 4월 21일	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D

이름	날짜	SHA1 지문
luxtrustglobalroot2	2018년 6월 18일	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5F:3F
visaecommerceroot	2018년 6월 18일	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
oistewisekeyglobalrootgbca	2018년 6월 18일	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
mozillacert8.pem	2018년 6월 18일	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:6A:0F
comodocertificationauthority	2018년 6월 18일	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	2016년 11월 23일	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
verisignc1g6.pem	2018년 6월 18일	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
trustcenterclass2caii	2018년 4월 21일	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
quovadisrootca1g3	2018년 6월 18일	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67
mozillacert40.pem	2018년 6월 18일	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
cadisigrootr2	2018년 6월 18일	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
cadisigrootr1	2018년 6월 18일	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
mozillacert32.pem	2018년 6월 18일	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C

이름	날짜	SHA1 지문
utndatacorpsgcca	2018년 4월 21일	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
mozillacert24.pem	2018년 6월 18일	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
addtrustclass1ca	2018년 4월 21일	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
mozillacert16.pem	2018년 6월 18일	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
affirmtrustnetworkingca	2018년 4월 21일	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert94.pem	2018년 6월 18일	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
mozillacert86.pem	2018년 6월 18일	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
mozillacert144.pem	2018년 6월 18일	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
mozillacert78.pem	2018년 6월 18일	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert136.pem	2018년 6월 18일	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert128.pem	2018년 6월 18일	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D
verisignclass1g2ca	2018년 4월 21일	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47

이름	날짜	SHA1 지문
hellenicacademicandresearchinstitutionsrootca2015	2018년 6월 18일	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
soneraclass1ca	2018년 4월 21일	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
hellenicacademicandresearchinstitutionsrootca2011	2018년 6월 18일	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
certumtrustednetworkca2	2018년 6월 18일	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
equifaxsecureca	2018년 4월 21일	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
thawteserverca	2018년 4월 21일	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
mozillacert7.pem	2018년 6월 18일	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
affirmtrustnetworking	2018년 6월 18일	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
deprecateditsecca	2012년 1월 27일	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
globalsignrootcar3	2018년 6월 18일	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
globalsignrootcar2	2018년 6월 18일	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE

이름	날짜	SHA1 지문
quovadisrootca	2018년 6월 18일	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert31.pem	2018년 6월 18일	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
entrustrootcertificationauthority	2018년 6월 18일	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert23.pem	2018년 6월 18일	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
mozillacert15.pem	2018년 6월 18일	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
verisignc2g3.pem	2018년 6월 18일	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
mozillacert93.pem	2018년 6월 18일	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17
mozillacert151.pem	2018년 6월 18일	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
mozillacert85.pem	2018년 6월 18일	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
certplusclass2primaryca	2018년 6월 18일	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
mozillacert143.pem	2018년 6월 18일	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
mozillacert77.pem	2018년 6월 18일	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
mozillacert135.pem	2018년 6월 18일	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18

이름	날짜	SHA1 지문
mozillacert69.pem	2018년 6월 18일	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
mozillacert127.pem	2018년 6월 18일	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert119.pem	2018년 6월 18일	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
geotrustprimarycag3	2018년 4월 21일	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
identrustpublicsectorrootca1	2018년 6월 18일	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
geotrustprimarycag2	2018년 4월 21일	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
trustcorrootcertca2	2018년 6월 18일	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
mozillacert6.pem	2018년 6월 18일	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
trustcorrootcertca1	2018년 6월 18일	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
networksolutionscertificate authority	2018년 6월 18일	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
twcarootcertificationauthority	2018년 6월 18일	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
addtrustexternalca	2018년 4월 21일	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68

이름	날짜	SHA1 지문
verisignclass3g5ca	2018년 4월 21일	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
autoridaddecertificacionfirmaprofesionalcifa62634068	2018년 6월 18일	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
hellenicacademicandresearchinstitutionseccrootca2015	2018년 6월 18일	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
verisigntsaca	2018년 4월 21일	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
utnuserfirsthardwareca	2018년 4월 21일	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
identrustcommercialrootca1	2018년 6월 18일	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
dtrustrootclass3ca22009	2018년 6월 18일	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
epkirootcertificationauthority	2018년 6월 18일	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert30.pem	2018년 6월 18일	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
teliasonerarootcav1	2018년 6월 18일	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
buypassclass3ca	2018년 4월 21일	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57

이름	날짜	SHA1 지문
mozillacert22.pem	2018년 6월 18일	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
mozillacert14.pem	2018년 6월 18일	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
verisignc2g2.pem	2018년 6월 18일	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
certumca	2018년 4월 21일	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert92.pem	2018년 6월 18일	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0
mozillacert150.pem	2018년 6월 18일	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
mozillacert84.pem	2018년 6월 18일	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2
ttelesecglobalrootclass3	2018년 6월 18일	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
globalsignrootca	2018년 6월 18일	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
ttelesecglobalrootclass2	2018년 6월 18일	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert142.pem	2018년 6월 18일	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
mozillacert76.pem	2018년 6월 18일	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert134.pem	2018년 6월 18일	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62

이름	날짜	SHA1 지문
mozillacert68.pem	2018년 6월 18일	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
etugracertificationauthority	2018년 6월 18일	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert126.pem	2018년 6월 18일	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
keynectisrootca	2018년 4월 21일	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
mozillacert118.pem	2018년 6월 18일	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45
quovadisrootca3	2018년 6월 18일	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	2018년 6월 18일	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert5.pem	2018년 6월 18일	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
verisignc1g3.pem	2018년 6월 18일	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
cybertrustglobalroot	2018년 6월 18일	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
amzninternalinfosecag3	2015년 2월 27일	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
starfieldrootcertificateauthorityg2	2018년 6월 18일	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
entrust2048ca	2018년 4월 21일	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31

이름	날짜	SHA1 지문
swisssignsilvercag 2	2018년 6월 18일	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25: 93:DF:A7:F0:40:D1:1D:CB
affirmtrustcommerc ial	2018년 6월 18일	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80: DC:E9:6E:2C:C7:B2:78:B7
certinomisrootca	2018년 6월 18일	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C: 01:B9:32:C5:34:E7:88:A8
xrampglobalcaroot	2018년 6월 18일	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30: 54:F3:4C:52:B7:E5:58:C6
secureglobalca	2018년 6월 18일	3A:44:73:5A:E5:81:90:1F:24:86:61:46: 1E:3B:9C:C4:5F:F5:3A:1B
swisssigngoldg2ca	2018년 4월 21일	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6: 45:25:3A:6F:9F:1A:27:61
mozillacert21.pem	2018년 6월 18일	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25: 93:DF:A7:F0:40:D1:1D:CB
mozillacert13.pem	2018년 6월 18일	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B: A9:03:D0:06:B7:97:09:91
verisignc2g1.pem	2018년 6월 18일	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0: CD:14:68:0A:4F:60:14:2A
mozillacert91.pem	2018년 6월 18일	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22: 93:D9:DF:F5:4B:81:C0:04
oistewisekeyglobal rootgaca	2018년 6월 18일	59:22:A1:E1:5A:EA:16:35:21:F8:98:39: 6A:46:46:B0:44:1B:0F:A9
mozillacert83.pem	2018년 6월 18일	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13: 0A:85:58:57:CC:9C:EA:46
entrustevca	2018년 4월 21일	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37: D4:4D:F5:D4:67:49:52:F9

이름	날짜	SHA1 지문
mozillacert141.pem	2018년 6월 18일	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
mozillacert75.pem	2018년 6월 18일	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
mozillacert133.pem	2018년 6월 18일	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
mozillacert67.pem	2018년 6월 18일	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
mozillacert125.pem	2018년 6월 18일	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert59.pem	2018년 6월 18일	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
thawtepremiumserverca	2018년 4월 21일	E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:66:66
mozillacert117.pem	2018년 6월 18일	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
utnuserfirstclientauthemailca	2018년 4월 21일	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
entrustrootcag2	2018년 4월 21일	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
mozillacert109.pem	2018년 6월 18일	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
digicerttrustedrootg4	2018년 6월 18일	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
gdroot-g2.pem	2018년 6월 18일	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B

이름	날짜	SHA1 지문
comodoaaaservicesroot	2018년 6월 18일	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert4.pem	2018년 6월 18일	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B
verisignclass3publicprimarycertificationauthorityg5	2018년 6월 18일	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
chambersofcommerceroot2008	2018년 6월 18일	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
verisignclass3publicprimarycertificationauthorityg4	2018년 6월 18일	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3publicprimarycertificationauthorityg3	2018년 6월 18일	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
thawtepersonalfreemailca	2018년 4월 21일	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
verisignc1g2.pem	2018년 6월 18일	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
gtecybertrustglobalca	2018년 4월 21일	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
trustcenteruniversalcai	2018년 4월 21일	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3

이름	날짜	SHA1 지문
camerfirmachambers commerceca	2018년 4월 21일	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0: DB:72:2E:31:30:61:F0:B1
verisignclass1ca	2018년 4월 21일	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45: 3E:64:09:EA:E8:7D:60:F1

해석기 매핑 템플릿 변경 로그

> **Note**
>
> 이제 우리는 주로 APPSYNC_JS 런타임과 해당 문서를 지원합니다. 여기에서 APPSYNC_JS 런타임과 해당 안내서를 사용해 보세요.

해석기 및 함수 매핑 템플릿에는 버전이 지정됩니다. 이 매핑 템플릿 버전(예: 2018-05-29)은 다음을 나타냅니다. * 요청 템플릿에서 제공하는 데이터 소스 요청 구성의 예상 모양 * 요청 매핑 템플릿 및 응답 매핑 템플릿의 실행 동작

버전은 YYYY-MM-DD 형식으로 표시되는데, 이후 날짜는 최신 버전에 해당합니다. 이 페이지에는 AWS AppSync에서 현재 지원되는 매핑 템플릿 버전 간의 차이가 나와 있습니다.

주제

- 버전 매트릭스에 따른 데이터 원본 작업 가용성
- 유닛 해석기 매핑 템플릿에 대한 버전 변경
- 함수에 대한 버전 변경
- 2018-05-29
- 2017-02-28

버전 매트릭스에 따른 데이터 원본 작업 가용성

지원되지 않는 작업/버전	2017-02-28	2018-05-29
AWS Lambda 호출	예	예

지원되지 않는 작업/버전	2017-02-28	2018-05-29
AWS Lambda BatchInvoke	예	예
None Datasource	예	예
Amazon OpenSearch GET	예	예
Amazon OpenSearch POST	예	예
Amazon OpenSearch PUT	예	예
Amazon OpenSearch DELETE	예	예
Amazon OpenSearch GET	예	예
DynamoDB GetItem	예	예
DynamoDB Scan	예	예
DynamoDB Query	예	예
DynamoDB DeleteItem	예	예
DynamoDB PutItem	예	예
DynamoDB BatchGetItem	아니요	예
DynamoDB BatchPutItem	아니요	예
DynamoDB BatchDeleteItem	아니요	예
HTTP	아니요	예
Amazon RDS	아니요	예

참고: 현재, 함수에서는 2018-05-29 버전만 지원됩니다.

유닛 해석기 매핑 템플릿에 대한 버전 변경

유닛 해석기의 경우 버전은 요청 매핑 템플릿 본문의 일부로 지정됩니다. 버전을 업데이트하려면 `version` 필드를 새 버전으로 업데이트하기만 하면 됩니다.

예를 들어, AWS Lambda 템플릿에서 버전을 업데이트하려면:

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

아래와 같이 버전 필드를 `2017-02-28`에서 `2018-05-29`로 업데이트해야 합니다.

```
{
    "version": "2018-05-29", ## Note the version
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

함수에 대한 버전 변경

함수의 경우 버전은 함수 객체에 대한 `functionVersion` 필드로 지정됩니다. 버전을 업데이트하려면 `functionVersion`을 업데이트하기만 하면 됩니다. 참고: 현재, 함수에는 `2018-05-29`만 지원됩니다.

다음은 기존 함수 버전을 업데이트하기 위한 CLI 명령 예제입니다.

```
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)"
```

참고: 적용되지 않을 것이기 때문에 버전 필드는 함수 요청 매핑 템플릿에서 생략하는 것이 좋습니다. 함수 요청 매핑 템플릿 내에서 버전을 지정하면 functionVersion 필드의 값으로 버전이 재지정됩니다.

2018-05-29

동작 변경

- 데이터 원본 호출 결과가 null이면 응답 매핑 템플릿이 실행됩니다.
- 데이터 원본을 호출했는데 오류가 발생하면 오류는 직접 처리해야 합니다. 평가된 응답 매핑 템플릿은 항상 GraphQL 응답 data 블록 내에 배치됩니다.

추론

- null 호출 결과에는 의미가 있는데, 일부 애플리케이션 사용 사례에서는 사용자 지정 방식으로 null 결과를 처리하고자 할 수 있습니다. 예를 들어, 애플리케이션이 권한 부여 확인을 수행하기 위해 Amazon DynamoDB 테이블에 레코드가 있는지 확인하려고 할 수 있습니다. 이러한 경우 null 호출 결과는 사용자가 인증되지 않았을 수 있음을 의미합니다. 이제, 응답 매핑 템플릿을 실행하면 권한 없음 오류를 발생시킬 수 있습니다. 이 동작은 API 설계자에게 더 정확한 제어 권한을 제공합니다.

다음 응답 매핑 템플릿에 대해 생각해 보겠습니다.

```
$util.toJson($ctx.result)
```

이전에는 2017-02-28을 사용했을 때 $ctx.result가 null을 반환하면 응답 매핑 템플릿이 실행되지 않았습니다. 이제, 2018-05-29를 사용하면 이 시나리오를 처리할 수 있습니다. 예를 들어, 다음과 같이 권한 부여 오류가 실행되도록 선택할 수 있습니다.

```
# throw an unauthorized error if the result is null
#if ( $util.isNull($ctx.result) )
    $util.unauthorized()
#end
$util.toJson($ctx.result)
```

참고: 데이터 원본에서 반환되는 오류는 때때로 치명적이지 않고 심지어 예측되는 경우도 있는데, 이는 응답 매핑 템플릿에 호출 오류를 처리하고 해당 오류를 무시하거나, 다시 실행하거나, 다른 오류를 실행할지 여부를 결정할 수 있는 유연성이 있어야 하기 때문입니다.

다음 응답 매핑 템플릿에 대해 생각해 보겠습니다.

```
$util.toJson($ctx.result)
```

이전에 `2017-02-28`을 사용했을 때 호출 오류가 발생하면 응답 매핑 템플릿이 평가되고 그 결과가 GraphQL 응답의 `errors` 블록에 자동으로 배치되었습니다. 이제, `2018-05-29`를 사용하면 오류와 관련해 수행할 작업을 선택하거나, 오류를 다시 실행하거나, 다른 오류를 실행하거나, 데이터를 반환하는 동안 오류를 추가할 수 있습니다.

호출 오류 다시 실행

다음 응답 템플릿에서는 데이터 원본에서 반환되는 것과 동일한 오류를 실행합니다.

```
#if ( $ctx.error )
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

호출 오류(예: `$ctx.error` is present)가 발생하면 응답은 다음과 같이 표시됩니다.

```
{
    "data": {
        "getPost": null
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "DynamoDB:ConditionalCheckFailedException",
            "message": "Conditional check failed exception...",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
```

```
            ]
        }
    ]
}
```

다른 오류 실행

다음 응답 템플릿에서는 데이터 원본에서 반환되는 오류를 처리한 후 고유한 사용자 지정 오류를 실행합니다.

```
#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)
```

호출 오류(예: `$ctx.error` is present)가 발생하면 응답은 다음과 같이 표시됩니다.

```
{
    "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
                }
            ]
```

```
        }
    ]
}
```

반환 데이터에 오류 추가

다음 응답 템플릿에서는 응답 내에서 데이터를 반환하는 동안 데이터 원본에서 반환되는 것과 동일한 오류를 추가합니다. 이를 부분 응답이라고도 합니다.

```
#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
```

호출 오류(예: `$ctx.error` is present)가 발생하면 응답은 다음과 같이 표시됩니다.

```
{
    "data": {
        "getPost": {
            "id": "1",
            "title: "A post"
        }
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "ConditionalCheckFailedException",
            "message": "Conditional check failed exception...",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

2017-02-28에서 2018-05-29로 마이그레이션

2017-02-28에서 2018-05-29로 마이그레이션은 간단합니다. 해석기 요청 매핑 템플릿 또는 함수 버전 객체에서 버전 필드를 변경합니다. 그러나 2018-05-29 실행은 2017-02-28 실행과 다르게 동작하는데, 동작 변경 사항은 여기에 간단하게 정리되어 있습니다.

2017-02-28에서 2018-05-29로 동일한 실행 동작 보존

경우에 따라 2018-05-29 버전의 템플릿을 실행하는 동안 2017-02-28 버전과 동일한 실행 동작을 유지할 수 있습니다.

예: DynamoDB PutItem

다음과 같은 2017-02-28 DynamoDB PutItem 요청 템플릿이 있습니다.

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "attributeValues" : {
        "baz" : ... typed value
    },
    "condition" : {
       ...
    }
}
```

응답 템플릿은 다음과 같습니다.

```
$util.toJson($ctx.result)
```

2018-05-29로 마이그레이션하면 이러한 템플릿은 다음과 같이 변경됩니다.

```
{
    "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" : {
       ...
    }
}
```

응답 템플릿은 다음과 같이 변경됩니다.

```
## 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)
```

따라서 오류를 처리하는 것은 고객의 책임입니다. 여기서는 DynamoDB에서 반환된 `$util.error()`를 사용하여 동일한 오류를 실행하도록 선택합니다. 이 코드 조각은 매핑 템플릿을 2018-05-29로 변환하도록 수정할 수 있습니다. 응답 템플릿이 다른 경우에는 실행 동작의 변경을 고려해야 합니다.

예: DynamoDB GetItem

다음과 같은 2017-02-28 DynamoDB GetItem 요청 템플릿이 있습니다.

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true
```

```
}
```

응답 템플릿은 다음과 같습니다.

```
## map table attribute postId to field Post.id
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))

$util.toJson($ctx.result)
```

2018-05-29로 마이그레이션하면 이러한 템플릿은 다음과 같이 변경됩니다.

```
{
    "version" : "2018-05-29", ## Note the new 2018-05-29 version
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true
}
```

응답 템플릿은 다음과 같이 변경됩니다.

```
## 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)
```

2017-02-28 버전에서 데이터 원본 호출이 `null`이었으면 이는 키를 일치시키는 DynamoDB 테이블에 항목이 없음을 의미해 응답 매핑 템플릿이 실행되지 않았을 것입니다. 대부분의 경우 이러한 결과는 괜

찾을 수 있지만, `$ctx.result`가 `null`이 아니길 기대하는 경우에는 이제, 해당 시나리오를 처리해야 합니다.

2017-02-28

특성

- 데이터 원본 호출 결과가 `null`이면 응답 매핑 템플릿이 실행되지 않습니다.
- 데이터 원본 호출 시 오류가 발생하면 응답 매핑 템플릿이 실행되고 평가된 결과가 GraphQL 응답 `errors.data` 블록 내에 배치됩니다.

유형 참조

이 섹션은 스키마 유형에 대한 참조로 사용됩니다.

AWS AppSync의 스칼라 형식

GraphQL 객체 유형에는 이름과 필드가 있으며 이러한 필드에는 하위 필드가 있을 수 있습니다. 궁극적으로 객체 유형의 필드는 쿼리의 리프를 나타내는 스칼라 유형으로 해석되어야 합니다. 객체 유형과 스칼라에 대한 자세한 내용은 GraphQL 웹 사이트의 스키마 및 유형을 참조하세요.

GraphQL 스칼라의 기본 세트 외에도 AWS AppSync를 사용하면 AWS 접두사로 시작하는 서비스 정의 스칼라를 사용할 수도 있습니다. AWS AppSync는 사용자 정의(사용자 정의) 스칼라 생성을 지원하지 않습니다. 기본 스칼라 또는 AWS 스칼라를 사용해야 합니다.

AWS를 사용자 지정 객체 형식의 접두사로 사용할 수 없습니다.

다음 섹션은 스키마 입력에 대한 참조입니다.

기본 스칼라

GraphQL은 다음과 같은 디폴트 스칼라를 정의합니다.

기본 스칼라 목록

`ID`

객체의 고유한 식별자입니다. 이 스칼라는 `String`처럼 직렬화되어 있지만 사람이 읽을 수는 없습니다.

`String`

UTF-8 문자 시퀀스입니다.

`Int`

$-(2^{31})$과 $2^{31}-1$ 사이의 정수 값입니다.

`Float`

IEEE 754 부동 소수점 값입니다.

`Boolean`

부울 값으로 `true` 또는 `false` 둘 중 하나입니다.

AWS AppSync 스칼라

AWS AppSync는 다음 스칼라를 정의합니다.

AWS AppSync 스칼라 목록

`AWSDate`

YYYY-MM-DD 형식의 확장 ISO 8601 날짜 문자열입니다.

`AWSTime`

hh:mm:ss.sss 형식의 확장 ISO 8601 시간 문자열입니다.

`AWSDateTime`

YYYY-MM-DDThh:mm:ss.sssZ 형식의 확장 ISO 8601 날짜 및 시간 문자열입니다.

> **Note**
>
> AWSDate, AWSTime 및 AWSDateTime 스칼라는 선택적으로 시간대 오프셋을 포함할 수 있습니다. 예를 들어,1970-01-01Z, 1970-01-01-07:00 및 1970-01-01+05:30 값은 AWSDate에 대해 모두 유효합니다. 시간대 오프셋은 Z(UTC)이거나 시간 및 분(및 선택적으로 초) 단위의 오프셋이어야 합니다. 예: ±hh:mm:ss. 시간대 오프셋의 초 필드는 ISO 8601 표준을 따르지 않더라도 유효한 것으로 간주됩니다.

`AWSTimestamp`

1970-01-01-T00:00Z 이전 또는 이후의 시간(초)을 나타내는 정수 값입니다.

`AWSEmail`

RFC 822에 정의된 local-part@domain-part 형식의 이메일 주소입니다.

`AWSJSON`

JSON 문자열입니다. 유효한 JSON 구문은 리터럴 입력 문자열이 아닌 맵, 목록 또는 스칼라 값으로 자동 구문 분석되어 해석기 코드에 로드됩니다. 인용되지 않은 문자열이나 잘못된 JSON으로 인해 GraphQL 검증 오류가 발생합니다.

AWSPhone

전화번호입니다. 이 값은 문자열로 저장됩니다. 전화번호에는 숫자 그룹을 구분하기 위한 공백이나 하이픈이 포함될 수 있습니다. 국가 코드가 없는 전화번호는 북아메리카 번호 체계(NANP)를 준수하는 미국/북미 번호로 간주됩니다.

AWSURL

RFC 1738에 정의된 URL입니다. 예: `https://www.amazon.com/dp/B000NZW3KC/` 또는 `mailto:example@example.com`. URL은 스키마(`http`, `mailto`)를 포함해야 하며 경로 부분에 슬래시(`//`)를 두 개 포함할 수 없습니다.

AWSIPAddress

유효한 IPv4 또는 IPv6 주소입니다. IPv4 주소는 네 개의 점으로 구분된 표기법(`123.12.34.56`)으로 표시되어야 합니다. IPv6 주소는 괄호가 없고 콜론으로 구분된 형식(`1a2b:3c4b::1234:4567`)이어야 합니다. 서브넷 마스크를 나타내는 선택적 CIDR 접미사(`123.45.67.89/16`)를 포함할 수 있습니다.

스키마 예제 사용

다음 예제 GraphQL 스키마는 모든 사용자 지정 스칼라를 '객체'로 사용하고 기본 put, get 및 list 작업에 대한 해석기 요청 및 응답 템플릿을 보여줍니다. 마지막으로 예제는 쿼리 및 뮤테이션을 실행할 때 이를 사용하는 방법을 보여줍니다.

```
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
}
```

putObject에 대한 요청 템플릿의 모양은 다음과 같습니다. putObject는 PutItem 작업을 사용하여 Amazon DynamoDB 테이블에서 항목을 생성하거나 업데이트합니다. 참고로 이 코드 조각에는 데이터 소스로 구성된 Amazon DynamoDB 테이블이 없습니다. 다음은 예시용으로만 사용되고 있습니다.

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

putObject의 응답 템플릿은 다음과 같은 결과를 반환합니다.

```
$util.toJson($ctx.result)
```

getObject에 대한 요청 템플릿의 모양은 다음과 같습니다. getObject는 GetItem 작업을 시작하여 주어진 기본 키와 함께 항목에 대한 속성 집합을 반환합니다. 참고로 이 코드 조각에는 데이터 소스로 구성된 Amazon DynamoDB 테이블이 없습니다. 다음은 예시용으로만 사용되고 있습니다.

```
{
    "version": "2017-02-28",
    "operation": "GetItem",
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
    }
}
```

getObject의 응답 템플릿은 다음과 같은 결과를 반환합니다.

```
$util.toJson($ctx.result)
```

listObjects에 대한 요청 템플릿의 모양은 다음과 같습니다. listObjects는 Scan 작업을 사용하여 하나 이상의 항목과 속성을 반환합니다. 참고로 이 코드 조각에는 데이터 소스로 구성된 Amazon DynamoDB 테이블이 없습니다. 다음은 예시용으로만 사용되고 있습니다.

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
}
```

listObjects의 응답 템플릿은 다음과 같은 결과를 반환합니다.

```
$util.toJson($ctx.result.items)
```

다음은 이 스키마를 GraphQL 쿼리와 함께 사용하는 몇 가지 예제입니다.

```
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
    }
}
```

GraphQL의 인터페이스 및 공용 구조체

GraphQL 형식 시스템은 인터페이스를 지원합니다. 인터페이스는 인터페이스 구현을 위해 형식에 포함해야 하는 특정 필드 집합을 노출시킵니다.

GraphQL 형식 시스템은 공용 구조체도 지원합니다. 공용 구조체는 필드의 공통 필드 집합을 정의하지 않는다는 점을 제외하면 인터페이스와 동일합니다. 가능한 형식이 논리적 계층 구조를 공유하지 않는 경우 공용 구조체는 일반적으로 인터페이스보다 우선합니다.

다음 섹션은 스키마 입력에 대한 참조입니다.

인터페이스 예시

어떤 종류의 활동 또는 사람들의 모임을 나타내는 Event 인터페이스를 표시할 수 있습니다. 가능한 이벤트 유형으로는Concert, Conference, Festival이 있습니다. 이러한 형식은 모두 동일한 특성을 공유하고, 모두 이름, 이벤트가 발생하는 장소와 시작 날짜 및 종료 날짜를 가지고 있습니다. 하지만 이러한 형식에도 차이가 있는데, Conference는 발표자 및 워크숍 목록을 제공하는 반면에 Concert에는 공연하는 밴드가 있습니다.

SDL(Schema Definition Language)에서 Event 인터페이스는 다음과 같이 정의됩니다.

```
interface Event {
        id: ID!
        name : String!
        startsAt: String
        endsAt: String
        venue: Venue
        minAgeRestriction: Int
}
```

Event 인터페이스를 구현하는 각 형식은 다음과 같습니다.

```
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]
}
```

인터페이스는 여러 형식일 수 있는 요소를 나타내는 데 유용합니다. 예를 들어, 특정 장소에서 개최되는 모든 이벤트를 검색할 수 있습니다. 스키마에 다음과 같이 `findEventsByVenue` 필드를 추가해 보겠습니다.

```
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]
}
```

`findEventsByVenue`는 `Event` 목록을 반환합니다. GraphQL 인터페이스 필드는 구현하는 모든 형식에 공통적이기 때문에 `Event` 인터페이스에 대해 모든 필드를 선택할 수 있습니다(`id`, `name`, `startsAt`, `endsAt`, `venue` 및 `minAgeRestriction`). 또한 GraphQL 조각을 사용하여 형식을 지정하는 한 구현하는 모든 형식에 대한 필드에 액세스할 수 있습니다.

인터페이스를 사용하는 GraphQL 쿼리의 예를 살펴보겠습니다.

```
query {
  findEventsAtVenue(venueId: "Madison Square Garden") {
    id
    name
    minAgeRestriction
    startsAt

    ... on Festival {
      performers
```

```
    }

    ... on Concert {
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

이전 쿼리에서는 결과 목록 하나가 생성되었는데 기본적으로 서버에서는 시작 날짜를 기준으로 이벤트를 정렬합니다.

```
{
  "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"
        ]
      }
    ]
  }
}
```

결과는 단일 이벤트 컬렉션으로 반환되므로 공통 특성을 나타내는 데 인터페이스를 사용하면 결과를 정렬하는 데 매우 유용합니다.

공용 구조체 예시

앞서 설명했듯이 공용 구조체는 공통 필드 집합을 정의하지 않습니다. 검색 결과에 여러 가지 다양한 형식이 나타날 수 있습니다. Event 스키마를 사용하면 SearchResult 공용 구조체를 다음과 같이 정의할 수 있습니다.

```
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
```

이 경우 SearchResult 공용 구조체에 대한 필드를 쿼리하려면 조각을 사용해야 합니다.

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
      id
      name
      performers
```

```
    }

    ... on Concert {
      id
      name
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

AWS AppSync의 형식 확인

형식 확인은 GraphQL 엔진이 확인된 값을 특정 객체 형식으로 식별하는 메커니즘입니다.

공용 구조체 검색의 예로 돌아가 보겠습니다. 쿼리 결과가 생성되면 결과 목록의 각 항목에는 `SearchResult` 공용 구조체에서 정의한 가능한 형식 중 하나(예: `Conference`, `Festival`, `Concert` 또는 `Venue`)로 자신이 표시되어 있어야 합니다.

`Festival`을 `Venue` 또는 `Conference`와 식별하는 로직은 애플리케이션 요구 사항을 따르기 때문에 GraphQL 엔진에 원시 결과로부터 가능한 형식을 식별하기 위한 힌트를 제공해야 합니다.

AWS AppSync에서 이러한 힌트는 값이 식별되는 객체 형식 이름에 해당하는 `__typename`이라는 메타 필드로 표시됩니다. `__typename`은 인터페이스 또는 공용 구조체인 반환 형식에 필요합니다.

형식 확인의 예

이전 스키마를 다시 사용하겠습니다. 콘솔로 이동해 스키마 페이지에서 다음을 추가하여 수행합니다.

```
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
}
```

Query.search 필드에 해석기를 연결해 보십시오. Resolvers 섹션에서 연결을 선택하고 없음 형식의 새 데이터 원본을 만든 다음 이름을 StubDataSource로 지정합니다. 이 예에서는 외부 소스에서 결과를 가져와 요청 매핑 템플릿에서 가져온 결과를 하드코딩한다고 가정합니다.

요청 매핑 템플릿 창에서 다음을 입력합니다.

```
{
    "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"]
        }
    ]
}
```

애플리케이션이 형식 이름을 id 필드의 일부로 반환하는 경우 형식 확인 로직은 id 필드를 구문 분석하여 형식 이름을 추출한 다음 __typename 필드를 각 결과에 추가해야 합니다. 다음과 같이 응답 매핑 템플릿에서 이러한 로직을 수행할 수 있습니다.

> **Note**
>
> Lambda 데이터 원본을 사용하는 경우 Lambda 함수의 일부분으로 이 작업을 수행할 수 있습니다.

```
#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)
```

다음 쿼리를 실행합니다.

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
        id
      name
      performers
    }

    ... on Concert {
      id
      name
      performingBand
    }

    ... on Conference {
      speakers
```

```
      workshops
    }
  }
}
```

위 쿼리는 다음과 같은 결과를 반환합니다.

```
{
  "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"
        ]
      }
    ]
  }
}
```

형식 확인 로직은 애플리케이션에 따라 다릅니다. 예를 들어, 특정 필드 또는 필드 조합이 있는지 검사하는 다른 식별 로직이 있을 수 있습니다. 즉, `Festival`을 식별하기 위해 `performers` 필드가 있는지

감지하거나 Conference를 식별하기 위해 speakers 및 workshops 필드의 조합이 있는지 감지할 수 있습니다. 궁극적으로 사용하려는 로직은 사용자가 정의하는 것입니다.

문제 해결 및 일반적인 실수

이 단원에서는 몇 가지 일반적인 오류와 이러한 오류를 해결하는 방법을 설명합니다.

잘못된 DynamoDB 키 매핑

GraphQL 작업에서 다음 오류 메시지를 반환하면 요청 매핑 템플릿 구조가 Amazon DynamoDB 키 구조와 일치하지 않기 때문일 수 있습니다.

```
The provided key element does not match the schema (Service: AmazonDynamoDBv2; Status
 Code: 400; Error Code
```

예를 들어, DynamoDB 테이블에 `"id"`라는 해시 키가 있고 다음 예에서처럼 템플릿에 `"PostID"`가 지정되어 있는데 `"id"`가 `"PostID"`와 일치하지 않기 때문에 위와 같은 오류가 발생합니다.

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "PostID" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

해석기 누락

쿼리 등과 같은 GraphQL 작업을 실행했는데 null 응답을 가져온 경우 해석기를 구성하지 않은 것이 원인일 수 있습니다.

예를 들어 `getCustomer(userId: ID!):` 필드를 정의하는 스키마를 가져왔는데 이 필드에 대해 해석기를 구성하지 않은 경우 `getCustomer(userId:"ID123"){...}` 등과 같은 쿼리를 실행하면 다음과 같은 응답을 얻습니다.

```
{
    "data": {
    "getCustomer": null
    }
}
```

매핑 템플릿 오류

매핑 템플릿이 제대로 구성되어 있지 않은 경우 `errorType`이 `MappingTemplate`인 GraphQL 응답이 수신됩니다. `message` 필드는 매핑 템플릿에서 문제가 발생한 지점을 나타내야 합니다.

예를 들어, 요청 매핑 템플릿에 `operation` 필드가 없거나 `operation` 필드 이름이 잘못된 경우 다음과 같은 응답을 얻을 수 있습니다.

```
{
    "data": {
        "searchPosts": null
    },
    "errors": [
        {
        "path": [
            "searchPosts"
        ],
        "errorType": "MappingTemplate",
        "locations": [
            {
            "line": 2,
            "column": 3
            }
        ],
        "message": "Value for field '$[operation]' not found."
        }
    ]
}
```

잘못된 반환 형식

데이터 원본의 반환 형식은 스키마에 정의한 객체 형식과 일치해야 합니다. 그렇지 않으면 다음과 같은 GraphQL 오류가 표시될 수 있습니다.

```
"errors": [
    {
    "path": [
        "posts"
    ],
    "locations": null,
```

```
    "message": "Can't resolve value (/posts) : type mismatch error, expected type LIST,
 got OBJECT"
    }
]
```

예를 들어, 쿼리 정의가 다음과 같은 경우 이러한 오류가 발생할 수 있습니다.

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

이 쿼리 정의는 [Posts] 객체 목록을 반환해야 합니다. 예를 들어, 다음과 같은 쿼리 정의가 포함된 Node.JS로 된 Lambda 함수가 있는 경우:

```
const result = { data: data.Items.map(item => { return item ; }) };
callback(err, result);
```

result가 객체이기 때문에 오류가 발생할 수 있습니다. 콜백을 result.data로 변경하거나 LIST를 반환하지 않도록 스키마를 변경해야 할 수 있습니다.

잘못된 요청 처리

잘못된 구문과 같은 잘못된 데이터로 인해 요청을 처리하고 필드 리졸버로 보낼 수 없는 경우 AWS AppSync 응답 페이로드는 값이 로 설정된 필드 null 데이터와 관련 오류가 있는 필드 데이터를 반환합니다.

기계 번역으로 제공되는 번역입니다. 제공된 번역과 원본 영어의 내용이 상충하는 경우에는 영어 버전이 우선합니다.