

---

# AWS AppSync

## Developer Guide

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

# Table of Contents

What is AWS AppSync?	1
Features of AWS AppSync	1
Are you a first-time AWS AppSync user?	1
Related services	1
Pricing for AWS AppSync	1
Quick Start	3
Launch a Sample Schema	3
Launch a Sample Schema	3
Taking a Tour of the Console	3
Schema Designer	4
Functions and Data Sources	4
Settings	4
Integration	4
Run Queries and Mutations	5
Add Data with a GraphQL Mutation	5
Retrieve Data with a GraphQL Query	5
Running an Application	6
Next Steps	6
System Overview and Architecture	7
Architecture	7
Concepts	7
GraphQL Proxy	7
Operation	7
Action	7
Data Source	7
Resolver	8
Function	8
Identity	8
AWS AppSync Client	8
GraphQL Overview	9
Designing a GraphQL API	11
Designing Your Schema	12
Creating an Empty Schema	12
Adding a Root Query Type	12
Defining a Todo Type	12
Adding a Mutation Type	13
Modifying the Todo Example with a Status	13
Subscriptions	14
Further Reading	14
Advanced - Relations and Pagination	14
Scalar types	16
Interfaces and unions in GraphQL	19
Attaching a Data Source	26
(Recommended) Automatic Provision	26
Adding a Data Source	27
Configuring Resolvers	29
Create Your First Resolver	29
Adding a Resolver for Mutations	29
Advanced Resolvers	30
Direct Lambda Resolvers	31
Test and Debug Resolvers	33
Pipeline Resolvers	35
Using Your API	40
Relations and Pagination	40

Next Steps .....	6
(Optional) Guided Schema Wizard .....	42
Create API .....	42
Populate Model .....	42
Execute GraphQL .....	42
Integrate with the App .....	42
(Optional) Provision from Schema .....	43
No Schema .....	43
Existing Schema .....	44
(Optional) Import from Amazon DynamoDB .....	46
Import a DynamoDB Table .....	46
Example Schema from Import .....	47
Building a Client App .....	50
Resolver tutorials .....	51
Tutorial: DynamoDB Resolvers .....	51
Setting Up Your DynamoDB Tables .....	52
Creating Your GraphQL API .....	52
Defining a Basic Post API .....	52
Configuring the Data Source for the DynamoDB Tables .....	53
Setting Up the addPost resolver (DynamoDB PutItem) .....	53
Setting Up the getPost Resolver (DynamoDB GetItem) .....	56
Create an updatePost Mutation (DynamoDB UpdateItem) .....	58
Modifying the updatePost Resolver (DynamoDB UpdateItem) .....	60
Create upvotePost and downvotePost Mutations (DynamoDB UpdateItem) .....	64
Setting Up the deletePost Resolver (DynamoDB DeleteItem) .....	67
Setting Up the allPost Resolver (DynamoDB Scan) .....	72
Setting Up the allPostsByAuthor Resolver (DynamoDB Query) .....	75
Using Sets .....	79
Using Lists and Maps .....	84
Conclusion .....	86
Tutorial: Lambda resolvers .....	86
Create a Lambda function .....	87
Configure a data source for Lambda .....	88
Create a GraphQL schema .....	89
Configure resolvers .....	29
Test your GraphQL API .....	91
Returning errors .....	92
Advanced use case: Batching .....	94
Tutorial: Amazon OpenSearch Service Resolvers .....	100
One-Click Setup .....	101
Create a New OpenSearch Service Domain .....	101
Configure Data Source for OpenSearch Service .....	101
Connecting a Resolver .....	102
Modifying Your Searches .....	104
Adding Data to OpenSearch Service .....	104
Retrieving a Single Document .....	105
Perform Queries and Mutations .....	106
Best Practices .....	106
Tutorial: Local Resolvers .....	106
Create the Paging Application .....	107
Send and subscribe to pages .....	107
Tutorial: Combining GraphQL Resolvers .....	108
Example Schema .....	108
Alter Data Through Resolvers .....	109
DynamoDB and OpenSearch Service .....	109
Tutorial: DynamoDB Batch Resolvers .....	112
Permissions .....	112

Data Source .....	7
Single Table Batch .....	113
Multi-Table Batch .....	116
Error Handling .....	121
Tutorial: DynamoDB Transaction Resolvers .....	125
Permissions .....	112
Data Source .....	7
Transactions .....	126
Tutorial: HTTP Resolvers .....	133
One-Click Setup .....	101
Creating a REST API .....	133
Creating Your GraphQL API .....	52
Creating a GraphQL Schema .....	89
Configure Your HTTP Data Source .....	135
Configuring Resolvers .....	29
Invoking AWS Services .....	137
Tutorial: Aurora Serverless .....	138
Create cluster .....	138
Enable Data API .....	139
Create database and table .....	139
GraphQL schema .....	139
Configuring Resolvers .....	29
Run mutations .....	143
Run Queries .....	144
Input Sanitization .....	145
Tutorial: Pipeline Resolvers .....	146
One-Click Setup .....	101
Manual Setup .....	147
Testing Your GraphQL API .....	91
Tutorial: Delta Sync .....	156
One-Click Setup .....	101
Schema .....	157
Mutations .....	158
Sync Queries .....	158
Example .....	159
Real-time data .....	164
GraphQL schema subscription directives .....	164
Using subscription arguments .....	166
Argument null value has meaning .....	167
Creating generic pub/sub APIs powered by serverless WebSockets .....	168
Using pub-sub APIs .....	168
Implementing pub-sub APIs into existing applications .....	170
Enhanced subscriptions filtering .....	170
Defining subscriptions .....	170
Creating enhanced subscription filters .....	171
Defining enhanced filters from the client .....	174
Unsubscribing connections .....	175
Using subscription invalidation .....	175
Using context variables in subscription invalidation filters .....	178
Building a real-time WebSocket client .....	178
Real-time WebSocket client implementation for GraphQL subscriptions .....	178
Handshake details to establish the WebSocket connection .....	180
Header parameter format based on AWS AppSync API authorization mode .....	181
Real-time WebSocket operation .....	185
Connection init message .....	185
Connection acknowledge message .....	185
Keep-alive message .....	185

Subscription registration message .....	186
Subscription acknowledgment message .....	188
Error message .....	188
Processing data messages .....	188
Subscription unregistration message .....	189
Disconnecting the WebSocket .....	189
Configuration and settings .....	190
Caching .....	190
Instance Types .....	190
Caching behavior .....	191
Cache encryption .....	191
Cache eviction .....	192
Evicting a cache entry .....	192
Evicting a cache entry based on identity .....	193
Configuring custom domain names .....	194
Registering and configuring a domain name .....	195
Creating a custom domain name in AWS AppSync .....	195
Wildcard custom domain names in AWS AppSync .....	196
Conflict Detection and Sync .....	196
Versioned Data Sources .....	196
Conflict Detection and Resolution .....	198
Sync Operations .....	205
Monitoring and logging .....	205
Setup and configuration .....	205
CloudWatch metrics .....	206
CloudWatch Logs .....	209
Log type reference .....	212
Analyzing your logs with CloudWatch Logs Insights .....	214
Analyze your logs with OpenSearch Service .....	215
Log format migration .....	215
Tracing with AWS X-Ray .....	215
Setup and Configuration .....	205
Tracing Your API with X-Ray .....	216
Logging AWS AppSync API calls using AWS CloudTrail .....	217
AWS AppSync information in CloudTrail .....	218
Understanding AWS AppSync log file entries .....	218
Authorization and Authentication .....	221
API_KEY Authorization .....	222
AWS_LAMBDA Authorization .....	222
AWS_IAM Authorization .....	225
OPENID_CONNECT Authorization .....	227
AMAZON_COGNITO_USER_POOLS Authorization .....	227
Using Additional Authorization Modes .....	228
Fine-Grained Access Control .....	230
Filtering Information .....	232
Data source access .....	232
Authorization Use Cases .....	233
Overview .....	233
Reading Data .....	233
Writing Data .....	236
Public and Private Records .....	238
Real-Time Data .....	238
Using AWS WAF to protect APIs .....	241
Integrate an AppSync API with AWS WAF .....	241
Creating rules for a web ACL .....	242
Security .....	245
Data Protection .....	245

Encryption in motion .....	246
Compliance Validation .....	246
Infrastructure Security .....	247
Resilience .....	247
Identity and access management .....	247
Audience .....	248
Authenticating with identities .....	248
Managing access using policies .....	250
How AWS AppSync works with IAM .....	251
Identity-based policies .....	256
Troubleshooting .....	257
Logging AWS AppSync API calls with AWS CloudTrail .....	259
AWS AppSync information in CloudTrail .....	259
Understanding AWS AppSync log file entries .....	260
Best practices .....	106
Understand authentication methods .....	262
Use TLS for HTTP resolvers .....	262
Use roles with the least permissions possible .....	262
IAM policy best practices .....	262
Resolver Mapping Template Reference .....	264
Resolver mapping template overview .....	264
Unit Resolvers .....	264
Pipeline resolvers .....	35
Example template .....	265
Evaluated mapping template deserialization rules .....	267
Resolver Mapping Template Programming Guide .....	268
Setup .....	268
Variables .....	269
Calling Methods .....	270
Strings .....	271
Loops .....	272
Arrays .....	272
Conditional Checks .....	273
Operators .....	274
Context .....	275
Filtering .....	275
Resolver mapping template context reference .....	279
Accessing the \$context .....	279
Sanitizing inputs .....	285
Resolver mapping template utility reference .....	286
Utility helpers in \$util .....	286
AWS AppSync directives .....	290
Time helpers in \$util.time .....	290
List helpers in \$util.list .....	292
Map helpers in \$util.map .....	292
DynamoDB helpers in \$util.dynamodb .....	293
Amazon RDS helpers in \$util.rds .....	298
HTTP helpers in \$util.http .....	300
XML helpers in \$util.xml .....	301
Transformation helpers in \$util.transform .....	302
Math helpers in \$util.math .....	311
String helpers in \$util.str .....	312
Extensions .....	312
Resolver Mapping Template Reference for DynamoDB .....	321
GetItem .....	321
PutItem .....	323
UpdateItem .....	325

DeleteItem .....	328
Query .....	330
Scan .....	333
Sync .....	336
BatchGetItem .....	338
BatchDeleteItem .....	340
BatchPutItem .....	342
TransactGetItems .....	345
TransactWriteItems .....	347
Type System (Request Mapping) .....	352
Type System (Response Mapping) .....	356
Filters .....	359
Condition Expressions .....	360
Transaction Condition Expressions .....	368
Resolver mapping template reference for RDS .....	370
Request mapping template .....	370
Version .....	371
Statements .....	371
VariableMap .....	371
Resolver Mapping Template Reference for OpenSearch .....	372
Request Mapping Template .....	370
Response Mapping Template .....	373
operation field .....	373
path field .....	373
params field .....	374
Passing Variables .....	375
Resolver mapping template reference for Lambda .....	375
Request mapping template .....	370
Response mapping template .....	373
Lambda function batched response .....	378
Direct Lambda Resolvers .....	378
Resolver mapping template reference for None data source .....	382
Request mapping template .....	370
Version .....	371
Payload .....	378
Response mapping template .....	373
Resolver Mapping Template Reference for HTTP .....	384
Request Mapping Template .....	370
Version .....	371
Method .....	386
ResourcePath .....	386
Params Field .....	374
Certificate Authorities (CA) Recognized by AWS AppSync for HTTPS Endpoints .....	387
Resolver mapping template changelog .....	416
Datasource Operation Availability Per Version Matrix .....	416
Changing the Version on a Unit Resolver Mapping Template .....	417
Changing the Version on a Function .....	418
2018-05-29 .....	418
2017-02-28 .....	423
Troubleshooting and Common Mistakes .....	424
Incorrect DynamoDB Key Mapping .....	424
Missing Resolver .....	424
Mapping Template Errors .....	424
Incorrect Return Types .....	425



# What is AWS AppSync?

AWS AppSync provides a robust, scalable GraphQL interface for application developers to combine data from multiple sources, including Amazon DynamoDB, AWS Lambda, and HTTP APIs.

## Topics

- [Features of AWS AppSync \(p. 1\)](#)
- [Are you a first-time AWS AppSync user? \(p. 1\)](#)
- [Related services \(p. 1\)](#)
- [Pricing for AWS AppSync \(p. 1\)](#)

## Features of AWS AppSync

AWS AppSync includes a variety of features to make building GraphQL a streamlined experience:

- Powerful GraphQL schema editing through the AWS AppSync console, including automatic GraphQL schema generation from DynamoDB
- Efficient data caching
- Integration with Amazon Cognito user pools for fine-grained access control at a per-field level

## Are you a first-time AWS AppSync user?

We recommend that first-time AWS AppSync users begin by reading the following sections:

- If you're unfamiliar with GraphQL, see [Quick Start \(p. 3\)](#).
- If you're building applications that consume GraphQL APIs, see [Building a Client App \(p. 50\)](#) and [Real-time data \(p. 164\)](#).
- If you're looking for GraphQL resolver information, see the [Resolver Mapping Template Reference \(p. 264\)](#).

## Related services

If you're building a web or mobile app from the ground up, consider using [AWS Amplify](#). Amplify leverages AWS AppSync and other AWS services to help you build more robust, powerful web and mobile apps with less work.

## Pricing for AWS AppSync

AWS AppSync is priced based on millions of requests and updates. Caching costs an additional fee. For more information, see [AWS AppSync pricing](#).

The following lists the exceptions to general AWS AppSync pricing:

- API caching in AWS AppSync is not eligible for the [AWS Free Tier](#).

- Requests are not charged for authorization and authentication failures.
- Calls to methods that require API keys are not charged when API keys are missing or invalid.

# Quick Start

This section describes how to use the AWS AppSync console to do the following:

- Launch a sample event app schema
- Automatically provision a DynamoDB data source and connect resolvers
- Write GraphQL queries and mutations
- Use the API in the sample app

AppSync provides a [guided schema creation wizard \(p. 42\)](#) that is recommended for first-time users who have never used GraphQL or AppSync before. Alternatively, you can [import from an existing Amazon DynamoDB table to create a real-time and offline GraphQL API \(p. 46\)](#) or [build the entire backend with or without a pre-existing schema \(p. 43\)](#).

You can also get started with AWS AppSync by writing a GraphQL schema from scratch, which can automatically provision and connect to a database for you. For more information, see [Designing Your Schema \(p. 12\)](#).

## Topics

- [Launch a Sample Schema \(p. 3\)](#)
- [Run Queries and Mutations \(p. 5\)](#)

## Launch a Sample Schema

This section describes how to use the AWS AppSync console to launch a sample schema and create and configure a GraphQL API.

### Launch a Sample Schema

The Event App sample schema enables users to create an application where events (for example, “Going to the movies” or “Dinner at Mom & Dad’s”) can be entered. Application users can also comment on the events (for example, “See you at 7!”). This app demonstrates how to use GraphQL operations where state is persisted in Amazon DynamoDB.

To get started, you need to create a sample schema and provision it.

#### To create the API

1. Sign in to the AWS Management Console and open the [AppSync console](#).
  - In the **Dashboard**, choose **Create API**.
2. In the bottom section list of samples, verify that **Event App** is selected. If it isn’t, select it. and choose **Start**.
3. Enter a friendly name for your API.
4. Choose **Create** and then wait for the provisioning process to finish.

### Taking a Tour of the Console

After your schema is deployed and your resources are provisioned, you can use the GraphQL API in the AWS AppSync console. The first page you see is **Queries**, which contains basic queries and mutations.

The **Queries** page is a built-in designer included in the console for writing and running GraphQL queries and mutations, including introspection and documentation. That's covered in [Run Queries and Mutations \(p. 5\)](#).

## Schema Designer

On the left side of the console, choose **Schema** to view the designer. The designer has your sample **Events** schema loaded. The code editor has linting and error checking capabilities that you can use when you write your own apps.

The right side of the console shows the GraphQL types that have been created and resolvers on different top-level types, such as queries. When adding new types to a schema (for example, `type TODO { ... }`), you can have AWS AppSync provision DynamoDB resources for you. These include the proper primary key, sort key, and index design to best match your GraphQL data access pattern. If you choose **Create Resources** at the top and choose one of these user-defined types from the menu, you can choose different field options in the schema design. Don't select anything now, but try this in the future when you [design a schema \(p. 12\)](#).

## Resolver Configuration

In the schema designer, choose one of the **resolvers** on the right, next to a field. A new page opens. This page shows the configured data source (the **Data Sources** tab of the console has a complete list) for a resolver, and the associated **Request** and **Response Mapping Template** designers. Sample mapping templates are provided for common use cases. This is also where you can configure custom logic for things such as parsing arguments from a GraphQL request, pagination token responses to clients, and custom query messages to Amazon OpenSearch Service.

## Functions and Data Sources

On the main navigation, choose **Functions**. A list of functions and the data sources that they're attached to are listed. Functions are single operations that you can run against a data source. Functions consist of a **Request Mapping Template** and a **Response Mapping Template**. The resolvers presented earlier in the sample schema are known as **Unit** resolvers. They're attached directly to a data source. You can also create a pipeline resolver, which contains one or more functions that are run in sequence on your GraphQL fields. Pipeline resolvers enable you to run operations against one or more data sources in a single network request. You can reuse logic across different resolvers, and mix or match data sources in a single resolver for different use cases (for example, authorization or data aggregation). For more information, see [Pipeline Resolvers \(p. 35\)](#).

## Settings

On the **Settings** tab, you can configure the authorization method for your API. For more information about these options, see [Security \(p. 221\)](#).

**Note:** The default authorization mode, `API_KEY`, uses an API key to test the application. However, for production GraphQL APIs, you should use one of the stronger authorization modes, such as AWS Identity and Access Management with Amazon Cognito identity pools or user pools. For more information, see [Security \(p. 221\)](#).

## Integration

The **Integration** page, summarizes the steps for setting up your API and outlines the next steps for building a client application. To locate this information, select the name of your AWS AppSync app in the navigation pane on the left side of the AWS AppSync console. In the **Getting Started** pane, find the section called **Integrate with your app**. This page provides details for using the [AWS Amplify toolchain](#)

to automate the process of connecting your API with iOS, Android, and JavaScript applications through config and code generation. The Amplify toolchain provides full support for building projects from your local workstation, including GraphQL provisioning and workflows for CI/CD.

This page also lists sample client applications (for example, JavaScript, iOS, etc.) for testing an end-to-end experience. You can clone and download these samples, and the configuration file that has the necessary information (such as your endpoint URL) you need to get started. Follow the instructions on [AWS Amplify toolchain](#) to run your app.

## Run Queries and Mutations

Now that you have taken a tour of the console it's time to get more familiar with GraphQL. In the AWS AppSync console choose the **Queries** tab on the left hand side. The pane on the right side enables you to click through the operations, including queries, mutations, and subscriptions that your schema has exposed. Choose the **Mutation** node to see a mutation. You can add a new event to it as follows: `createEvent( ... ):Event`. Use this to add something to your database with GraphQL.

### Add Data with a GraphQL Mutation

Your first step is to add more data with a GraphQL mutation. To do this, you use the `mutation` keyword and pass in the appropriate arguments (similar to how a function works). You can also select which data you want returned in the response by putting the fields inside curly braces. To get started, copy the following into the query editor and then choose **Run**:

```
mutation add {
  createEvent(
    name:"My first GraphQL event"
    where:"Day 1"
    when:"Friday night"
    description:"Catching up with friends"
  ){
    id
    name
    where
    when
    description
  }
}
```

The record is parsed by the GraphQL engine and inserted into your Amazon DynamoDB table by a resolver that is connected to a data source. (You can check this in the DynamoDB console.) Notice that you didn't need to pass in an `id`. An `id` is generated and returned in the results specified between the curly braces. As a best practice, this sample uses an `autoId( )` function in a GraphQL resolver for the partition key set on your DynamoDB resources. For now, just make a note of the returned `id` value, which you'll use in the next section.

### Retrieve Data with a GraphQL Query

Now that there is a record in your database, you'll get results when you run a query. One of the main advantages of GraphQL is the ability to specify the exact data requirements that your application has in a query. This time, only add a few of the fields inside the curly braces, pass the `id` argument to `getEvent( )`, and then choose **Run** at the top:

```
query get {
  getEvent(id:"XXXXXX-XXXX-XXXXXXX-XXXX-XXXXXXXXXX"){
```

```
    name
    where
    description
  }
}
```

This time, only the fields you specified are returned. You can also try listing all events as follows:

```
query getAllEvents {
  listEvents{
    items{
      id
      name
      when
    }
  }
}
```

This time the query shows nested types and gives the query a friendly name (`getAllEvents`), which is optional. Experiment by adding or removing, and then running the query again. When you're done, it's time to [connect a client application \(p. 50\)](#).

## Running an Application

Now that your API is working, you can use a client application to interact with it. AWS AppSync provides samples in several programming languages to get you started. Go to the **Integration** page at the root of the console navigation on the left by selecting the **name of your API**, and you'll see a list of platforms. At the bottom clone the appropriate sample for the **Event app** to your local workstation, download the configuration file and, if necessary, use the Amplify CLI in the instructions to perform code generation with your API ID. The configuration file contains details, such as the endpoint URL of your GraphQL API and the API key, to include when getting started. You can change this information later when leveraging IAM or Amazon Cognito user pools in production. For more information, see [Security \(p. 221\)](#).

## Next Steps

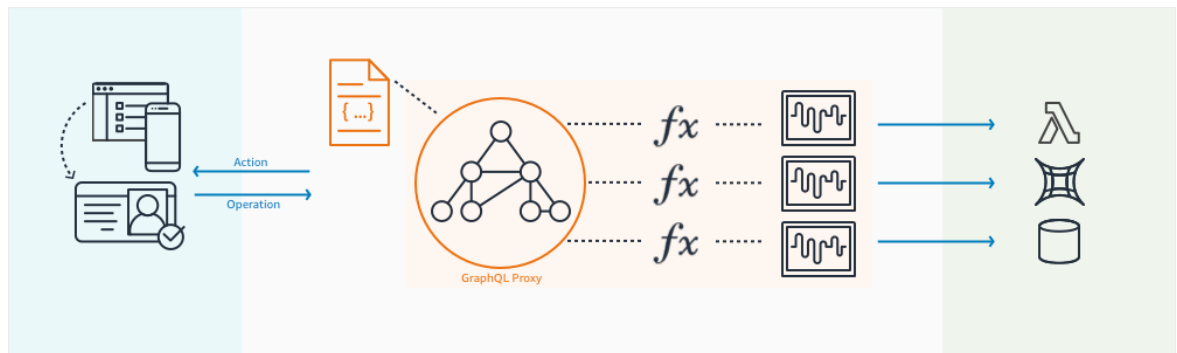
Now that you've run through the preconfigured schema, you can choose to build an API from scratch, incorporate an existing data source, or build a client application. For more information, see the following sections:

- [Designing a GraphQL API \(p. 11\)](#)
- [Connecting Data Sources and Resolvers \(p. 51\)](#)
- [Building Client Applications \(p. 50\)](#)

# System Overview and Architecture

AWS AppSync enables developers to interact with their data by using a managed GraphQL service. GraphQL offers many benefits over traditional gateways, encourages declarative coding style, and works seamlessly with modern tools and frameworks, including React, React Native, iOS, and Android.

## Architecture



## Concepts

### GraphQL Proxy

A component that runs the GraphQL engine for processing requests and mapping them to logical functions for data operations or triggers. The data resolution process performs a batching process (called the Data Loader) to your data sources. This component also manages conflict detection and resolution strategies.

### Operation

AWS AppSync supports the three GraphQL operations: query (read-only fetch), mutation (write followed by a fetch), and subscription (long-lived requests that receive data in response to events).

### Action

There is one action that AWS AppSync defines. This action is a notification to connected subscribers, which is the result of a mutation. Clients become subscribers through a handshake process following a GraphQL subscription.

### Data Source

A persistent storage system or a trigger, along with credentials for accessing that system or trigger. Your application state is managed by the system or trigger defined in a data source. Examples of data sources include NoSQL databases, relational databases, AWS Lambda functions, and HTTP APIs.

## Resolver

A function that converts the GraphQL payload to the underlying storage system protocol and executes if the caller is authorized to invoke it. Resolvers are comprised of request and response mapping templates, which contain transformation and execution logic.

### Unit Resolver

A unit resolver is a resolver that performs a single operation against a predefined data source.

### Pipeline Resolver

A pipeline resolver is a resolver that allows executing multiple operations against one or more data sources. A pipeline resolver is composed of a list of functions. Each function is executed in sequence and can execute a single operation against a predefined data source.

## Function

A function defines a single operation that can be used across pipeline resolvers. Functions can be reused to perform redundant logic throughout the GraphQL Proxy. Functions are comprised of a request and a response mapping template, a data source name, and a version.

## Identity

A representation of the caller based on a set of credentials, which must be sent along with every request to the GraphQL proxy. It includes permissions to invoke resolvers. Identity information is also passed as context to a resolver and the conflict handler to perform additional checks.

## AWS AppSync Client

The location where GraphQL operations are defined. The client performs appropriate authorization wrapping of request statements before submitting to the GraphQL proxy. Responses are persisted in an offline store and mutations are made in a write-through pattern.



# GraphQL Overview

[GraphQL](#) is a data language that was developed to enable apps to fetch data from servers. It has a declarative, self-documenting style. In a GraphQL operation, the client specifies how to structure the data when it is returned by the server. This makes it possible for the client to query only for the data it needs, in the format that it needs it in.

GraphQL has three top-level operations:

- **Query** - read-only fetch
- **Mutation** - write, followed by a fetch
- **Subscription** - long-lived connection for receiving data

GraphQL exposes these operations via a schema that defines the capabilities of an API. A schema is comprised of types, which can be root types (query, mutation, or subscription) or user-defined types. Developers start with a schema to define the capabilities of their GraphQL API, which a client application will communicate with. For more information about this process, see [Designing Your Schema \(p. 12\)](#).

After a schema is defined, the fields on a type need to return some data. The way this happens in a GraphQL API is through a GraphQL resolver. This is a function that either calls out to a data source or invokes a trigger to return some value (such as an individual record or a list of records). Resolvers can have many types of data sources, such as NoSQL databases, relational databases, or search engines. You can aggregate data from multiple data sources and return identical types, mixing and matching to meet your needs.

After a schema is connected to a resolver function, a client app can issue a GraphQL query or, optionally, a mutation or subscription. A query will have the `query` keyword followed by curly braces, and then the field name, such as `allPosts`. After the field name is a second set of curly braces with the data that you want to return. For example:

```
query {  
  allPosts {  
    id  
    author  
    title  
    content  
  }  
}
```

This query invokes a resolver function against the `allPosts` field and returns just the `id`, `author`, `title`, and `content` values. If there were many posts in the system (assuming that `allPosts` return blog posts, for example), this would happen in a single network call. Though designs can vary, in traditional systems, this is usually modeled in separate network calls for each post. This reduction in network calls reduces bandwidth requirements and therefore saves battery life and CPU cycles consumed by client applications.

These capabilities make prototyping new applications, and modifying existing applications, very fast. A benefit of this is that the application's data requirements are colocated in the application with the UI code for your programming language of choice. This enables client and backend teams to work independently, instead of encoding data modeling on backend implementations.

Finally, the type system provides powerful mechanisms for pagination, relations, inheritance, and interfaces. You can relate different types between separate NoSQL tables when using the GraphQL type system.

For further reading, see the following resources:

- [GraphQL](#)
- [Designing a GraphQL API \(p. 11\)](#)
- [Data Sources and Resolvers Tutorial \(p. 51\)](#)

# Designing a GraphQL API

If you are building a GraphQL API, there are some concepts you need to know, such as schema design and how to connect to data sources.

In this section, we show you how to build a schema from scratch, provision resources automatically, manually define a data source, and connect to it with a GraphQL resolver. AWS AppSync can also [build out a schema and resolvers from scratch](#), if you have an existing [Amazon DynamoDB table \(p. 46\)](#) or [build the entire backend with or without a pre-existing schema \(p. 43\)](#). Additionally, AWS AppSync provides a [guided schema creation wizard \(p. 42\)](#) that is recommended for first-time GraphQL or AWS AppSync users.

## GraphQL Schema

Each GraphQL API is defined by a single GraphQL schema. The GraphQL Type system describes the capabilities of a GraphQL server and is used to determine if a query is valid. A server's type system is referred to as that server's schema. It is made up of a set of object types, scalars, input types, interfaces, enums, and unions. It defines the shape of the data that flows through your API and also the operations that can be performed. GraphQL is a strongly typed protocol and all data operations are validated against this schema.

## Data Source

Data sources are resources in your AWS account that GraphQL APIs can interact with. AWS AppSync supports AWS Lambda, Amazon DynamoDB, relational databases (Amazon Aurora Serverless), Amazon OpenSearch Service, and HTTP endpoints as data sources.

An AWS AppSync API can be configured to interact with multiple data sources, enabling you to aggregate data in a single location. AWS AppSync can use AWS resources from your account that already exist or can provision DynamoDB tables on your behalf from a schema definition.

## Resolvers

GraphQL resolvers connect the fields in a type's schema to a data source. Resolvers are the mechanism by which requests are fulfilled. AWS AppSync can [automatically create and connect resolvers \(p. 43\)](#) from a schema or [create a schema and connect resolvers from an existing table \(p. 46\)](#) without you needing to write any code.

Resolvers in AWS AppSync use mapping templates written in [Apache Velocity Template Language \(VTL\)](#) to convert a GraphQL expression into a format the data source can use.

An introductory tutorial-style programming guide for writing resolvers can be found in [Resolver Mapping Template Programming Guide \(p. 268\)](#) and helper utilities available to use when programming can be found in [Resolver Mapping Template Context Reference \(p. 279\)](#). AWS AppSync also has built-in test and debug flows that you can use when you're editing or authoring from scratch. For more information, see [Test and Debug Resolvers \(p. 33\)](#).

## Topics

- [Designing Your Schema \(p. 12\)](#)
- [Attaching a Data Source \(p. 26\)](#)
- [Configuring Resolvers \(p. 29\)](#)
- [Using Your API \(p. 40\)](#)
- [\(Optional\) Guided Schema Wizard \(p. 42\)](#)
- [\(Optional\) Provision from Schema \(p. 43\)](#)
- [\(Optional\) Import from Amazon DynamoDB \(p. 46\)](#)

# Designing Your Schema

## Creating an Empty Schema

Schema files are text files, usually named `schema.graphql`. You can create this file and submit it to AWS AppSync by using the CLI or navigating to the console and adding the following under the **Schema** page:

```
schema {  
}
```

Every schema has this root for processing. This fails to process until you add a root query type.

## Adding a Root Query Type

For this example, we create a `Todo` application. A GraphQL schema must have a root query type, so we add a root type named `Query` with a single `getTodos` field that returns a list containing `Todo` objects. Add the following to your `schema.graphql` file:

```
schema {  
  query: Query  
}  
  
type Query {  
  getTodos: [Todo]  
}
```

Notice that we haven't yet defined the `Todo` object type. Let's do that now.

## Defining a Todo Type

Now, create a type that contains the data for a `Todo` object:

```
schema {  
  query: Query  
}  
  
type Query {  
  getTodos: [Todo]  
}  
  
type Todo {  
  id: ID!  
  name: String  
  description: String  
  priority: Int  
}
```

Notice that the `Todo` object type has fields that are [scalar types](#), such as strings and integers. AWS AppSync also has enhanced [Scalar types in AWS AppSync \(p. 16\)](#) in addition to the base GraphQL scalars that you can use in a schema. Any field that ends in an exclamation point is a required field. The ID scalar type is a unique identifier that can be either `String` or `Int`. You can control these in your resolver mapping templates for automatic assignment, which is covered later.

There are similarities between the `Query` and `Todo` types. In GraphQL, the root types (that is, `Query`, `Mutation`, and `Subscription`) are similar to the ones you define, but they're different in that you

expose them from your schema as the entry point for your API. For more information, see [The Query and Mutation types](#).

## Adding a Mutation Type

Now that you have an object type and can query the data, if you want to add, update, or delete data via the API you need to add a mutation type to your schema. For the `Todo` example, add this as a field named `addTodo` on a mutation type:

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

type Query {
  getTodos: [Todo]
}

type Mutation {
  addTodo(id: ID!, name: String, description: String, priority: Int): Todo
}

type Todo {
  id: ID!
  name: String
  description: String
  priority: Int
}
```

Note that the mutation is also added to this schema type because it is a root type.

## Modifying the Todo Example with a Status

At this point, your GraphQL API is functioning structurally for reading and writing `Todo` objects—it just doesn't have a data source, which is described in the next section. You can modify this API with more advanced functionality, such as adding a status to your `Todo`, which comes from a set of values defined as an `ENUM`:

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

type Query {
  getTodos: [Todo]
}

type Mutation {
  addTodo(id: ID!, name: String, description: String, priority: Int, status: TodoStatus):
  Todo
}

type Todo {
  id: ID!
  name: String
  description: String
  priority: Int
  status: TodoStatus
}
```

```
enum TodoStatus {  
  done  
  pending  
}
```

Choose **Save Schema**.

An `ENUM` is like a string, but it can take one of a set of values. In the previous example, you added this type, modified the `Todo` type, and added the `Todo` field to contain this functionality.

**Note:** You can skip directly to [Attaching a Data Source \(p. 26\)](#) and begin connecting your schema to a data source now, or keep reading to modify your schema with paginations and relational structure.

## Subscriptions

Subscriptions in AWS AppSync are invoked as a response to a mutation. You configure this with a `Subscription` type and `@aws_subscribe()` directive in the schema to denote which mutations invoke one or more subscriptions. For more information about configuring subscriptions, see [Real-Time Data \(p. 164\)](#).

## Further Reading

For more information, see the [GraphQL type system](#).

## Advanced - Relations and Pagination

Suppose you had a million `todos`. You wouldn't want to fetch all of these every time, instead you would want to paginate through them. Make the following changes to your schema:

- To the `getTodos` field, add two input arguments: `limit` and `nextToken`.
- Add a new `TodoConnection` type that has `todos` and `nextToken` fields.
- Change `getTodos` so that it returns `TodoConnection` (not a list of `Todos`).

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Query {  
  getTodos(limit: Int, nextToken: String): TodoConnection  
}  
  
type Mutation {  
  addTodo(id: ID!, name: String, description: String, priority: Int, status: TodoStatus):  
    Todo  
}  
  
type Todo {  
  id: ID!  
  name: String  
  description: String  
  priority: Int  
  status: TodoStatus  
}  
  
type TodoConnection {  
  todos: [Todo]
```

```
    nextToken: String
  }

enum TodoStatus {
  done
  pending
}
```

The `TodoConnection` type allows you to return a list of `todos` and a `nextToken` for getting the next batch of `todos`. Note that it is a single `TodoConnection` and not a list of connections. Inside the connection is a list of todo items (`[Todo]`) which gets returned with a pagination token. In AWS AppSync, this is connected to Amazon DynamoDB with a mapping template and automatically generated as an encrypted token. This converts the value of the `limit` argument to the `maxResults` parameter and the `nextToken` argument to the `exclusiveStartKey` parameter. For examples and the built-in template samples in the AWS AppSync console, see [Resolver Mapping Template Reference \(p. 264\)](#).

Next, suppose your `todos` have comments, and you want to run a query that returns all the comments for a `todo`. This is handled through GraphQL connections, as you created in the previous schema. Modify your schema to have a `Comment` type, add a `comments` field to the `Todo` type, and add an `addComment` field on the `Mutation` type as follows:

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

type Query {
  getTodos(limit: Int, nextToken: String): TodoConnection
}

type Mutation {
  addTodo(id: ID!, name: String, description: String, priority: Int, status:
  TodoStatus): Todo
  addComment(todoId: ID!, content: String): Comment
}

type Comment {
  todoId: ID!
  commentId: String!
  content: String
}

type Todo {
  id: ID!
  name: String
  description: String
  priority: Int
  status: TodoStatus
  comments: [Comment]
}

type TodoConnection {
  todos: [Todo]
  nextToken: String
}

enum TodoStatus {
  done
  pending
}
```

Choose **Save Schema**.

Note that the `Comment` type has the `todoId` that it's associated with, `commentId`, and `content`. This corresponds to a primary key + sort key combination in the Amazon DynamoDB table you create later.

The application graph on top of your existing data sources in AWS AppSync enables you to return data from two separate data sources in a single GraphQL query. In the example, the assumption is that there is both a `Todos` table and a `Comments` table. We'll show how to do this in [Attaching a Data Source](#) (p. 26) and [Configuring Resolvers](#) (p. 29).

## Scalar types in AWS AppSync

A GraphQL object type has a name and fields, and those fields can have sub-fields. Ultimately, an object type's fields must resolve to *scalar* types, which represent the leaves of the query. For more information about object types and scalars, see [Schemas and Types](#) on the GraphQL website.

In addition to a default set of GraphQL scalars, AWS AppSync includes a set of reserved scalars that start with an `AWS` prefix. AWS AppSync does not support custom scalars.

### Note

You cannot use `AWS` as a prefix for custom object types.

GraphQL defines the following default scalars:

`ID`

A unique identifier for an object. This scalar is serialized like a `String` but isn't meant to be human-readable.

`String`

A UTF-8 character sequence.

`Int`

An integer value between  $-(2^{31})$  and  $2^{31}-1$ .

`Float`

An IEEE 754 floating point value.

`Boolean`

A Boolean value, either `true` or `false`.

AWS AppSync defines the following scalars:

`AWSDate`

An extended [ISO 8601 date](#) string in the format `YYYY-MM-DD`.

`AWSTime`

An extended [ISO 8601 time](#) string in the format `hh:mm:ss.sss`.

`AWSDateTime`

An extended [ISO 8601 date and time](#) string in the format `YYYY-MM-DDThh:mm:ss.sssZ`.

### Note

The `AWSDate`, `AWSTime`, and `AWSDateTime` scalars can optionally include a [time zone offset](#). For example, the values `1970-01-01Z`, `1970-01-01-07:00`, and `1970-01-01+05:30` are all valid for `AWSDate`. The time zone offset must be either `Z` (UTC) or an offset in hours and minutes (and, optionally, seconds). For example, `±hh:mm:ss`. The seconds field in the time zone offset is considered valid even though it's not part of the ISO 8601 standard.



#### AWSTimestamp

An integer value representing the number of seconds before or after 1970-01-01-T00:00Z.

#### AWSEmail

An email address in the format `local-part@domain-part` as defined by [RFC 822](#).

#### AWSJSON

A JSON string. Any valid JSON construct is automatically parsed and loaded in the resolver mapping templates as maps, lists, or scalar values rather than as the literal input strings. Unquoted strings or otherwise invalid JSON result in a GraphQL validation error.

#### AWSPhone

A phone number. This value is stored as a string. Phone numbers can contain either spaces or hyphens to separate digit groups. Phone numbers without a country code are assumed to be US/ North American numbers adhering to the [North American Numbering Plan \(NANP\)](#).

#### AWSURL

A URL as defined by [RFC 1738](#). For example, `https://www.amazon.com/dp/B000NZW3KC/` or `mailto:example@example.com`. URLs must contain a schema (`http`, `mailto`) and can't contain two forward slashes (`//`) in the path part.

#### AWSIPAddress

A valid IPv4 or IPv6 address. IPv4 addresses are expected in quad-dotted notation (`123.12.34.56`). IPv6 addresses are expected in non-bracketed, colon-separated format (`1a2b:3c4b::1234:567`). You can include an optional CIDR suffix (`123.45.67.89/16`) to indicate subnet mask.

## Schema usage example

If you're unfamiliar with creating GraphQL APIs in AWS AppSync, or with connecting resolvers with mapping templates, we recommend first reviewing [Designing a GraphQL API \(p. 11\)](#) and [Resolver tutorials \(p. 51\)](#).

The following example GraphQL schema uses all of the custom scalars as an "object" and shows the resolver request and response templates for basic put, get, and list operations. Finally, the example shows how you can use this when running queries and mutations.

```
type Mutation {
  putObject(
    email: AWSEmail,
    json: AWSJSON,
    date: AWSDate,
    time: AWSTime,
    datetime: AWSDateTime,
    timestamp: AWSTimestamp,
    url: AWSURL,
    phoneno: AWSPhone,
    ip: AWSIPAddress
  ): Object
}

type Object {
  id: ID!
  email: AWSEmail
  json: AWSJSON
  date: AWSDate
  time: AWSTime
  datetime: AWSDateTime
  timestamp: AWSTimestamp
}
```

```
    url: AWSURL
    phoneno: AWSPhone
    ip: AWSIPAddress
  }

  type Query {
    getObject(id: ID!): Object
    listObjects: [Object]
  }

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

Use the following request template for `putObject`:

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

The response template for `putObject` is:

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

Use the following request template for `getObject`:

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

The response template for `getObject` is:

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

Use the following request template for `listObjects`:

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

The response template for `listObjects` is:

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

The following are some examples of using this schema with GraphQL queries:

```
mutation CreateObject {
```

```
    putObject(email: "example@example.com"
      json: "{\"a\":1, \"b\":3, \"string\": 234}"
      date: "1970-01-01Z"
      time: "12:00:34."
      datetime: "1930-01-01T16:00:00-07:00"
      timestamp: -123123
      url:"https://amazon.com"
      phoneno: "+1 555 764 4377"
      ip: "127.0.0.1/8"
    ) {
      id
      email
      json
      date
      time
      datetime
      url
      timestamp
      phoneno
      ip
    }
  }

  query getObject {
    getObject(id:"0d97daf0-48e6-4ffc-8d48-0537e8a843d2"){
      email
      url
      timestamp
      phoneno
      ip
    }
  }

  query listObjects {
    listObjects {
      json
      date
      time
      datetime
    }
  }
}
```

## Interfaces and unions in GraphQL

This topic provides an overview of interface and union types in GraphQL.

### Interfaces

The GraphQL type system supports [Interfaces](#). An interface exposes a certain set of fields that a type must include to implement the interface. If you are just getting started with GraphQL, you should return to this topic at a later time when you want to evolve your schema or add more features. To simplify getting started with GraphQL, learn how to [automatically create and connect resolvers \(p. 43\)](#) from a schema.

For example, we could represent an `Event` interface that represents any kind of activity or gathering of people. Possible kinds of events are `Concert`, `Conference`, and `Festival`. These types all share common characteristics, including a name, a venue where the event is taking place, and a start and end date. These types also have differences, a `Conference` offers a list of speakers and workshops, while a `Concert` features a performing band.

In Schema Definition Language (SDL), the `Event` interface is defined as follows:

```
interface Event {  
  id: ID!  
  name : String!  
  startsAt: String  
  endsAt: String  
  venue: Venue  
  minAgeRestriction: Int  
}
```

And each of the types implements the Event interface as follows:

```
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]  
}
```

Interfaces are useful to represent elements that might be of several types. For example, we could search for all events happening at a specific venue. Let's add a `findEventsByVenue` field to the schema as follows:

```
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` returns a list of `Event`. Because GraphQL interface fields are common to all the implementing types, it's possible to select any fields on the `Event` interface (`id`, `name`, `startsAt`, `endsAt`, `venue`, and `minAgeRestriction`). Additionally, you can access the fields on any implementing type using GraphQL [fragments](#), as long as you specify the type.

Let's examine an example of a GraphQL query that uses the interface.

```
query {
  findEventsAtVenue(venueId: "Madison Square Garden") {
    id
    name
    minAgeRestriction
    startsAt

    ... on Festival {
      performers
    }

    ... on Concert {
      performingBand
    }

    ... on Conference {
      speakers
    }
  }
}
```

```
      workshops
    }
  }
}
```

The previous query yields a single list of results, and the server could sort the events by start date by default.

```
{
  "data": {
    "findEventsAtVenue": [
      {
        "id": "Festival-2",
        "name": "Festival 2",
        "minAgeRestriction": 21,
        "startsAt": "2018-10-05T14:48:00.000Z",
        "performers": [
          "The Singers",
          "The Screammers"
        ]
      },
      {
        "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"
        ]
      }
    ]
  }
}
```

Since results are returned as a single collection of events, using interfaces to represent common characteristics is very helpful for sorting results.

## Unions

The GraphQL type system also supports [Unions](#). Unions are identical to interfaces, except that they don't define a common set of fields. Unions are generally preferred over interfaces when the possible types do not share a logical hierarchy.

For example, a search result might represent many different types. Using the `Event` schema, you can define a `SearchResult` union as follows:

```
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
```

In this case, to query any field on our `SearchResult` union, you must use fragments. Let's examine the following example:

```
query {  
  search(query: "Madison") {  
    ... on Venue {  
      id  
      name  
      address  
    }  
  
    ... on Festival {  
      id  
      name  
      performers  
    }  
  
    ... on Concert {  
      id  
      name  
      performingBand  
    }  
  
    ... on Conference {  
      speakers  
      workshops  
    }  
  }  
}
```

## Type resolution in AWS AppSync

Type resolution is the mechanism by which the GraphQL engine identifies a resolved value as a specific object type.

Coming back to the union search example, provided our query yielded results, each item in the results list must present itself as one of the possible types that the `SearchResult` union defined (that is, `Conference`, `Festival`, `Concert`, or `Venue`).

Because the logic to identify a `Festival` from a `Venue` or a `Conference` is dependent on the application requirements, the GraphQL engine must be given a hint to identify our possible types from the raw results.

With AWS AppSync, this hint is represented by a meta field named `__typename`, whose value corresponds to the identified object type name. `__typename` is required for return types that are interfaces or unions.

## Type resolution example

Let's reuse the previous schema. You can follow along by navigating to the console and adding the following under the **Schema** page:

```
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
}
```

Let's attach a resolver to the `Query.search` field. In the console, choose **Attach Resolver**, create a new **Data Source** of type *NONE*, and then name it *StubDataSource*. For the sake of this example, we'll pretend we fetched results from an external source, and hard code the fetched results in the request mapping template.

In the request mapping template pane, enter the following:

```
{
```



```
"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 Screammers"]
  },
  {
    "id": "Concert-3",
    "name": "Concert 3",
    "performingBand": "The Jumpers"
  },
  {
    "id": "Conference-4",
    "name": "Conference 4",
    "speakers": ["The Storytellers"],
    "workshops": ["Writing", "Reading"]
  }
]
```

If the application returns the type name as part of the `id` field, the type resolution logic must parse the `id` field to extract the type name and then add the `__typename` field to each of the results. You can perform that logic in the response mapping template as follows:

**Note:** You can also perform this task as part of your Lambda function, if you are using the Lambda data source.

```
#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)
```

Run the following query:

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
      id
      name
      performers
    }

    ... on Concert {
      id
      name
      performingBand
    }
  }
}
```

```
    }  
    ... on Conference {  
      speakers  
      workshops  
    }  
  }  
}
```

The query yields the following results:

```
{  
  "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 Screammers"  
        ]  
      },  
      {  
        "id": "Concert-3",  
        "name": "Concert 3",  
        "performingBand": "The Jumpers"  
      },  
      {  
        "speakers": [  
          "The Storytellers"  
        ],  
        "workshops": [  
          "Writing",  
          "Reading"  
        ]  
      }  
    ]  
  }  
}
```

The type resolution logic varies depending on the application. For example, you could have a different identifying logic that checks for the existence of certain fields or even a combination of fields. That is, you could detect the presence of the `performers` field to identify a `Festival` or the combination of the `speakers` and the `workshops` fields to identify a `Conference`. Ultimately, it is up to you to define the logic you want to use.

## Attaching a Data Source

### (Recommended) Automatic Provision

Continuing on from [Designing Your Schema \(p. 12\)](#), you can have AWS AppSync automatically create tables based on your schema definition. This is an optional step, but recommended step if you are just getting started. AWS AppSync also creates all resolvers for you during this process and you can immediately write GraphQL queries, mutations, and subscriptions. You can follow this process in

(Optional) [Provision from Schema \(p. 43\)](#). The rest of this tutorial assumes you are skipping the automatic provisioning process and building from scratch.

## Adding a Data Source

Now that you created a schema in the AWS AppSync console and saved it, you can add a data source. The schema in the previous section assumes that you have a Amazon DynamoDB table called `todos` with a primary key called `id` of type **String**. You can create this manually in the Amazon DynamoDB console or using the following AWS CloudFormation stack:



### To add your data source

1. Choose the **Data Sources** tab in the console, and then choose **New**.
  - Give your data source a friendly name, such as `TodosTable`.
2. Choose **Amazon DynamoDB Table** as the type.
  - Choose the appropriate region.
3. Choose your **Todos** table. Then either create a new role (recommended) or choose an existing role that has IAM permissions for `PutItem` and `scan` for your table. Existing roles need a trust policy, as explained later.
4. If you have existing tables, you could also generate CRUD, List, and Query operations automatically by selecting **Automatically generate GraphQL** as outlined in [\(Optional\) Import from Amazon DynamoDB \(p. 46\)](#) but for this tutorial leave it unselected.

If you aren't following the advanced part of the tutorial where your schema uses pagination and relations (with GraphQL connections), you can go directly to [Configuring Resolvers \(p. 29\)](#).

If you're doing the advanced section with pagination and relations, you need to repeat the above with a table named `Comments` with a primary key of `todoId` and a sort key of `commentId`, where both are of type **String**. Additionally, you must create a global secondary index on the table called `todoId-index` with a partition key `todoId` of type **String**. You can create this manually in the Amazon DynamoDB console or using the following AWS CloudFormation stack:



You need IAM permissions of `code:PutItem` and `code:Query` on the `Comments` table. We recommend you use **create new role** as shown previously.

Now that you've connected a data source, you can connect it to your schema with a resolver. Move onto [Configuring Resolvers \(p. 29\)](#).

## IAM Trust Policy

If you're using an existing IAM role for your data source, you need to grant that role the appropriate permissions to perform operations on your AWS resource, such as `PutItem` on an Amazon DynamoDB table. You also need to modify the trust policy on that role to allow AWS AppSync to use it for resource access as shown in the following example policy:

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

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

You can also add conditions to your trust policy to limit the access to the data source as desired. Currently, `SourceArn` and `SourceAccount` keys can be used in these conditions. For example, the following policy limits access to your data source to the account 123456789012:

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

For more information regarding cross-account access of AWS Lambda resolvers for AWS AppSync, see [Building cross-account AWS Lambda resolvers for AWS AppSync](#).

Alternatively, you can limit access to a data source to a specific API, such as `abcdefghijklmnopq`, using the following policy:

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

You can limit access to all AWS AppSync APIs from a specific region, such as `us-east-1`, using the following policy:

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

## Configuring Resolvers

In this section, you walk through how to create a resolver, add a resolver for mutations, and use advanced configurations.

### Create Your First Resolver

In the AWS AppSync console, go to the **Schema** page. In the **Query** type on the right side, choose **Attach resolver** next to the `getTodos` field. On the **Create Resolver** page, choose the data source you just created, and then choose a default template or paste in your own. For common use cases, the AWS AppSync console has built-in templates that you can use for getting items from data sources (for example, all item queries, individual lookups, etc.). For example, on the simple version of the schema from [Designing Your Schema \(p. 12\)](#) where `getTodos` didn't have pagination, the mapping template is as follows:

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

You always need a response mapping template. The console provides a default with the following passthrough value for lists:

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

In this example, the context object (aliased as `$ctx`) for lists of items has the form `$context.result.items`. If your GraphQL operation returns a single item, it would be `$context.result`. AWS AppSync provides helper functions for common operations, such as the `$util.toJson` function listed previously, to format responses properly. For a full list of functions, see [Resolver Mapping Template Utility Reference \(p. 286\)](#).

**Note:** The default resolver you have just created is a **Unit** resolver, but AppSync also supports creating **Pipeline** resolvers to run operations against multiple data sources in a sequence. See [Pipeline Resolvers \(p. 35\)](#) for more information.

### Adding a Resolver for Mutations

Repeat the preceding process, starting at the **Schema** page and choosing **Attach resolver** for the `addTodo` mutation. Because this is a mutation where you're adding a new item to DynamoDB, use the following request mapping template:

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

AWS AppSync automatically converts arguments defined in the `addTodo` field from your GraphQL schema into DynamoDB operations. The previous example stores records in DynamoDB using a key of `id`, which is passed through from the mutation argument as `$ctx.args.id`. All of the other fields you pass through are automatically mapped to DynamoDB attributes with `$util.dynamodb.toMapValuesJson($ctx.args)`.

For this resolver, use the following response mapping template:

```
$utils.toJson($context.result)
```

AWS AppSync also supports test and debug workflows for editing resolvers. You can use a mock context object to see the transformed value of the template before invoking. Optionally, you can view the full request execution to a data source interactively when you run a query. For more information, see [Test and Debug Resolvers \(p. 33\)](#) and [Monitoring and Logging \(p. 205\)](#).

At this point, if you're not using the advanced resolvers you can begin using your GraphQL API as outlined in [Using Your API \(p. 40\)](#).

## Advanced Resolvers

If you are following the Advanced section and you're building a sample schema in [Designing Your Schema \(p. 12\)](#) to do a paginated scan, use the following request template for the `getTodos` field:

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

For this pagination use case, the response mapping is more than just a passthrough because it must contain both the *cursor* (so that the client knows what page to start at next) and the result set. The mapping template is as follows:

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

The fields in the preceding response mapping template should match the fields defined in your `TodoConnection` type.

For the case of relations where you have a `Comments` table and you're resolving the comments field on the `Todo` type (which returns a type of `[Comment]`), you can use a mapping template that runs a query against the second table. To do this, you must have already created a data source for the `Comments` table as outlined in [Attaching a Data Source \(p. 26\)](#).

**Note:** We're using a query operation against a second table for illustrative purposes only. You could use another operation against DynamoDB instead. Further, you could pull the data from another data source,

such as AWS Lambda or Amazon OpenSearch Service, because the relation is controlled by your GraphQL schema.

On the **Schema** page in the console, choose the `comments` field in **Todo type**, and then choose **Attach**. Choose **Comments table data source** and use the following request mapping template:

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

The `context.source` references the parent object of the current field that's being resolved. In this example, `source.id` refers to the individual `Todo` object, which is then used for the query expression.

You can use the passthrough response mapping template as follows:

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

Finally, on the **Schema** page in the console, attach a resolver to the `addComment` field, and specify the data source for the `Comments` table. The request mapping template in this case is a simple `PutItem` with the specific `todoid` that is commented on as an argument, but you use the `$utils.autoId()` utility to create a unique sort key for the comment as follows:

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

Use a passthrough response template as follows:

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

## Direct Lambda Resolvers

### Direct Lambda Resolvers

With Direct Lambda Resolvers, you can circumvent the use of VTL mapping templates when using AWS Lambda data sources. AppSync can provide a default payload to your Lambda function as well as a default translation from a Lambda function's response to a GraphQL type. You can choose to provide a request template, a response template, or neither and AWS AppSync will handle it accordingly.

To learn more about the default request payload and response translation that AWS AppSync provides, see the [Direct Lambda resolver reference \(p. 378\)](#). For more information on setting up an AWS Lambda data source and setting up an IAM Trust Policy, see [attaching a data source \(p. 26\)](#).

## Add a Lambda data source

Before you can activate Direct Lambda resolvers, you first have to add a Lambda data source.

### To add a Lambda data source

1. Sign in to the AWS Management Console and open the [AWS AppSync Console](#).
2. In the navigation pane, choose **Data Sources**, and then choose **Create data source**.
  - a. For **Data source name**, enter a name for your data source, such as **myFunction**.
  - b. For **Data source type**, choose **AWS Lambda function**.
  - c. For **Region**, choose the appropriate region.
  - d. For **Function ARN**, choose the Lambda function from the dropdown list. You can search for the function name or manually enter the ARN of the function you want to use.
  - e. Choose the **Create** button.
3. Create a new IAM role (recommended) or choose an existing role that has the `lambda:invokeFunction` IAM permission. Existing roles need a trust policy, as explained in [attaching a data source \(p. 26\)](#).

The following is an example IAM policy that has the required permissions to perform operations on the resource:

```
{
  "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:*"
      ]
    }
  ]
}
```

## Activate direct Lambda resolvers

After creating a Lambda data source and setting up the appropriate IAM role to allow AWS AppSync to invoke the function, you can link it to a resolver or pipeline function.

### To link a Lambda data source to a resolver or pipeline function

1. Sign in to the AWS Management Console and open the [AWS AppSync Console](#).
2. In the navigation pane, choose **Schema**.
3. In the **Resolvers** section, choose a field or operation and then select the **Attach** button.
4. In **Create new resolver**, choose the Lambda function from the dropdown list.
5. In order to leverage Direct Lambda resolvers, confirm that request and response mapping templates are disabled in the **Configure mapping templates** section.

By disabling a mapping template, you are signalling to AWS AppSync that you accept the default data translations specified in the [Direct Lambda resolver reference \(p. 378\)](#).

By disabling the request mapping template, your Lambda data source will receive a payload consisting of the entire [Context \(p. 279\)](#) object.



By disabling the response mapping template, the result of your Lambda invocation will be translated depending on the version of the request mapping template or if the request mapping template is also disabled.

## Test and Debug Resolvers

AWS AppSync executes resolvers on a GraphQL field against a data source. As described in [Resolver Mapping Template Overview \(p. 264\)](#), resolvers communicate with data sources by using a templating language. This enables you to customize the behavior, and apply logic and conditions before and after communicating with the data source. For an introductory tutorial-style programming guide for writing resolvers, see the [Resolver Mapping Template Programming Guide \(p. 268\)](#).

To help developers write, test, and debug these resolvers, the AWS AppSync console also provides tools to create a GraphQL request and response with mock data, down to the individual field resolver. Additionally, you can perform queries, mutations, and subscriptions in the AWS AppSync console, and see a detailed log stream from Amazon CloudWatch of the entire request. This includes results from a data source.

### Testing with Mock Data

When a GraphQL resolver is invoked, it contains a `context` object that has relevant information about the request for you to program against. This includes arguments from a client, identity information, and data from the parent GraphQL field. It also has results from the data source, which can be used in the response template. For more information about this structure and the available helper utilities to use when programming, see the [Resolver Mapping Template Context Reference \(p. 279\)](#).

When writing or editing a resolver function, you can pass a *mock* or *test* context object into the console editor. This enables you to see how both the request and the response templates evaluate without actually running against a data source. For example, you can pass a test `firstname: Shaggy` argument and see how it evaluates when using `$ctx.args.firstname` in your template code. You could also test the evaluation of any utility helpers such as `$util.autoId()` or `util.time.nowISO8601()`.

### Test a Resolver

In the AWS AppSync console, go to the **Schema** page, and choose an existing resolver on the right to edit it. Or, choose **Attach** to add a new resolver. At the top of the page, choose **Select test context**, choose **Create new context**, and then enter a name. Next, either select from an existing sample context object or populate the JSON manually, and then choose **Save**. To evaluate your resolver using this mocked context object, choose **Run Test**.

For example, suppose you have an app storing a GraphQL type of `Dog` that uses automatic ID generation for objects and stores them in Amazon DynamoDB. You also want to write some values from the arguments of a GraphQL mutation, and allow only specific users to see a response. The following shows what the schema might look like:

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

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

When you add a resolver for the `addDog` mutation, you can populate a context object like the one following. The following has arguments from the client of `name` and `age`, and a `username` populated in the `identity` object:

```
{
  "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"
  }
}
```

You can test this using the following request and response mapping templates:

#### Request Template

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

#### Response Template

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

The evaluated template has the data from your test context object and the generated value from `$util.autoId()`. Additionally, if you were to change the `username` to a value other than `Nadia`, the results won't be returned because the authorization check would fail. For more information about fine grained access control, see [Authorization Use Cases \(p. 233\)](#).

## Debugging a Live Query

There's no substitute for an end-to-end test and logging to debug a production application. AWS AppSync lets you log errors and full request details using Amazon CloudWatch. Additionally, you can use the AWS AppSync console to test GraphQL queries, mutations, and subscriptions and live stream log data for each request back into the query editor to debug in real time. For subscriptions, the logs display connection-time information.

To perform this, you need to have Amazon CloudWatch logs enabled in advance, as described in [Monitoring and Logging \(p. 205\)](#). Next, in the AWS AppSync console, choose the **Queries** tab and then enter a valid GraphQL query. In the lower-right section, select the **Logs** check box to open the logs view. At the top of the page, choose the play arrow icon to run your GraphQL query. In a few moments, your

full request and response logs for the operation are streamed to this section and you can view them in the console.

## Pipeline Resolvers

AWS AppSync executes resolvers on a GraphQL field. In some cases, applications require executing multiple operations to resolve a single GraphQL field. With pipeline resolvers, developers can now compose operations (called Functions) and execute them in sequence. Pipeline resolvers are useful for applications that, for instance, require performing an authorization check before fetching data for a field.

### Anatomy of a pipeline resolver

A pipeline resolver is composed of a **Before** mapping template, an **After** mapping template, and a list of Functions. Each Function has a **request** and **response** mapping template that it executes against a Data Source. As a pipeline resolver delegates execution to a list of functions, it is therefore not linked to any data source. Unit resolvers and functions are primitives that execute operation against data sources. See the [Resolver Mapping Template Overview \(p. 264\)](#) for more information.

#### Before Mapping template

The request mapping template of a pipeline resolver or also called the **Before** step, allows to perform some preparation logic before executing the defined functions.

#### Functions list

The list of functions a pipeline resolver will run in sequence. The pipeline resolver request mapping template evaluated result is made available to the first function as `$ctx.prev.result`. Each function output is available to the next function as `$ctx.prev.result`.

#### After mapping template

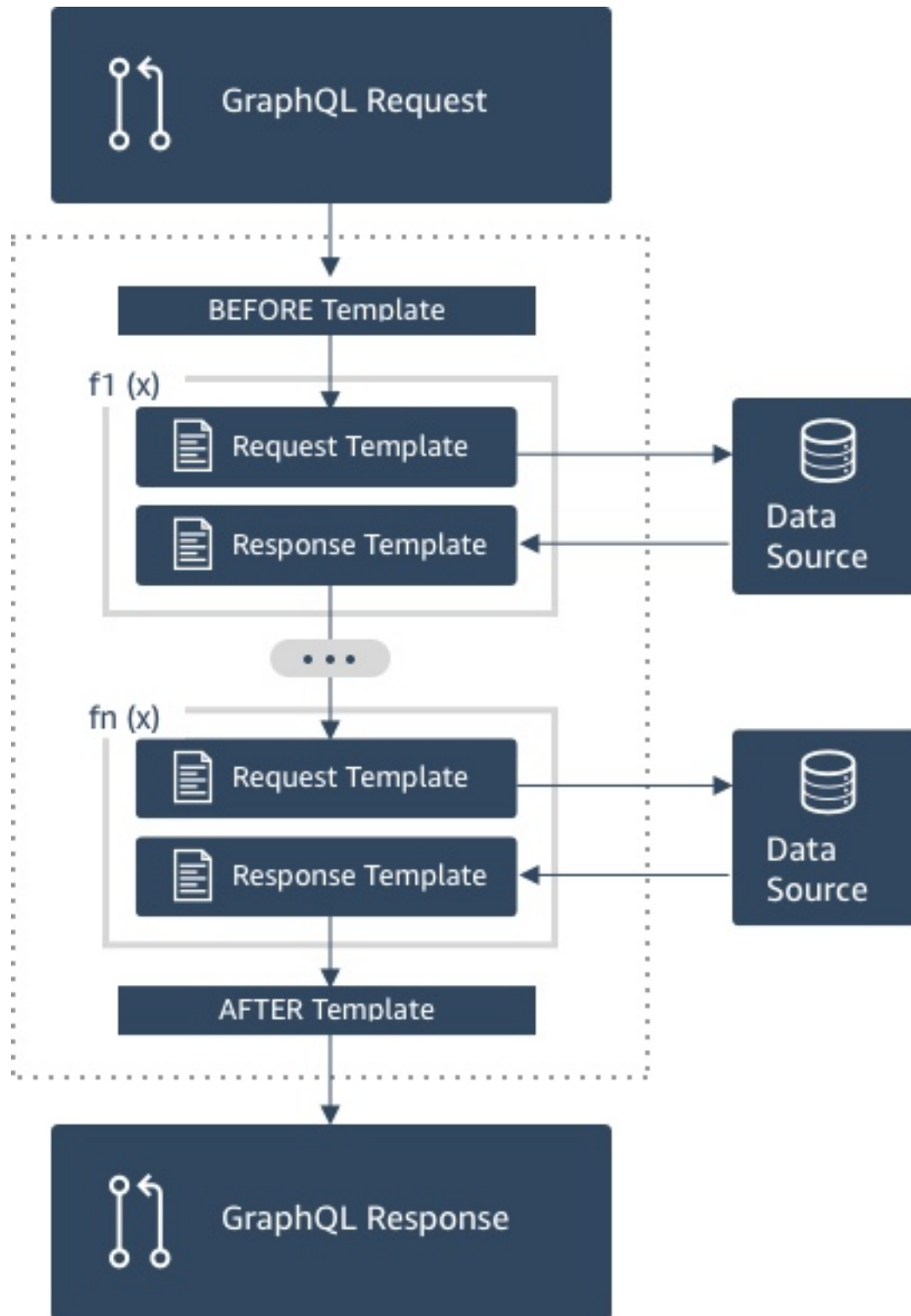
The response mapping template of a pipeline resolver or also called the **After** step, allows to perform some final mapping logic from the output of the last function to the expected GraphQL field type. The output of the last function in the functions list is available in the pipeline resolver mapping template as `$ctx.prev.result` or `$ctx.result`.

#### Execution Flow

Given a pipeline resolver comprised of 2 functions, the list below represents the execution flow when the resolver is invoked:

1. Pipeline resolver BEFORE mapping template
2. Function 1: Function request mapping template
3. Function 1: Data source invocation
4. Function 1: Function response mapping template
5. Function 2: Function request mapping template
6. Function 2: Data source invocation
7. Function 2: Function response mapping template
8. Pipeline resolver AFTER mapping template

Pipeline resolver execution flow is unidirectional and defined statically on the resolver.



## Useful Apache Velocity Template Language (VTL) Utilities

As the complexity of an application increases, VTL utilities and directives are here to facilitate development productivity. The following utilities can help you when you're working with pipeline resolvers.

### `$ctx.stash`

The stash is a Map that is made available inside each resolver and function mapping template. The same stash instance lives through a single resolver execution. What this means is you can use the stash to pass arbitrary data across request and response mapping templates, and across functions in a pipeline resolver. The stash exposes the same methods as the [Java Map](#) data structure.

### `$ctx.prev.result`

The `$ctx.prev.result` represents the result of the previous operation that was executed in the pipeline. If the previous operation was the pipeline resolver request mapping template, then `$ctx.prev.result` represents the output of the evaluation of the template and is made available to the first function in the chain. If the previous operation was the first function, then `$ctx.prev.result` represents the output of the first function and is made available to the second function in the pipeline. If the previous operation was the last function, then `$ctx.prev.result` represents the output of the last function and is made available to the pipeline resolver response mapping template.

### `#return(data: Object)`

The `#return(data: Object)` directive comes handy if you need to return prematurely from any mapping template. `#return(data: Object)` is analogous to the `return` keyword in programming languages because it returns from the closest scoped block of logic. What this means is that using `#return` inside a resolver mapping template returns from the resolver. Using `#return(data: Object)` in a resolver mapping template sets data on the GraphQL field. Additionally, using `#return(data: Object)` from a function mapping template returns from the function and continues the execution to either the next function in the pipeline or the resolver response mapping template.

### `#return`

Same as `#return(data: Object)` but `null` will be returned instead.

### `$util.error`

The `$util.error` utility is useful to throw a field error. Using `$util.error` inside a function mapping template throws a field error immediately, which prevents subsequent functions from being executed. For more details and other `$util.error` signatures, visit the [Resolver Mapping Template Utility Reference](#) (p. 286).

### `$util.appendError`

The `$util.appendError` is similar to the `$util.error()`, with the major distinction that it doesn't interrupt the evaluation of the mapping template. Instead, it signals there was an error with the field, but allows the template to be evaluated and consequently return data. Using `$util.appendError` inside a function will not disrupt the execution flow of the pipeline. For more details and other `$util.error` signatures, visit the [Resolver Mapping Template Utility Reference](#) (p. 286).

## Create A Pipeline Resolver

In the AWS AppSync console, go to the **Schema** page.

Save the following schema:

```
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
}
```

We are going to wire a pipeline resolver to the **signUp** field on the **Mutation** type. In the **Mutation** type on the right side, choose **Attach resolver** next to the `signUp` field. On the **Create Resolver** page, click on the **Switch to Pipeline** button. The page should now show 3 sections, a **Before Mapping Template** text area, a **Functions** section, and a **After Mapping template** text area.

Our pipeline resolver signs up a user by first validating the email address input and then saving the user in the system. We are going to encapsulate the email validation inside a **validateEmail** function, and the saving of the user inside a **saveUser** function. The **validateEmail** function executes first, and if the email is valid, then the **saveUser** function executes.

The execution flow will be as follow:

1. Mutation.signUp resolver request mapping template
2. validateEmail function
3. saveUser function
4. Mutation.signUp resolver response mapping template

Because we will probably reuse the **validateEmail** function in other resolvers on our API, we want to avoid accessing `$ctx.args` since these will change from one GraphQL field to another. Instead, we can use the `$ctx.stash` to store the email attribute from the `signUp(input: Signup)` input field argument.

BEFORE mapping template:

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

The console provides a default passthrough AFTER mapping template that will we use:

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

## Create A Function

From the pipeline resolver page, on the **Functions** section, click on **Create Function**. It is also possible to create functions without going through the resolver page, to do this, in the AWS AppSync console, go to

the **Functions** page. Choose the **Create Function** button. Let's create a function that checks if an email is valid and comes from a specific domain. If the email is not valid, the function raises an error. Otherwise, it forwards whatever input it was given.

Select **NONE** data source on the function page, and fill in the **validateEmail** request mapping template:

```
#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) }
}
```

and response mapping template:

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

We just created our **validateEmail** function. Repeat these steps to create the **saveUser** function with the following request and response mapping templates. For the sake of simplicity we use a **NONE** data source and pretend the user has been saved in the system after the function executes.

Request mapping template:

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

and response mapping template:

```
## 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)
```

We just created out **saveUser** function.

## Adding a Function to a Pipeline Resolver

Our functions should have been automatically added to the pipeline resolver we just created. If you happened to have created the functions through the console **Functions** page, you can click on **Add Function** on the resolver page to attach them. Add both **validateEmail** and **saveUser** functions to the resolver. The **validateEmail** function should be placed before the **saveUser** function. As you add more functions you can use the up and down arrows to reorganize the order of execution of your functions.

## Executing a Query

In the AWS AppSync console, go to the **Queries** page. Enter the following query:

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

```
}
```

should return something like:

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

We have successfully signed up our user and validated the input email using a pipeline resolver. To follow a more complete tutorial focusing on pipeline resolvers, you can go to [Tutorial: Pipeline Resolvers \(p. 146\)](#)

## Using Your API

Now that you have a GraphQL API with a schema uploaded, data sources configured, and resolvers connected to your types, you can test your API. The following two examples assume you've used the basic schema from [Designing Your Schema \(p. 12\)](#).

In the AWS AppSync, choose the **Queries** tab and then enter the following text in the editor:

```
mutation add {
  addTodo(id:"123" name:"My TODO" description:"Testing AWS AppSync" priority:2){
    id
    name
    description
    priority
  }
}
```

Choose the button at the top to run your mutation. After the run is complete, the result from your selection set (that is, `id`, `name`, `description`, and `priority`) appear on the right. The data is also in the Amazon DynamoDB table for your data source, which you can verify using the console.

Now, run a query as follows:

```
query list {
  getTodos {
    id
    name
  }
}
```

This returns your data, but it only returns the two fields (`id` and `name`) from your selection set. **Note:** If you are getting an error such as **Validation error of type FieldUndefined**, you might have followed the instructions for setting up relations and pagination. If so, use the queries in the next section.

## Relations and Pagination

If you have set up the advanced schema with relations and pagination, your `getTodos` query will look a bit different. First, add a comment to your `todo` as follows:

```
mutation addComment {
```



```
    addComment(todoId:"123" content:"GraphQL is fun"){
      todoId
      commentId
      content
    }
  }
}
```

In this case, the `commentId` is automatically created. Add a few more todos and comments, set the `limit` to 2 records, and then run `getTodos` as follows:

```
query list {
  getTodos(limit:2){
    todos {
      id
      name
      status
      comments{
        commentId
        content
      }
    }
    nextToken
  }
}
```

Only two todos come back (with the comments) because you specified `limit:2` in the arguments. A long string should be returned for the `nextToken` field. If you copy this string, you can pass it as a second argument in `getTodos` by replacing `XXXXXXXXXXXX` in `getTodos(limit:2 nextToken: "XXXXXXXXXXXX")`. For example:

```
query list {
  getTodos(limit:2
    nextToken: "eyJ2ZXJzaW9uejE1a2RPanZPQzFCMlFTdGNxdUFBQUJxekNDQWFjR0NTcUdTSWlzMjRFFSEJxQ0NBWmd3Z2dHVUFnRUU"
  ) {
    todos {
      id
      name
      status
      comments{
        commentId
        content
      }
    }
    nextToken
  }
}
```

In a client application you would pass this token through as an argument based on your pagination flow and logic.

## Next Steps

Now that you've created a schema, you can walk through how to use it with different data sources, learn more about the advanced resolver capabilities, or build a client application. For more information, see the following sections:

- [Data Sources tutorials \(p. 51\)](#)
- [Resolvers and Mapping Templates \(p. 264\)](#)
- [Building Client Applications \(p. 50\)](#)

## (Optional) Guided Schema Wizard

AWS AppSync can guide you through the process of creating your GraphQL API, including deploying data sources and connecting with resolvers. This guided creation or wizard gives you a form builder interface where you can add or remove attributes while creating a data model for your application to store in the cloud with GraphQL.

### Create API

From the AWS AppSync home page, choose **Create API** and then choose **Create with wizard**. You are prompted to define your model and its related fields and functionality. A *model* is an object type that is added to your API and is backed by Amazon DynamoDB that comes preconfigured with GraphQL queries, mutations, and subscriptions. Models are how you define the data types of your application and how they are stored in the cloud. A model contains one or more *fields* that hold the specific attributes of your data type. For example, a *blog* model might have the following fields: id, title, and isPublished. These fields could have primitive types of id, string, or Boolean in addition to the custom scalars that AWS AppSync supports.

### Populate Model

After you enter the model form, you can change the name of your model, which in turn defines the names of your GraphQL operations (that is, queries, mutations, and subscriptions). This also defines the name of your Amazon DynamoDB tables, but you can override that in the **Configure model table** section at the bottom of the screen.

Next, add one or more fields using logical names and types that suit your model. To view all available primitive types in AWS AppSync, choose the drop down under **Types**. Additionally, you can select one or both of the check boxes next to each field to mark it as a **List** or **Required**.

Types that are marked as **List** can hold multiple values. For example, if you had a blog Post model and a field called **Categories** you could mark this as a **List** to return multiple categories when running a query. **Required** fields must be entered as input when executing a GraphQL mutation.

Finally, you can use the **Configure model table** section at the bottom to optionally add an index. This is useful when you know that certain fields in your model will be queried frequently. For example, if the isPublished field of your blog model is going to be frequently queried to give all blogs that are published, you could create an index on that field. This automatically creates proper Amazon DynamoDB indexes and GraphQL queries for you.

After you are satisfied with your model and optional table configuration, choose **Next** and then the resource is created.

### Execute GraphQL

After the process is complete, pre-populated GraphQL mutations and queries open in the editor for you to run on the **Queries** page of the AWS AppSync console. Choose the play button at the top and choose the option that starts with **Create <model name>** (this option name changes depending on your model name). A GraphQL mutation runs and places data in your Amazon DynamoDB table. You can then choose the play button again and select the option starting with **List** to view all of the records by running a GraphQL query.

### Integrate with the App

After you complete a tour of the console, choose the root of the AWS AppSync navigation bar on the left to follow [instructions for integrating a GraphQL API with your mobile or web app](#) (p. 50).

## (Optional) Provision from Schema

AWS AppSync can automatically provision Amazon DynamoDB tables from a schema definition, create data sources, and connect the resolvers on your behalf. This can be useful if you want to let AWS AppSync define the appropriate table layout and indexing strategy based on your schema definition and data access patterns.

You can also start with no schema and build it up from a type, letting AWS AppSync create the schema definition and different filtering options. You can use the two flows outlined in this topic independently or together throughout the lifecycle of your API.

### No Schema

After you create an AWS AppSync API, go to the **Schema** page and choose **Create Resources**. The editor at the top contains a pre-defined template for a type. You can change the name, field types, and add or remove entries. For example, use the following sample type:

```
type Blog {
  id: ID!
  title: String!
  content: String
  rating: Int
  approved: Boolean
}
```

Lower on the page, you can change the table name, change the layout, or add indexes. Indexes are recommended as a best practice for performance if you plan to query often using a specific field. When creating an index, AWS AppSync builds the GraphQL schema and connects resolvers for you. For example, choose **Add Index** and name the index **rating**. In **Primary Key**, choose **rating**. In **Sort Key**, leave the default **none**. You should see a query like the following in the GraphQL that's generated:

```
queryBlogsByRating(rating:Int!, first:Int, nextToken: String): BlogConnection
```

After you've finished editing your type and creating any needed indices, choose **Create**, and then wait for the process to complete creating resources and connecting data sources to resolvers. You can go to the **Queries** page immediately and start running mutations, queries, and subscriptions. For example, create a blog post with the following mutation:

```
mutation add {
  createBlog(input:{
    title:"My first post"
    content:"You can use AppSync with GraphQL"
    rating:5
    approved:true
  }){
    id
  }
}
```

Notice that the `id` field is automatically generated. AWS AppSync does this if your type has a **Primary Key** field on a table of `id: ID!`. Otherwise, AWS AppSync require the **Primary Key** field to be passed as part of the argument. After creating a few blog posts with different content, run the following query:

```
query list {
  listBlogs(filter:{ content:{
    contains:"AppSync"
  }})
```

```
    }}{  
      items{  
        id  
        title  
        content  
        rating  
      }  
    }  
  }  
}
```

It should only return the blogs with the string “**AppSync**” in the content field. You can explore the different filters on fields via GraphQL introspection on the **Queries** page of the console. AWS AppSync generates several queries and filters for common use cases on different scalar types to help you get up and running fast. It's important to note that as your application usage grows larger and you have more complex data requirements around scale or larger tables, all of these filters might no longer be appropriate and you should leverage different DynamoDB best practices or combine data sources in your GraphQL API, such as Amazon OpenSearch Service for complex searches.

AWS AppSync creates filters on GraphQL scalar types (that is, ID, String, Int, Float, Boolean) for list operations. This is done by creating a base input type that contains the different fields for your defined type. Each one of these fields has an input filter type as appropriate for things like string searches, Boolean comparisons, etc. For example, only **String** supports **BEGINS\_WITH** while **Boolean** supports **EQ**, **NE**, etc. You can view the supported operations in the generated GraphQL input types on the **Schema** page and a list of DynamoDB operations at [DynamoDB operations](#).

Continuing with the blog example type, the following generates a base input type in your schema:

```
input BlogInput {  
  id: TableIDFilterInput  
  title: TableStringFilterInput  
  content: TableStringFilterInput  
  rating: TableIntFilterInput  
  approved: TableBooleanFilterInput  
}
```

The AWS AppSync console automatically connects this to a query operation. If you want to do this yourself, it might look like the following:

```
listBlog(filter: BlogInput): [Blog]
```

When you attach a resolver to the `listBlog` query, the request mapping template uses the input on the filter key along with `$util.transform.toDynamoDBFilterExpression` as follows:

```
{  
  "version": "2017-02-28",  
  "operation": "Scan",  
  "filter": #if($context.args.filter)  
    $util.transform.toDynamoDBFilterExpression($ctx.args.filter) #else null #end  
}
```

## Existing Schema

These instructions start with the schema outlined in [Designing Your Schema \(p. 12\)](#). From the AWS AppSync console, go to the **Schema** page, enter the following schema into the editor, and then choose **Save Schema**:

```
schema {
```

```
    query: Query
    mutation: Mutation
  }

  type Query {
    allTodo: [Todo]
  }

  type Mutation {
    addTodo(id: ID!, name: String, description: String, priority: Int, status: TodoStatus):
      Todo
  }

  type Todo {
    id: ID!
    name: String
    description: String
    priority: Int
    status: TodoStatus
  }

  enum TodoStatus {
    done
    pending
  }
```

After you save a schema, choose **Create resources** at the top of the page. Choose **Use existing type**, and then choose your `Todo` type. In the form that appears, you can configure the table details. You can change your DynamoDB primary or sort keys here, and add additional indexes. At the bottom of the page is a corresponding section for the GraphQL queries and mutations that are available to you, based on different key selections. AWS AppSync will provision DynamoDB tables that best match your data access pattern for efficient use of your database throughput. You can also select indices for different query options, which set up a DynamoDB local secondary index or global secondary indexes, as appropriate.

For the example schema, you can simply have `id` selected as the primary key and choose **Create**. If your type doesn't have `id` set as the primary key, all of the fields for that type are required for create operations. Otherwise, AWS AppSync automatically generates unique IDs for you in the resolvers. After a moment, your DynamoDB tables are created, data sources are created, and resolvers are connected. You can run mutations and queries as described in [Using Your API \(p. 40\)](#).

**Note:** There's a GraphQL input type for the arguments of the created schema. For example, if you provision from a schema with a GraphQL type `Books { ... }`, there might be an input type like the following:

```
input CreateBooksInput {
  ISBN: String!
  Author: String
  Title: String
  Price: Int
}
```

To use this in a GraphQL query or mutation, you would use the following:

```
mutation add {
  createBooks(input:{
    ISBN:"2349238"
    Author:"Nadia Bailey"
    Title:"Running in the park"
    Price:10
  }){

```

```
    ISBN
    Author
  }
}
```

Also, as explained earlier in **No Schema** section of this document, default filters will be created for list operations. For example, if you want to return all items where the price is greater than 5, you could run the following:

```
query list {
  listBooks(filter:{ Price:{
    gt:5
  }}){
    items{
      ISBN
      Author
      Title
      Price
    }
  }
}
```

## (Optional) Import from Amazon DynamoDB

AWS AppSync can automatically create a GraphQL schema and connect resolvers to existing Amazon DynamoDB tables. This can be useful if you have DynamoDB tables that you want to expose data through a GraphQL endpoint, or if you're more comfortable starting first with your database design instead of a GraphQL schema.

### Import a DynamoDB Table

From the AWS AppSync console, go to the **Data Sources** page, choose **New**, enter a friendly name for your data source, and then choose Amazon DynamoDB as the data source type. Choose the appropriate table, and then choose **Automatically generate GraphQL**.

The following two code editors with the GraphQL schema appear:

- **Top editor** - You can use this editor to give your type a custom name (such as type `MYNAME { ... }`), which will contain the data from your DynamoDB table when you run queries or mutations. You can also add fields to the type, such as DynamoDB non-key attributes (which cannot be detected on import).
- **Bottom editor** - Use this read-only editor to review generated GraphQL schema snippets. It shows what types, queries, and mutations will be merged into your schema. If you edit the type in the top editor, the bottom editor changes as appropriate.

Choose **Create**. Your schema is merged and resolvers are created. After this is complete, you can run mutations and queries as described in [Using Your API \(p. 40\)](#).

**Note:** A GraphQL input type is created for the arguments of the created schema. For example, if you import a table called Books, there might be an input type like the following:

```
input CreateBooksInput {
  ISBN: String!
  Author: String
  Title: String
}
```

```
    Price: String
  }
```

To use this in a GraphQL query or mutation, do the following:

```
mutation add {
  createBooks(input:{
    ISBN:2349238
    Author:"Nadia Bailey"
    Title:"Running in the park"
    Price:"10"
  }){
    ISBN
    Author
  }
}
```

## Example Schema from Import

Suppose that you have a DynamoDB table with the following format:

```
{
  Table: {
    AttributeDefinitions: [
      {
        AttributeName: 'authorId',
        AttributeType: 'S'
      },
      {
        AttributeName: 'bookId',
        AttributeType: 'S'
      },
      {
        AttributeName: 'title',
        AttributeType: 'S'
      }
    ],
    TableName: 'BookTable',
    KeySchema: [
      {
        AttributeName: 'authorId',
        KeyType: 'HASH'
      },
      {
        AttributeName: 'title',
        KeyType: 'RANGE'
      }
    ],
    TableArn: 'arn:aws:dynamodb:us-west-2:012345678910:table/BookTable',
    LocalSecondaryIndexes: [
      {
        IndexName: 'authorId-bookId-index',
        KeySchema: [
          {
            AttributeName: 'authorId',
            KeyType: 'HASH'
          },
          {
            AttributeName: 'bookId',
            KeyType: 'RANGE'
          }
        ]
      }
    ]
  }
}
```

```
        Projection: {
          ProjectionType: 'ALL'
        },
        IndexSizeBytes: 0,
        ItemCount: 0,
        IndexArn: 'arn:aws:dynamodb:us-west-2:012345678910:table/BookTable/index/authorId-bookId-index'
      },
    ],
    GlobalSecondaryIndexes: [
      {
        IndexName: 'title-authorId-index',
        KeySchema: [
          {
            AttributeName: 'title',
            KeyType: 'HASH'
          },
          {
            AttributeName: 'authorId',
            KeyType: 'RANGE'
          }
        ],
        Projection: {
          ProjectionType: 'ALL'
        },
        IndexArn: 'arn:aws:dynamodb:us-west-2:012345678910:table/BookTable/index/title-authorId-index'
      }
    ]
  }
}
```

The type editor at the top shows the following:

```
type Book {
  # Key attributes. Changing these may result in unexpected behavior.
  authorId: String!
  title: String!

  # Index attributes. Changing these may result in unexpected behavior.
  bookId: String

  # Add additional non-key attributes below.
  isPublished: Boolean
}
```

This top editor is writable, and you need to add the non-key attributes shown in the bottom editor (for example, like `isPublished`) manually because they can't be inferred from DynamoDB automatically. For example, if you had another attribute on an item in your DynamoDB table called `rating`, you need to add it under `isPublished` to have it populated in the GraphQL schema. In this example, the bottom editor would have the following proposed schema merges:

```
type Query {
  getBook(authorId: ID!, title: String!): Book
  listBooks(first: Int, after: String): BookConnection
  getBookByAuthorIdBookIdIndex(authorId: ID!, bookId: ID!): Book
  queryBooksByAuthorIdBookIdIndex(authorId: ID!, first: Int, after: String):
    BookConnection
  getBookByTitleAuthorIdIndex(title: String!, authorId: ID!): Book
  queryBooksByTitleAuthorIdIndex(title: String!, first: Int, after: String):
    BookConnection
}
type Mutation {
```



```
    createBook(input: CreateBookInput!): Book
    updateBook(input: UpdateBookInput!): Book
    deleteBook(input: DeleteBookInput!): Book
  }
  type Subscription {
    onCreateBook(authorId: ID, title: String, bookId: ID, isPublished: Boolean): Book
    @aws_subscribe(mutations: ["createBook"])
    onUpdateBook(authorId: ID, title: String, bookId: ID, isPublished: Boolean): Book
    @aws_subscribe(mutations: ["updateBook"])
    onDeleteBook(authorId: ID, title: String, bookId: ID, isPublished: Boolean): Book
    @aws_subscribe(mutations: ["deleteBook"])
  }
  input CreateBookInput {
    authorId: ID!
    title: String!
    bookId: ID!
    isPublished: Boolean
  }
  input UpdateBookInput {
    authorId: ID!
    title: String!
    bookId: ID
    isPublished: Boolean
  }
  input DeleteBookInput {
    authorId: ID!
    title: String!
  }
  type BookConnection {
    items: [Book]
    nextToken: String
  }
}
```

# Building a Client App

You can use any HTTP or GraphQL client to connect to a GraphQL API in AWS AppSync.

For front-end web and mobile development, we recommend using the [AWS Amplify](#) clients, which are optimized to connect to the AWS AppSync backend. There are some options depending on your application's use case:

- For Amazon DynamoDB data sources, use the `DataStore` category in the AWS Amplify client. It provides the best developer experience and built-in conflict detection and resolution.
- For non-DynamoDB data sources in scenarios where you have no offline requirements, use the API (GraphQL) category in the AWS Amplify client.
- For use cases where you are utilizing the Apollo V3 JavaScript client, use the Apollo Links provided in the [AWS AppSync SDK repository](#) to easily enable authorization and subscriptions.

See the AWS Amplify documentation for more information on the Amplify clients:

- [JavaScript documentation](#)
- [iOS documentation](#)
- [Android documentation](#)

To connect to your AWS AppSync GraphQL endpoints from your backend applications (such as AWS Lambda function or Amazon EC2 instances), we recommend using an HTTP client and signing your requests using IAM by following the [Authorization and authentication](#) guide.

For more information on working with GraphQL in the backend, see [GraphQL from NodeJS](#).

# Resolver tutorials

Data sources and resolvers are how AWS AppSync translates GraphQL requests and fetches information from your AWS resources. AWS AppSync has support for automatic provisioning and connections with certain data source types. AWS AppSync supports AWS Lambda, Amazon DynamoDB, relational databases (Amazon Aurora Serverless), Amazon OpenSearch Service, and HTTP endpoints as data sources. You can use a GraphQL API with your existing AWS resources or build data sources and resolvers. This section takes you through this process in a series of tutorials for better understanding how the details work and tuning options.

AWS AppSync uses *mapping templates* written in Apache Velocity Template Language (VTL) for resolvers. For more information about using mapping templates, see the [Resolver mapping template reference \(p. 264\)](#). More information about working with VTL is available in the [Resolver mapping template programming guide \(p. 268\)](#).

AWS AppSync supports the automatic provisioning of DynamoDB tables from a GraphQL schema as described in [\(Optional\) Provision from Schema \(p. 43\)](#) and [Launch a Sample Schema \(p. 3\)](#). You can also import from an existing DynamoDB table which will create schema and connect resolvers. This is outlined in [\(Optional\) Import from Amazon DynamoDB \(p. 46\)](#).

## Topics

- [Tutorial: DynamoDB Resolvers \(p. 51\)](#)
- [Tutorial: Lambda resolvers \(p. 86\)](#)
- [Tutorial: Amazon OpenSearch Service Resolvers \(p. 100\)](#)
- [Tutorial: Local Resolvers \(p. 106\)](#)
- [Tutorial: Combining GraphQL Resolvers \(p. 108\)](#)
- [Tutorial: DynamoDB Batch Resolvers \(p. 112\)](#)
- [Tutorial: DynamoDB Transaction Resolvers \(p. 125\)](#)
- [Tutorial: HTTP Resolvers \(p. 133\)](#)
- [Tutorial: Aurora Serverless \(p. 138\)](#)
- [Tutorial: Pipeline Resolvers \(p. 146\)](#)
- [Tutorial: Delta Sync \(p. 156\)](#)

## Tutorial: DynamoDB Resolvers

This tutorial shows how you can bring your own Amazon DynamoDB tables to AWS AppSync and connect them to a GraphQL API.

You can let AWS AppSync provision DynamoDB resources on your behalf. Or, if you prefer, you can connect your existing tables to a GraphQL schema by creating a data source and a resolver. In either case, you'll be able to read and write to your DynamoDB database through GraphQL statements and subscribe to real-time data.

There are specific configuration steps that need to be completed in order for GraphQL statements to be translated to DynamoDB operations, and for responses to be translated back into GraphQL. This tutorial outlines the configuration process through several real-world scenarios and data access patterns.

## Setting Up Your DynamoDB Tables

To begin this tutorial, first you need to provision AWS resources using the following AWS CloudFormation template:

```
aws cloudformation create-stack \  
  --stack-name AWSAppSyncTutorialForAmazonDynamoDB \  
  --template-url https://s3.us-west-2.amazonaws.com/awssappsync/resources/dynamodb/  
AmazonDynamoDBCFTemplate.yaml \  
  --capabilities CAPABILITY_NAMED_IAM
```

You can launch the following AWS CloudFormation stack in the US West 2 (Oregon) region in your AWS account.



This creates the following:

- A DynamoDB table called `AppSyncTutorial-Post` that will hold `Post` data.
- An IAM role and associated IAM managed policy to allow AWS AppSync to interact with the `Post` table.

To see more details about the stack and the created resources, run the following CLI command:

```
aws cloudformation describe-stacks \  
  --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

To delete the resources later, you can run the following:

```
aws cloudformation delete-stack \  
  --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

## Creating Your GraphQL API

To create the GraphQL API in AWS AppSync:

- Open the AWS AppSync console and choose **Create API**.
- Set the name of the API to `AWSAppSyncTutorial`.
- Choose **Custom schema**.
- Choose **Create**.

The AWS AppSync console creates a new GraphQL API for you using the API key authentication mode. You can use the console to set up the rest of the GraphQL API and run queries against it for the rest of this tutorial.

## Defining a Basic Post API

Now that you set up an AWS AppSync GraphQL API, you can set up a basic schema that allows the basic creation, retrieval, and deletion of post data.

In the AWS AppSync console, choose the **Schema** tab. In the **Schema** pane, replace the contents with the following code, and then choose the **Save**:

```
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!
}
```

This schema defines a `Post` type and operations to add and get `Post` objects.

## Configuring the Data Source for the DynamoDB Tables

Next, link the queries and mutations defined in the schema to the `AppSyncTutorial-PostDynamoDB` table.

First, AWS AppSync needs to be aware of your tables. You do this by setting up a data source in AWS AppSync:

- Choose the **Data source** tab.
- Choose **New** to create a new data source.
- For the data source name, enter `PostDynamoDBTable`.
- For data source type, choose **Amazon DynamoDB table**.
- For region, choose **US-WEST-2**.
- From the list of tables, choose the **AppSyncTutorial-PostDynamoDB** table.
- In the **Create or use an existing role** section, choose **Existing role**.
- Choose **Create**.

## Setting Up the `addPost` resolver (DynamoDB `PutItem`)

After AWS AppSync is aware of the DynamoDB table, you can link it to individual queries and mutations by defining **Resolvers**. The first resolver you create is the `addPost` resolver, which enables you to create a post in the `AppSyncTutorial-PostDynamoDB` table.

A resolver has the following components:

- The location in the GraphQL schema to attach the resolver. In this case, you are setting up a resolver on the `addPost` field on the `Mutation` type. This resolver will be invoked when the caller calls `mutation { addPost(...){...} }`.
- The data source to use for this resolver. In this case, you want to use the `PostDynamoDBTable` data source you defined earlier, so you can add entries into the `AppSyncTutorial-Post` DynamoDB table.
- The request mapping template. The purpose of the request mapping template is to take the incoming request from the caller and translate it into instructions for AWS AppSync to perform against DynamoDB.
- The response mapping template. The job of the response mapping template is to take the response from DynamoDB and translate it back into something that GraphQL expects. This is useful if the shape of the data in DynamoDB is different to the `Post` type in GraphQL, but in this case they have the same shape, so you just pass the data through.

To set up the resolver:

- Choose the **Schema** tab.
- In the **Data types** pane on the right, find the **addPost** field on the **Mutation** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
  "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 }
  }
}
```

**Note:** A *type* is specified on all the keys and attribute values. For example, you set the `author` field to `{ "S" : "${context.arguments.author}" }`. The `S` part indicates to AWS AppSync and DynamoDB that the value will be a string value. The actual value gets populated from the `author` argument. Similarly, the `version` field is a number field because it uses `N` for the type. Finally, you're also initializing the `ups`, `downs` and `version` field.

For this tutorial you've specified that the GraphQL `ID!` type, which indexes the new item that is inserted to DynamoDB, comes as part of the client arguments. AWS AppSync comes with a utility for automatic ID generation called `$utils.autoId()` which you could have also used in the form of `"id" : { "S" : "${utils.autoId()}" }`. Then you could simply leave the `id: ID!` out of the schema definition of `addPost()` and it would be inserted automatically. You won't use this technique for this tutorial, but you should consider it as a good practice when writing to DynamoDB tables.

For more information about mapping templates, see the [Resolver Mapping Template Overview \(p. 264\)](#) reference documentation. For more information about `GetItem` request mapping, see the [GetItem \(p. 321\)](#) reference documentation. For more information about types, see the [Type System \(Request Mapping\) \(p. 352\)](#) reference documentation.

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

**Note:** Because the shape of the data in the AppSyncTutorial-Post table exactly matches the shape of the Post type in GraphQL, the response mapping template just passes the results straight through. Also note that all of the examples in this tutorial use the same response mapping template, so you only create one file.

- Choose **Save**.

## Call the API to Add a Post

Now that the resolver is set up, AWS AppSync can translate an incoming addPost mutation to a DynamoDB PutItem operation. You can now run a mutation to put something in the table.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following mutation:

```
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  
  }  
}
```

- Choose **Execute query** (the orange play button).
- The results of the newly created post should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{  
  "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  
    }  
  }  
}
```

Here's what happened:

- AWS AppSync received an `addPost` mutation request.
- AWS AppSync took the request, and the request mapping template, and generated a request mapping document. This would have looked like:

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "123" }
  },
  "attributeValues" : {
    "author": { "S" : "AUTHORNAME" },
    "title": { "S" : "Our first post!" },
    "content": { "S" : "This is our first post." },
    "url": { "S" : "https://aws.amazon.com/appsync/" },
    "ups" : { "N" : 1 },
    "downs" : { "N" : 0 },
    "version" : { "N" : 1 }
  }
}
```

- AWS AppSync used the request mapping document to generate and execute a `DynamoDBPutItem` request.
- AWS AppSync took the results of the `PutItem` request and converted them back to GraphQL types.

```
{
  "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
}
```

- Passed it through the response mapping document, which just passed it through unchanged.
- Returned the newly created object in the GraphQL response.

## Setting Up the `getPost` Resolver (DynamoDB `GetItem`)

Now that you're able to add data to the `AppSyncTutorial-PostDynamoDB` table, you need to set up the `getPost` query so it can retrieve that data from the `AppSyncTutorial-Post` table. To do this, you set up another resolver.

- Choose the **Schema** tab.
- In the **Data types** pane on the right, find the `getPost` field on the **Query** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

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



```
}
```

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

- Choose **Save**.

## Call the API to Get a Post

Now the resolver has been set up, AWS AppSync knows how to translate an incoming `getPost` query to a `DynamoDBGetItem` operation. You can now run a query to retrieve the post you created earlier.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following:

```
query getPost {  
  getPost(id:123) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

- Choose **Execute query** (the orange play button).
- The post retrieved from DynamoDB should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{  
  "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  
    }  
  }  
}
```

Here's what happened:

- AWS AppSync received a `getPost` query request.
- AWS AppSync took the request, and the request mapping template, and generated a request mapping document. This would have looked like:

```
{  
  "version" : "2017-02-28",  
  "operation" : "GetItem",  
  "key" : {
```

```
      "id" : { "S" : "123" }  
    }  
  }
```

- AWS AppSync used the request mapping document to generate and execute a DynamoDB GetItem request.
- AWS AppSync took the results of the GetItem request and converted it back to GraphQL types.

```
{  
  "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  
}
```

- Passed it through the response mapping document, which just passed it through unchanged.
- Returned the retrieved object in the response.

## Create an updatePost Mutation (DynamoDB UpdateItem)

So far you can create and retrieve `Post` objects in DynamoDB. Next, you'll set up a new mutation to allow us to update object. You'll do this using the `UpdateItem` DynamoDB operation.

- Choose the **Schema** tab.
- In the **Schema** pane, modify the **Mutation** type to add a new `updatePost` mutation as follows:

```
type Mutation {  
  updatePost(  
    id: ID!,  
    author: String!,  
    title: String!,  
    content: String!,  
    url: String!  
  ): Post  
  addPost(  
    author: String!  
    title: String!  
    content: String!  
    url: String!  
  ): Post!  
}
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **updatePost** field on the **Mutation** type and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

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

**Note:** This resolver is using the DynamoDB UpdateItem, which is significantly different from the PutItem operation. Instead of writing the entire item, you're just asking DynamoDB to update certain attributes. This is done using DynamoDB Update Expressions. The expression itself is specified in the expression field in the update section. It says to set the author, title, content and url attributes, and then increment the version field. The values to use do not appear in the expression itself; the expression has placeholders that have names starting with a colon, which are then defined in the expressionValues field. Finally, DynamoDB has reserved words that cannot appear in the expression. For example, url is a reserved word, so to update the url field you can use name placeholders and define them in the expressionNames field.

For more info about UpdateItem request mapping, see the [UpdateItem \(p. 325\)](#) reference documentation. For more information about how to write update expressions, see the [DynamoDB UpdateExpressions](#) documentation.

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

## Call the API to Update a Post

Now the resolver has been set up, AWS AppSync knows how to translate an incoming update mutation to a DynamoDB Update operation. You can now run a mutation to update the item you wrote earlier.

- Choose the **Queries** tab.
- In **Queries** pane, paste the following mutation. You'll also need to update the id argument to the value you noted down earlier.

```
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
  }
}
```

- Choose **Execute query** (the orange play button).
- The updated post in DynamoDB should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

In this example, the `ups` and `downs` fields were not modified because the request mapping template did not ask AWS AppSync and DynamoDB to do anything with those fields. Also, the `version` field was incremented by 1 because you asked AWS AppSync and DynamoDB to add 1 to the `version` field.

## Modifying the updatePost Resolver (DynamoDB UpdateItem)

This is a good start to the `updatePost` mutation, but it has two main problems:

- If you want to update just a single field, you have to update all of the fields.
- If two people are modifying the object, you could potentially lose information.

To address these issues, you're going to modify the `updatePost` mutation to only modify arguments that were specified in the request, and then add a condition to the `UpdateItem` operation.

- Choose the **Schema** tab.
- In the **Schema** pane, modify the `updatePost` field in the **Mutation** type to remove the exclamation marks from the `author`, `title`, `content`, and `url` arguments, making sure to leave the `id` field as is. This will make them optional argument. Also, add a new, required `expectedVersion` argument.

```
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!  
}
```

- Choose **Save**.
- In the **Data types** pane on the right, find the **updatePost** field on the **Mutation** type.
- Choose **PostDynamoDBTable** to open the existing resolver.
- In **Configure the request mapping template**, modify the request mapping template as follows:

```
{  
  "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)
    }
}
}
```

- Choose **Save**.

This template is one of the more complex examples. It demonstrates the power and flexibility of mapping templates. It loops through all of the arguments, skipping over `id` and `expectedVersion`. If the argument is set to something, it asks AWS AppSync and DynamoDB to update that attribute on the object in DynamoDB. If the attribute is set to null, it asks AWS AppSync and DynamoDB to remove that attribute from the post object. If an argument wasn't specified, it leaves the attribute alone. It also increments the `version` field.

Also, there is a new `condition` section. A condition expression enables you tell AWS AppSync and DynamoDB whether or not the request should succeed based on the state of the object already in DynamoDB before the operation is performed. In this case, you only want the `UpdateItem` request to succeed if the `version` field of the item currently in DynamoDB exactly matches the `expectedVersion` argument.

For more information about condition expressions, see the [Condition Expressions \(p. 360\)](#) reference documentation.

## Call the API to Update a Post

Let's try updating the `Post` object with the new resolver:

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following mutation. You'll also need to update the `id` argument to the value you noted down earlier.

```
mutation updatePost {  
  updatePost(  
    id:123  
    title: "An empty story"  
    content: null  
    expectedVersion: 2  
  ) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

- Choose **Execute query** (the orange play button).
- The updated post in DynamoDB should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{  
  "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  
    }  
  }  
}
```

In this request, you asked AWS AppSync and DynamoDB to update the `title` and `content` field only. It left all the other fields alone (other than incrementing the `version` field). You set the `title` attribute to a new value, and removed the `content` attribute from the post. The `author`, `url`, `ups`, and `downs` fields were left untouched.

Try executing the mutation request again, leaving the request exactly as is. You should see a response similar to the following:

```
{  
  "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:
ABCDEFGHIJKLMNPOQRSTUVWXYZABCDEFGHIJKLMNPOQRSTUVWXYZ)"
}
]
```

The request fails because the condition expression evaluates to false:

- The first time you ran the request, the value of the `version` field of the post in DynamoDB was 2, which matched the `expectedVersion` argument. The request succeeded, which meant the `version` field was incremented in DynamoDB to 3.
- The second time you ran the request, the value of the `version` field of the post in DynamoDB was 3, which did not match the `expectedVersion` argument.

This pattern is typically called *optimistic locking*.

A feature of an AWS AppSync DynamoDB resolver is that it returns the current value of the post object in DynamoDB. You can find this in the `data` field in the `errors` section of the GraphQL response. Your application can use this information to decide how it should proceed. In this case, you can see the `version` field of the object in DynamoDB is set to 3, so you could just update the `expectedVersion` argument to 3 and the request would succeed again.

For more information about handling condition check failures, see the [Condition Expressions \(p. 360\)](#) mapping template reference documentation.

## Create upvotePost and downvotePost Mutations (DynamoDB UpdateItem)

The `Post` type has `ups` and `downs` fields to enable record upvotes and downvotes, but so far the API doesn't let us do anything with them. Let's add some mutations to let us upvote and downvote the posts.

- Choose the **Schema** tab.
- In the **Schema** pane, modify the **Mutation** type to add new `upvotePost` and `downvotePost` mutations as follows:

```
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!
}
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **upvotePost** field on the **Mutation** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

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

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **downvotePost** field on the **Mutation** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

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

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

- Choose **Save**.

## Call the API to upvote and downvote a Post

Now the new resolvers have been set up, AWS AppSync knows how to translate an incoming upvotePost or downvote mutation to DynamoDB UpdateItem operation. You can now run mutations to upvote or downvote the post you created earlier.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following mutation. You'll also need to update the `id` argument to the value you noted down earlier.

```
mutation votePost {  
  upvotePost(id:123) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

- Choose **Execute query** (the orange play button).
- The post is updated in DynamoDB and should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{  
  "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  
    }  
  }  
}
```

- Choose **Execute query** a few more times. You should see the `ups` and `version` field incrementing by 1 each time you execute the query.
- Change the query to call the `downvotePost` mutation as follows:

```
mutation votePost {  
  downvotePost(id:123) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

```
}
```

- Choose **Execute query** (the orange play button). This time, you should see the `downs` and `version` field incrementing by 1 each time you execute the query.

```
{
  "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
    }
  }
}
```

## Setting Up the deletePost Resolver (DynamoDB DeleteItem)

The next mutation you want to set up is to delete a post. You'll do this using the `DeleteItem` DynamoDB operation.

- Choose the **Schema** tab.
- In the **Schema** pane, modify the **Mutation** type to add a new `deletePost` mutation as follows:

```
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!
}
```

This time you made the `expectedVersion` field optional, which is explained later when you add the request mapping template.

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **delete** field on the **Mutation** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
  "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
}
```

**Note:** The `expectedVersion` argument is an optional argument. If the caller set an `expectedVersion` argument in the request, the template adds a condition that only allows the `DeleteItem` request to succeed if the item is already deleted or if the `version` attribute of the post in DynamoDB exactly matches the `expectedVersion`. If left out, no condition expression is specified on the `DeleteItem` request. It succeeds regardless of the value of `version`, or whether or not the item exists in DynamoDB.

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

**Note:** Even though you're deleting an item, you can return the item that was deleted, if it was not already deleted.

- Choose **Save**.

For more info about `DeleteItem` request mapping, see the [DeleteItem \(p. 328\)](#) reference documentation.

## Call the API to Delete a Post

Now the resolver has been set up, AWS AppSync knows how to translate an incoming `delete` mutation to a `DynamoDBDeleteItem` operation. You can now run a mutation to delete something in the table.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following mutation. You'll also need to update the `id` argument to the value you noted down earlier.

```
mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choose **Execute query** (the orange play button).
- The post is deleted from DynamoDB. Note that AWS AppSync returns the value of the item that was deleted from DynamoDB, which should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

The value is only returned if this call to `deletePost` was the one that actually deleted it from DynamoDB.

- Choose **Execute query** again.
- The call still succeeds, but no value is returned.

```
{
  "data": {
    "deletePost": null
  }
}
```

Now let's try deleting a post, but this time specifying an `expectedValue`. First though, you'll need to create a new post because you've just deleted the one you've been working with so far.

- In the **Queries** pane, paste the following mutation:

```
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
  }
}
```

- Choose **Execute query** (the orange play button).

- The results of the newly created post should appear in the results pane to the right of the query pane. Note down the `id` of the newly created object because you need it in just a moment. It should look similar to the following:

```
{
  "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
    }
  }
}
```

Now let's try to delete that post, but put in the wrong value for `expectedVersion`:

- In the **Queries** pane, paste the following mutation. You'll also need to update the `id` argument to the value you noted down earlier.

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choose **Execute query** (the orange play button).

```
{
  "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:
ABCDEFGHIJKLMNPOQRSTUVWXYZABCDEFGHIJKLMNPOQRSTUVWXYZ)"
}
```

The request failed because the condition expression evaluates to false: the value for version of the post in DynamoDB does not match the expectedValue specified in the arguments. The current value of the object is returned in the data field in the errors section of the GraphQL response.

- Retry the request, but correct the expectedVersion:

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Choose **Execute query** (the orange play button).
- This time the request succeeds, and the value that was deleted from DynamoDB is returned:

```
{
  "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
    }
  }
}
```

- Choose **Execute query** again.
- The call still succeeds, but this time no value is returned because the post was already deleted in DynamoDB.

```
{
  "data": {
    "deletePost": null
  }
}
```

```
}  
}
```

## Setting Up the allPost Resolver (DynamoDB Scan)

So far the API is only useful if you know the `id` of each post you want to look at. Let's add a new resolver that returns all the posts in the table.

- Choose the **Schema** tab.
- In the **Schema** pane, modify the **Query** type to add a new `allPost` query as follows:

```
type Query {  
  allPost(count: Int, nextToken: String): PaginatedPosts!  
  getPost(id: ID): Post  
}
```

- Add a new `PaginatedPosts` type:

```
type PaginatedPosts {  
  posts: [Post!]!  
  nextToken: String  
}
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **allPost** field on the **Query** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{  
  "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  
}
```

This resolver has two optional arguments: `count`, which specifies the maximum number of items to return in a single call, and `nextToken`, which can be used to retrieve the next set of results (you'll show where the value for `nextToken` comes from later).

- In **Configure the response mapping template**, paste the following:

```
{  
  "posts": $utils.toJson($context.result.items)  
  #if( ${context.result.nextToken} )  
    , "nextToken": $util.toJson($context.result.nextToken)  
  #end  
}
```

**Note:** This response mapping template is different from all the others so far. The result of the `allPost` query is a `PaginatedPosts`, which contains a list of posts and a pagination token. The shape of this object is different to what is returned from the AWS AppSync DynamoDB Resolver: the list of posts is called `items` in the AWS AppSync DynamoDB Resolver results, but is called `posts` in `PaginatedPosts`.



- Choose **Save**.

For more information about Scan request mapping, see the [Scan \(p. 333\)](#) reference documentation.

## Call the API to Scan All Posts

Now the resolver has been set up, AWS AppSync knows how to translate an incoming allPost query to a DynamoDBScan operation. You can now scan the table to retrieve all the posts.

Before you can try it out though, you need to populate the table with some data because you've deleted everything you've worked with so far.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following mutation:

```
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 }
}
```

- Choose **Execute query** (the orange play button).

Now, let's scan the table, returning five results at a time.

- In the **Queries** pane, paste the following query:

```
query allPost {
  allPost(count: 5) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).
- The first five posts should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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":
"eyJ2ZXJzaW9uIjoxLCJ0b2t1biI6IkFRSUNBSG04eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI3Z0lnRkU="
}
```

You got five results and a `nextToken` that you can use to get the next set of results.

- Update the `allPost` query to include the `nextToken` from the previous set of results:

```
query allPost {
  allPost(
    count: 5
    nextToken:
    "eyJ2ZXJzaW9uIjoxLCJ0b2t1biI6IkFRSUNBSG04eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI3Z0lnRkU="
  ) {
    posts {
      id
      author
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).
- The remaining four posts should appear in the results pane to the right of the query pane. There is no `nextToken` in this set of results because you've paged through all nine posts, with none remaining. It should look similar to the following:

```
{
  "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
}
```

## Setting Up the allPostsByAuthor Resolver (DynamoDB Query)

In addition to scanning DynamoDB for all posts, you can also query DynamoDB to retrieve posts created by a specific author. The DynamoDB table you created earlier already has a `GlobalSecondaryIndex` called `author-index` you can use with a `DynamoDBQuery` operation to retrieve all posts created by a specific author.

- Choose the **Schema** tab.
- In the **Schema** pane, modify the `Query` type to add a new `allPostsByAuthor` query as follows:

```
type Query {
  allPostsByAuthor(author: String!, count: Int, nextToken: String): PaginatedPosts!
  allPost(count: Int, nextToken: String): PaginatedPosts!
  getPost(id: ID): Post
}
```

**Note:** This uses the same `PaginatedPosts` type that you used with the `allPost` query.

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created `allPostsByAuthor` field on the **Query** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
  "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
}
```

Like the `allPost` resolver, this resolver has two optional arguments: `count`, which specifies the maximum number of items to return in a single call, and `nextToken`, which can be used to retrieve the next set of results (the value for `nextToken` can be obtained from a previous call).

- In **Configure the response mapping template**, paste the following:

```
{
  "posts": $utils.toJson($context.result.items)
  #if( ${context.result.nextToken} )
    , "nextToken": $util.toJson($context.result.nextToken)
  #end
}
```

**Note:** This is the same response mapping template that you used in the `allPost` resolver.

- Choose **Save**.

For more information about `Query` request mapping, see the [Query \(p. 330\)](#) reference documentation.

## Call the API to Query All Posts by an Author

Now the resolver has been set up, AWS AppSync knows how to translate an incoming `allPostsByAuthor` mutation to a `DynamoDBQuery` operation against the `author-index` index. You can now query the table to retrieve all the posts by a specific author.

Before you do that, however, let's populate the table with some more posts, because every post so far has the same author.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following mutation:

```
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 }
}
```

- Choose **Execute query** (the orange play button).

Now, let's query the table, returning all posts authored by Nadia.

- In the **Queries** pane, paste the following query:

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).

- All the posts authored by Nadia should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

Pagination works for Query just the same as it does for Scan. For example, let's look for all posts by AUTHORNAME, getting five at a time.

- In the **Queries** pane, paste the following query:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).
- All the posts authored by AUTHORNAME should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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":
"eyJ2ZXJzaW9uIjoxLCJ0b2t1biI6IkFRSUNBSG04eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI3Z0lnSEx"
  }
}
```

- Update the `nextToken` argument with the value returned from the previous query as follows:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
    nextToken:
"eyJ2ZXJzaW9uIjoxLCJ0b2t1biI6IkFRSUNBSG04eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI3Z0lnSEx"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).
- The remaining posts authored by `AUTHORNAME` should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "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
    }
  }
}
```

## Using Sets

Up to this point the `Post` type has been a flat key/value object. You can also model complex objects with the AWS AppSyncDynamoDB resolver, such as sets, lists, and maps.

Let's update the `Post` type to include tags. A post can have 0 or more tags, which are stored in DynamoDB as a String Set. You'll also set up some mutations to add and remove tags, and a new query to scan for posts with a specific tag.

- Choose the **Schema** tab.
- In the **Schema** pane, modify the `Post` type to add a new `tags` field as follows:

```
type Post {  
  id: ID!  
  author: String  
  title: String  
  content: String  
  url: String  
  ups: Int!  
  downs: Int!  
  version: Int!  
  tags: [String!]  
}
```

- In the **Schema** pane, modify the `Query` type to add a new `allPostsByTag` query as follows:

```
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  
}
```

- In the **Schema** pane, modify the `Mutation` type to add new `addTag` and `removeTag` mutations as follows:

```
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!  
}
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created `allPostsByTag` field on the **Query** type, and then choose **Attach**.

- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
  "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
}
```

- In **Configure the response mapping template**, paste the following:

```
{
  "posts": $utils.toJson($context.result.items)
  #if( ${context.result.nextToken} )
    , "nextToken": $util.toJson($context.result.nextToken)
  #end
}
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **addTag** field on the **Mutation** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

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

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **removeTag** field on the **Mutation** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:



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

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

- Choose **Save**.

## Call the API to Work with Tags

Now that you've set up the resolvers, AWS AppSync knows how to translate incoming `addTag`, `removeTag`, and `allPostsByTag` requests into `DynamoDBUpdateItem` and `Scan` operations.

To try it out, let's select one of the posts you created earlier. For example, let's use a post authored by Nadia.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following query:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).
- All of Nadia's posts should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
```

```
        "title": "Did you know...?"
      }
    ],
    "nextToken": null
  }
}
```

- Let's use the one with the title "The cutest dog in the world". Note down its id because you'll use it later.

Now let's try adding a dog tag.

- In the **Queries** pane, paste the following mutation. You'll also need to update the id argument to the value you noted down earlier.

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

- Choose **Execute query** (the orange play button).
- The post is updated with the new tag.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

You can add more tags as follows:

- Update the mutation to change the tag argument to puppy.

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- Choose **Execute query** (the orange play button).
- The post is updated with the new tag.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
```

```
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

You can also delete tags:

- In the **Queries** pane, paste the following mutation. You'll also need to update the `id` argument to the value you noted down earlier.

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- Choose **Execute query** (the orange play button).
- The post is updated and the puppy tag is deleted.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

You can also search for all posts that have a tag:

- In the **Queries** pane, paste the following query:

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

- Choose **Execute query** (the orange play button).
- All posts that have the `dog` tag are returned as follows:

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
```

```
        "id": "10",
        "title": "The cutest dog in the world",
        "tags": [
            "dog",
            "puppy"
        ]
    },
    ],
    "nextToken": null
}
}
```

## Using Lists and Maps

In addition to using DynamoDB sets, you can also use DynamoDB lists and maps to model complex data in a single object.

Let's add the ability to add comments to posts. This will be modeled as a list of map objects on the `Post` object in DynamoDB.

**Note:** in a real application, you would model comments in their own table. For this tutorial, you'll just add them in the `Post` table.

- Choose the **Schema** tab.
- In the **Schema** pane, add a new `Comment` type as follows:

```
type Comment {
  author: String!
  comment: String!
}
```

- In the **Schema** pane, modify the `Post` type to add a new `comments` field as follows:

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
  comments: [Comment!]
}
```

- In the **Schema** pane, modify the `Mutation` type to add a new `addComment` mutation as follows:

```
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!
}
```

- Choose **Save**.
- In the **Data types** pane on the right, find the newly created **addComment** field on the **Mutation** type, and then choose **Attach**.
- In **Data source name**, choose **PostDynamoDBTable**.
- In **Configure the request mapping template**, paste the following:

```
{
  "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)
    }
  }
}
```

This update expression will append a list containing our new comment to the existing `comments` list. If the list doesn't already exist, it will be created.

- In **Configure the response mapping template**, paste the following:

```
$utils.toJson($context.result)
```

- Choose **Save**.

## Call the API to Add a Comment

Now that you've set up the resolvers, AWS AppSync knows how to translate incoming `addComment` requests into `DynamoDBUpdateItem` operations.

Let's try it out by adding a comment to the same post you added the tags to.

- Choose the **Queries** tab.
- In the **Queries** pane, paste the following query:

```
mutation addComment {  
  addComment(  
    id:10  
    author: "Steve"  
    comment: "Such a cute dog."  
  ) {  
    id  
    comments {  
      author  
      comment  
    }  
  }  
}
```

- Choose **Execute query** (the orange play button).
- All of Nadia's posts should appear in the results pane to the right of the query pane. It should look similar to the following:

```
{  
  "data": {  
    "addComment": {  
      "id": "10",  
      "comments": [  
        {  
          "author": "Steve",  
          "comment": "Such a cute dog."  
        }  
      ]  
    }  
  }  
}
```

If you execute the request multiple times, multiple comments will be appended to the list.

## Conclusion

In this tutorial, you've built an API that lets us manipulate Post objects in DynamoDB using AWS AppSync and GraphQL. For more information, see the [Resolver Mapping Template Reference \(p. 264\)](#).

To clean up, you can delete the AppSync GraphQL API from the console.

To delete the DynamoDB table and the IAM role you created for this tutorial, you can run the following to delete the `AWSAppSyncTutorialForAmazonDynamoDB` stack, or visit the AWS CloudFormation console and delete the stack:

```
aws cloudformation delete-stack \  
  --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

## Tutorial: Lambda resolvers

You can use AWS Lambda with AWS AppSync to resolve any GraphQL field. For example, a GraphQL query might send a call to an Amazon Relational Database Service (Amazon RDS) instance, and a GraphQL mutation might write to an Amazon Kinesis stream. In this section, we'll show you how to write a Lambda function that performs business logic based on the invocation of a GraphQL field operation.

## Create a Lambda function

The following example shows a Lambda function written in Node.js that performs different operations on blog posts as part of a blog post application.

```
exports.handler = (event, context, callback) => {
  console.log("Received event {}", JSON.stringify(event, 3));
  var posts = [
    {
      "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"},
    {
      "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"},
    {
      "id": "3", "title": "Third book", "author": "Author3", "url": null, "content": null, "ups": null, "downs": null },
    {
      "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", "ups": "1000", "downs": "0"},
    {
      "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;
    }
};
```

This Lambda function retrieves a post by ID, adds a post, retrieves a list of posts, and fetches related posts for a given post.

**Note:** The Lambda function uses the `switch` statement on `event.field` to determine which field is currently being resolved.

Create this Lambda function using the AWS Management Console or an AWS CloudFormation stack. To create the function from a CloudFormation stack, you can use the following AWS Command Line Interface (AWS CLI) command:

```
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
```

You can also launch the AWS CloudFormation stack in the US West (Oregon) AWS Region in your AWS account from here:



## Configure a data source for Lambda

After you create the Lambda function, navigate to your GraphQL API in the AWS AppSync console, and then choose the **Data Sources** tab.

Choose **Create data source**, enter a friendly **Data source name** (for example, **Lambda**), and then for **Data source type**, choose **AWS Lambda function**. For **Region**, choose the same Region as your function. (If you created the function from the provided CloudFormation stack, the function is probably in **US-WEST-2**.) For **Function ARN**, choose the Amazon Resource Name (ARN) of your Lambda function.

After choosing your Lambda function, you can either create a new AWS Identity and Access Management (IAM) role (for which AWS AppSync assigns the appropriate permissions) or choose an existing role that has the following inline policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/LAMBDA_FUNCTION"
    }
  ]
}
```

You must also set up a trust relationship with AWS AppSync for the IAM role as follows:

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



```
        "Principal": {
            "Service": "appsync.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
    }
]
```

## Create a GraphQL schema

Now that the data source is connected to your Lambda function, create a GraphQL schema.

From the schema editor in the AWS AppSync console, make sure that your schema matches the following schema:

```
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]
}
```

## Configure resolvers

Now that you've registered a Lambda data source and a valid GraphQL schema, you can connect your GraphQL fields to your Lambda data source using resolvers.

To create a resolver, you'll need mapping templates. To learn more about mapping templates, see [Resolver Mapping Template Overview \(p. 264\)](#).

For more information about Lambda mapping templates, see [Resolver mapping template reference for Lambda \(p. 375\)](#).

In this step, you attach a resolver to the Lambda function for the following fields: `getPost(id:ID!): Post`, `allPosts: [Post]`, `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`, and `Post.relatedPosts: [Post]`.

From the schema editor in the AWS AppSync console, on the right side, choose **Attach Resolver** for `getPost(id:ID!): Post`.

Choose your Lambda data source. In the **request mapping template** section, choose **Invoke And Forward Arguments**.

Modify the payload object to add the field name. Your template should look like the following:

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $utils.toJson($context.arguments)
  }
}
```

In the **response mapping template** section, choose **Return Lambda Result**.

In this case, use the base template as-is. It should look like the following:

```
$utils.toJson($context.result)
```

Choose **Save**. You have successfully attached your first resolver. Repeat this operation for the remaining fields as follows:

For `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` request mapping template:

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "addPost",
    "arguments": $utils.toJson($context.arguments)
  }
}
```

For `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` response mapping template:

```
$utils.toJson($context.result)
```

For `allPosts: [Post]` request mapping template:

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "allPosts"
  }
}
```

For `allPosts: [Post]` response mapping template:

```
$utils.toJson($context.result)
```

For `Post.relatedPosts: [Post]` request mapping template:

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "relatedPosts",

```

```
        "source": $utils.toJson($context.source)
    }
}
```

For `Post.relatedPosts`: `[Post]` response mapping template:

```
$utils.toJson($context.result)
```

## Test your GraphQL API

Now that your Lambda function is connected to GraphQL resolvers, you can run some mutations and queries using the console or a client application.

On the left side of the AWS AppSync console, choose **Queries**, and then paste in the following code:

### addPost Mutation

```
mutation addPost {
  addPost(
    id: 6
    author: "Author6"
    title: "Sixth book"
    url: "https://www.amazon.com/"
    content: "This is the book is a tutorial for using GraphQL with AWS AppSync."
  ) {
    id
    author
    title
    content
    url
    ups
    downs
  }
}
```

### getPost Query

```
query {
  getPost(id: "2") {
    id
    author
    title
    content
    url
    ups
    downs
  }
}
```

### allPosts Query

```
query {
  allPosts {
    id
    author
    title
    content
    url
  }
}
```

```
    ups
    downs
    relatedPosts {
      id
      title
    }
  }
}
```

## Returning errors

Any given field resolution can result in an error. With AWS AppSync, you can raise errors from the following sources:

- Request or response mapping template
- Lambda function

### From the mapping template

To raise intentional errors, you can use the `$utils.error` helper method from the Velocity Template Language (VTL) template. It takes as argument an `errorMessage`, an `errorType`, and an optional data value. The data is useful for returning extra data back to the client when an error occurs. The data object is added to the `errors` in the GraphQL final response.

The following example shows how to use it in the `Post.relatedPosts: [Post]` response mapping template:

```
$utils.error("Failed to fetch relatedPosts", "LambdaFailure", $context.result)
```

This yields a GraphQL response similar to the following:

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

Where `allPosts[0].relatedPosts` is *null* because of the error and the `errorMessage`, `errorType`, and `data` are present in the `data.errors[0]` object.

## From the Lambda function

AWS AppSync also understands errors that the Lambda function throws. The Lambda programming model lets you raise *handled* errors. If the Lambda function throws an error, AWS AppSync fails to resolve the current field. Only the error message returned from Lambda is set in the response. Currently, you can't pass any extraneous data back to the client by raising an error from the Lambda function.

**Note:** If your Lambda function raises an *unhandled* error, AWS AppSync uses the error message that Lambda set.

The following Lambda function raises an error:

```
exports.handler = (event, context, callback) => {
  console.log("Received event {}", JSON.stringify(event, 3));
  callback("I fail. Always.");
};
```

This returns a GraphQL response similar to the following:

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

```
}
```

## Advanced use case: Batching

The Lambda function in this example has a `relatedPosts` field that returns a list of related posts for a given post. In the example queries, the `allPosts` field invocation from the Lambda function returns five posts. Because we specified that we also want to resolve `relatedPosts` for each returned post, the `relatedPosts` field operation is invoked five times.

```
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
    }
  }
}
```

While this might not sound substantial in this specific example, this compounded over-fetching can quickly undermine the application.

If you were to fetch `relatedPosts` again on the returned related Posts in the same query, the number of invocations would increase dramatically.

```
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
      }
    }
  }
}
```

In this relatively simple query, AWS AppSync would invoke the Lambda function  $1 + 5 + 25 = 31$  times.

This is a fairly common challenge and is often called the N+1 problem (in this case,  $N = 5$ ), and it can incur increased latency and cost to the application.

One approach to solving this issue is to batch similar field resolver requests together. In this example, instead of having the Lambda function resolve a list of related posts for a single given post, it could instead resolve a list of related posts for a given batch of posts.

To demonstrate this, let's switch the `Post.relatedPosts: [Post]` resolver to a batch-enabled resolver.

On the right side of the AWS AppSync console, choose the existing `Post.relatedPosts: [Post]` resolver. Change the request mapping template to the following:

```
{
  "version": "2017-02-28",
  "operation": "BatchInvoke",
  "payload": {
    "field": "relatedPosts",
    "source": $utils.toJson($context.source)
  }
}
```

Only the `operation` field has changed from `Invoke` to `BatchInvoke`. The `payload` field now becomes an array of whatever is specified in the template. In this example, the Lambda function receives the following as input:

```
[
  {
    "field": "relatedPosts",
    "source": {
      "id": 1
    }
  },
  {
    "field": "relatedPosts",
    "source": {
      "id": 2
    }
  },
  ...
]
```

When `BatchInvoke` is specified in the request mapping template, the Lambda function receives a list of requests and returns a list of results.

Specifically, the list of results must match the size and order of the request payload entries so that AWS AppSync can match the results accordingly.

In this batching example, the Lambda function returns a batch of results as follows:

```
[
  [{ "id": "2", "title": "Second book" }, { "id": "3", "title": "Third book" } ], // relatedPosts
  for id=1
  [{ "id": "3", "title": "Third book" } ]
  // relatedPosts for id=2
]
```

The following Lambda function in Node.js demonstrates this batching functionality for the `Post.relatedPosts` field as follows:

```
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;
    }
};

```

## Returning individual errors

The previous examples show that it's possible to return a single error from the Lambda function or raise an error from the mapping templates. For batched invocations, raising an error from the Lambda function flags an entire batch as failed. This might be acceptable for specific scenarios where an irrecoverable error occurs, such as a failed connection to a data store. However, in cases where some items in the batch succeed and others fail, it's possible to return both errors and valid data. Because AWS AppSync requires the batch response to list elements matching the original size of the batch, you must define a data structure that can differentiate valid data from an error.

For example, if the Lambda function is expected to return a batch of related posts, you could choose to return a list of Response objects where each object has optional *data*, *errorMessage*, and *errorType* fields. If the *errorMessage* field is present, it means that an error occurred.

The following code shows how you could update the Lambda function:

```

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", "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", "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;
    }
  }
};
```

For this example, the following response mapping template parses each item of the Lambda function and raises any errors that occur:

```
#if( $context.result && $context.result.errorMessage )
  $utils.error($context.result.errorMessage, $context.result.errorType,
    $context.result.data)
#else
  $utils.toJson($context.result.data)
#end
```

This example returns a GraphQL response similar to the following:

```
{
  "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"
  }
]
```

## Configuring the maximum batching size

By default, when using `BatchInvoke`, AWS AppSync sends requests to your Lambda function in batches of up to five items. You can configure the maximum batch size of your Lambda resolvers.

To configure the maximum batching size on a resolver, use the following command in the AWS Command Line Interface (AWS CLI):

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name relatedPosts \
  --request-mapping-template "<template>" --response-mapping-template "<template>" --data-
source-name "<lambda-datasource>" \
  --max-batch-size X
```

### Note

When providing a request mapping template, you must use the `BatchInvoke` operation to use batching.

You can also use the following command to enable and configure batching on Direct Lambda Resolvers:

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name relatedPosts \
  --data-source-name "<lambda-datasource>" \
  --max-batch-size X
```

## Maximum batching size configuration with VTL templates

For Lambda Resolvers that have VTL in-request templates, the maximum batch size will have no effect unless they have directly specified it as a `BatchInvoke` operation in VTL. Similarly, if you are performing a top-level mutation, batching is not conducted for mutations because the GraphQL specification requires parallel mutations to be executed sequentially.

For example, take the following mutations:

```
type Mutation {
  putItem(input: Item): Item
  putItems(inputs: [Item]): [Item]
}
```

Using the first mutation, we can create 10 Items as shown in the snippet below:

```
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
}
}
```

In this example, the `Items` will not be batched in a group of 10 even if the maximum batch size is set to 10 in the Lambda Resolver. Instead, they will execute sequentially according to the GraphQL specification.

To perform an actual batch mutation, you may follow the example below using the second mutation:

```
mutation MyMutation {
  putItems([$someItem1, $someItem2, $someItem3,$someItem4, $someItem5, $someItem6,
    $someItem7, $someItem8, $someItem9, $someItem10]) {
    id,
    name
  }
}
```

For more information about using batching with Direct Lambda Resolvers, see [Direct Lambda Resolvers \(p. 378\)](#).

## Tutorial: Amazon OpenSearch Service Resolvers

AWS AppSync supports using Amazon OpenSearch Service from domains that you have provisioned in your own AWS account, provided they don't exist inside a VPC. After your domains are provisioned, you

can connect to them using a data source, at which point you can configure a resolver in the schema to perform GraphQL operations such as queries, mutations, and subscriptions. This tutorial will take you through some common examples.

For more information, see the [Resolver Mapping Template Reference for OpenSearch \(p. 372\)](#).

## One-Click Setup

To automatically set up a GraphQL endpoint in AWS AppSync with Amazon OpenSearch Service configured you can use this AWS CloudFormation template:



After the AWS CloudFormation deployment completes you can skip directly to [running GraphQL queries and mutations \(p. 106\)](#).

## Create a New OpenSearch Service Domain

To get started with this tutorial, you need an existing OpenSearch Service domain. If you don't have one, you can use the following sample. Note that it can take up to 15 minutes for an OpenSearch Service domain to be created before you can move on to integrating it with an AWS AppSync data source.

```
aws cloudformation create-stack --stack-name AppSyncOpenSearch \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/elasticsearch/
ESResolverCFTemplate.yaml \
--parameters ParameterKey=ESDomainName,ParameterValue=ddtestdomain
ParameterKey=Tier,ParameterValue=development \
--capabilities CAPABILITY_NAMED_IAM
```

You can launch the following AWS CloudFormation stack in the US West 2 (Oregon) region in your AWS account:



## Configure Data Source for OpenSearch Service

After the OpenSearch Service domain is created, navigate to your AWS AppSync GraphQL API and choose the **Data Sources** tab. Choose **New** and enter a friendly name for the data source, such as "oss". Then choose **Amazon OpenSearch domain** for **Data source type**, choose the appropriate region, and you should see your OpenSearch Service domain listed. After selecting it you can either create a new role and AWS AppSync will assign the role-appropriate permissions, or you can choose an existing role, which has the following inline policy:

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

You'll also need to set up a trust relationship with AWS AppSync for that role:

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

Additionally, the OpenSearch Service domain has its own **Access Policy** which you can modify through the Amazon OpenSearch Service console. You will need to add a policy similar to the following, with the appropriate actions and resource for the OpenSearch Service domain. Note that the **Principal** will be the AppSync data source role, which if you let the console create this, can be found in the IAM console.

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

## Connecting a Resolver

Now that the data source is connected to your OpenSearch Service domain, you can connect it to your GraphQL schema with a resolver, as shown in the following example:

```
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): Post
}

type Post {
  id: ID!
  author: String
  title: String
  url: String
  ups: Int
  downs: Int
  content: String
}
...
```

Note that there is a user-defined `Post` type with a field of `id`. In the following examples, we assume there is a process (which can be automated) for putting this type into your OpenSearch Service domain, which would map to a path root of `/id/post`, where `id` is the index and `post` is the type. From this root path, you can perform individual document searches, wildcard searches with `/id/post*` or multi-document searches with a **path** of `/id/post/_search`. If you have another type `User`, for example, one that is indexed under the same index `id`, you can perform multi-document searches with a **path** of `/id/_search`. This searches for fields on both `Post` and `User`.

From the schema editor in the AWS AppSync console, modify the preceding `Posts` schema to include a `searchPosts` query:

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

Save the schema. On the right side, for `searchPosts`, choose **Attach resolver**. Choose your OpenSearch Service data source. Under the **request mapping template** section, select the drop-down for **Query posts** to get a base template. Modify the path to be `/id/post/_search`. It should look like the following:

```
{
  "version": "2017-02-28",
  "operation": "GET",
  "path": "/id/post/_search",
  "params": {
    "headers": {},
    "queryString": {},
    "body": {
      "from": 0,
      "size": 50
    }
  }
}
```

This assumes that the preceding schema has documents with an `id` field, and that the documents have been indexed in OpenSearch Service by this field. If you structure your data differently, then you'll need to update accordingly.

Under the **response mapping template** section, you need to specify the appropriate `_source` filter if you want to get back the data results from an OpenSearch Service query and translate to GraphQL. Use the following template:

```
[
  #foreach($entry in $context.result.hits.hits)
  #if( $velocityCount > 1 ) , #end
  $utils.toJson($entry.get("_source"))
  #end
]
```

## Modifying Your Searches

The preceding request mapping template performs a simple query for all records. Suppose you want to search by a specific author. Further, suppose you want that author to be an argument defined in your GraphQL query. In the schema editor of the AWS AppSync console, add an `allPostsByAuthor` query:

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

Now choose **Attach resolver** and select the OpenSearch Service data source, but use the following example in the **response mapping template**:

```
{
  "version": "2017-02-28",
  "operation": "GET",
  "path": "/id/post/_search",
  "params": {
    "headers": {},
    "queryString": {},
    "body": {
      "from": 0,
      "size": 50,
      "query": {
        "term": {
          "author": "$util.toJson($context.arguments.author)"
        }
      }
    }
  }
}
```

Note that the `body` is populated with a term query for the `author` field, which is passed through from the client as an argument. You could optionally have prepopulated information, such as standard text, or even use other [utilities](#) (p. 279).

If you're using this resolver, fill in the **response mapping template** with the same information as the previous example.

## Adding Data to OpenSearch Service

You may want to add data to your OpenSearch Service domain as the result of a GraphQL mutation. This is a powerful mechanism for searching and other purposes. Because you can use GraphQL subscriptions



to [make your data real-time \(p. 164\)](#), it serves as a mechanism for notifying clients of updates to data in your OpenSearch Service domain.

Return to the **Schema** page in the AWS AppSync console and select **Attach resolver** for the `addPost()` mutation. Select the OpenSearch Service data source again and use the following **response mapping template** for the `Posts` schema:

```
{
  "version": "2017-02-28",
  "operation": "PUT",
  "path": $util.toJson("/id/post/$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)
    }
  }
}
```

As before, this is an example of how your data might be structured. If you have different field names or indexes, you need to update the `path` and `body` as appropriate. This example also shows how to use `$context.arguments` to populate the template from your GraphQL mutation arguments.

Before moving on, use the following response mapping template, which will be explained more in the next section:

```
$utils.toJson($context.result.get("_source"))
```

## Retrieving a Single Document

Finally, if you want to use the `getPost(id:ID)` query in your schema to return an individual document, find this query in the schema editor of the AWS AppSync console and choose **Attach resolver**. Select the OpenSearch Service data source again and use the following mapping template:

```
{
  "version": "2017-02-28",
  "operation": "GET",
  "path": $util.toJson("/id/post/$context.arguments.id"),
  "params": {
    "headers": {},
    "queryString": {},
    "body": {}
  }
}
```

Because the `path` above uses the `id` argument with an empty body, this returns the single document. However, you need to use the following response mapping template, because now you're returning a single item and not a list:

```
$utils.toJson($context.result.get("_source"))
```

## Perform Queries and Mutations

You should now be able to perform GraphQL operations against your OpenSearch Service domain. Navigate to the **Queries** tab of the AWS AppSync console and add a new record:

```
mutation {
  addPost(
    id: "12345"
    author: "Fred"
    title: "My first book"
    content: "This will be fun to write!"
  ){
    id
    author
    title
  }
}
```

If the record is inserted successfully, you'll see the fields on the right. Similarly, you can now run a `searchPosts` query against your OpenSearch Service domain:

```
query {
  searchPosts {
    id
    title
    author
    content
  }
}
```

## Best Practices

- OpenSearch Service should be for querying data, not as your primary database. You may want to use OpenSearch Service in conjunction with Amazon DynamoDB as outlined in [Combining GraphQL Resolvers \(p. 108\)](#).
- Only give access to your domain by allowing the AWS AppSync service role to access the cluster.
- You can start small in development, with the lowest-cost cluster, and then move to a larger cluster with high availability (HA) as you move into production.

## Tutorial: Local Resolvers

AWS AppSync allows you to use supported data sources (AWS Lambda, Amazon DynamoDB, or Amazon OpenSearch Service) to perform various operations. However, in certain scenarios, a call to a supported data source might not be necessary.

This is where the local resolver comes in handy. Instead of calling a remote data source, the local resolver will just **forward** the result of the request mapping template to the response mapping template. The field resolution will not leave AWS AppSync.

Local resolvers are useful for several use cases. The most popular use case is to publish notifications without triggering a data source call. To demonstrate this use case, let's build a paging application; where users can page each other. This example leverages *Subscriptions*, so if you aren't familiar with *Subscriptions*, you can follow the [Real-Time Data \(p. 164\)](#) tutorial.

## Create the Paging Application

In our paging application, clients can subscribe to an inbox, and send pages to other clients. Each page includes a message. Here is the schema:

```
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
}
```

Let's attach a resolver on the `Mutation.page` field. In the **Schema** pane, click on *Attach Resolver* next to the field definition on the right panel. Create a new data source of type *None* and name it *PageDataSource*.

For the request mapping template, enter:

```
{
  "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()"
  }
}
```

And for the response mapping template, select the default *Forward the result*. Save your resolver. Your application is now ready, let's page!

## Send and subscribe to pages

For clients to receive pages, they must first be subscribed to an inbox.

In the **Queries** pane let's execute the `inbox` subscription:

```
subscription Inbox {
  inbox(to: "Nadia") {
    body
  }
}
```

```
        to
        from
        sentAt
    }
}
```

*Nadia* will receive pages whenever the `Mutation.page` mutation is invoked. Let's invoke the mutation by executing the mutation:

```
mutation Page {
  page(to: "Nadia", body: "Hello, World!") {
    body
    to
    from
    sentAt
  }
}
```

We just demonstrated the use of local resolvers, by sending a `Page` and receiving it without leaving AWS AppSync.

## Tutorial: Combining GraphQL Resolvers

Resolvers and fields in a GraphQL schema have 1:1 relationships with a large degree of flexibility. Because a data source is configured on a resolver independently of a schema, you have the ability for GraphQL types to be resolved or manipulated through different data sources, mixing and matching on a schema to best meet your needs.

The following example scenarios demonstrate how to mix and match data sources in your schema. Before you begin, we recommend that you are familiar with setting up data sources and resolvers for AWS Lambda, Amazon DynamoDB, and Amazon OpenSearch Service as described in the previous tutorials.

### Example Schema

The following schema has a type of `Post` with 3 `Query` operations and 3 `Mutation` operations defined:

```
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  
}
```

In this example you would have a total of 6 resolvers to attach. One possible way would to have all of these come from an Amazon DynamoDB table, called `Posts`, where `AllPosts` runs a scan and `searchPosts` runs a query, as outlined in the [DynamoDB Resolver Mapping Template Reference \(p. 321\)](#). However, there are alternatives to meet your business needs, such as having these GraphQL queries resolve from Lambda or OpenSearch Service.

## Alter Data Through Resolvers

You might have the need to return results from a database such as DynamoDB (or Amazon Aurora) to clients with some of the attributes changed. This might be due to formatting of the data types, such as timestamp differences on clients, or to handle backwards compatibility issues. For illustrative purposes, in the following example, an AWS Lambda function manipulates the up-votes and down-votes for blog posts by assigning them random numbers each time the GraphQL resolver is invoked:

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

This is a perfectly valid Lambda function and could be attached to the `AllPosts` field in the GraphQL schema so that any query returning all the results gets random numbers for the ups/downs.

## DynamoDB and OpenSearch Service

For some applications, you might perform mutations or simple lookup queries against DynamoDB, and have a background process transfer documents to OpenSearch Service. You can then simply attach the `searchPosts` Resolver to the OpenSearch Service data source and return search results (from data that

originated in DynamoDB) using a GraphQL query. This can be extremely powerful when adding advanced search operations to your applications such keyword, fuzzy word matches or even geospatial lookups. Transferring data from DynamoDB could be done through an ETL process or alternatively you can stream from DynamoDB using Lambda. You can launch a complete example of this using the following AWS CloudFormation stack in the US West 2 (Oregon) Region in your AWS account:



The schema in this example lets you add posts using a DynamoDB resolver as follows:

```
mutation add {
  putPost(author:"Nadia"
    title:"My first post"
    content:"This is some test content"
    url:"https://aws.amazon.com/appsync/")
  {
    id
    title
  }
}
```

This writes data to DynamoDB which then streams data via Lambda to Amazon OpenSearch Service which you could search for all posts by different fields. For example, since the data is in Amazon OpenSearch Service you can search either the author or content fields with free-form text, even with spaces, as follows:

```
query searchName{
  searchAuthor(name:"  Nadia  "){
    id
    title
    content
  }
}

query searchContent{
  searchContent(text:"test"){
    id
    title
    content
  }
}
```

Because the data is written directly to DynamoDB, you can still perform efficient list or item lookup operations against the table with the `allPosts{...}` and `singlePost{...}` queries. This stack uses the following example code for DynamoDB streams:

**Note:** This code is for example only.

```
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);
};
```

You can then use DynamoDB streams to attach this to a DynamoDB table with a primary key of `id`, and any changes to the source of DynamoDB would stream into your OpenSearch Service domain. For more information about configuring this, see the [DynamoDB Streams documentation](#).

## Tutorial: DynamoDB Batch Resolvers

AWS AppSync supports using Amazon DynamoDB batch operations across one or more tables in a single region. Supported operations are `BatchGetItem`, `BatchPutItem`, and `BatchDeleteItem`. By using these features in AWS AppSync, you can perform tasks such as:

- Pass a list of keys in a single query and return the results from a table
- Read records from one or more tables in a single query
- Write records in bulk to one or more tables
- Conditionally write or delete records in multiple tables that might have a relation

Using batch operations with DynamoDB in AWS AppSync is an advanced technique that takes a little extra thought and knowledge of your backend operations and table structures. Additionally, batch operations in AWS AppSync have two key differences from non-batched operations:

- The data source role must have permissions to all tables which the resolver will access.
- The table specification for a resolver is part of the mapping template.

## Permissions

Like other resolvers, you need to create a data source in AWS AppSync and either create a role or use an existing one. Because batch operations require different permissions on DynamoDB tables, you need to grant the configured role permissions for read or write actions:

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

**Note:** Roles are tied to data sources in AWS AppSync, and resolvers on fields are invoked against a data source. Data sources configured to fetch against DynamoDB only have one table specified, to keep configuration simple. Therefore, when performing a batch operation against multiple tables in a single resolver, which is a more advanced task, you must grant the role on that data source access to any tables the resolver will interact with. This would be done in the **Resource** field in the IAM policy above. Configuration of the tables to make batch calls against is done in the resolver template, which we describe below.

## Data Source

For the sake of simplicity, we'll use the same data source for all the resolvers used in this tutorial. On the **Data sources** tab, create a new DynamoDB data source and name it **BatchTutorial**. The table name can be anything because table names are specified as part of the request mapping template for batch operations. We will give the table name empty.

For this tutorial, any role with the following inline policy will work:

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

## Single Table Batch

For this example, suppose you have a single table named **Posts** to which you want to add and remove items with batch operations. Use the following schema, noting that for the query, we'll pass in a list of IDs:

```
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
  }
```

Attach a resolver to the `batchAdd()` field with the following **Request Mapping Template**. This automatically takes each item in the GraphQL input `PostInput` type and builds a map, which is needed for the `BatchPutItem` operation:

```
#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)
  }
}
```

In this case, the **Response Mapping Template** is a simple passthrough, but the table name is appended as `..data.Posts` to the context object as follows:

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

Now navigate to the **Queries** page of the AWS AppSync console and run the following **batchAdd** mutation:

```
mutation add {
  batchAdd(posts:[{
    id: 1 title: "Running in the Park",{
    id: 2 title: "Playing fetch"
  }]){
    id
    title
  }
}
```

You should see the results printed to the screen, and can independently validate through the DynamoDB console that both values wrote to the **Posts** table.

Next, attach a resolver to the `batchGet()` field with the following **Request Mapping Template**. This automatically takes each item in the GraphQL `ids: [ ]` type and builds a map that is needed for the `BatchGetItem` operation:

```
#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
    }
  }
}
```

The **Response Mapping Template** is again a simple passthrough, with again the table name appended as `..data.Posts` to the context object:

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

Now go back to the **Queries** page of the AWS AppSync console, and run the following **batchGet Query**:

```
query get {
  batchGet(ids:[1,2,3]){
    id
    title
  }
}
```

This should return the results for the two `id` values that you added earlier. Note that a `null` value returned for the `id` with a value of 3. This is because there was no record in your **Posts** table with that value yet. Also note that AWS AppSync returns the results in the same order as the keys passed in to the query, which is an additional feature that AWS AppSync does on your behalf. So if you switch to `batchGet(ids:[1,3,2])`, you'll see the order changed. You'll also know which `id` returned a `null` value.

Finally, attach a resolver to the `batchDelete()` field with the following **Request Mapping Template**. This automatically takes each item in the GraphQL `ids: [ ]` type and builds a map that is needed for the `BatchDeleteItem` operation:

```
#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)
  }
}
```

```
}
```

The **Response Mapping Template** is again a simple passthrough, with again the table name appended as `..data.Posts` to the context object:

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

Now go back to the **Queries** page of the AWS AppSync console, and run the following **batchDelete** mutation:

```
mutation delete {  
  batchDelete(ids:[1,2]){ id }  
}
```

The records with `id` 1 and 2 should now be deleted. If you re-run the `batchGet()` query from earlier, these should return `null`.

## Multi-Table Batch

AWS AppSync also enables you to perform batch operations across tables. Let's build a more complex application. Imagine we are building a Pet Health app, where sensors report the pet location and body temperature. The sensors are battery powered and attempt to connect to the network every few minutes. When a sensor establishes connection, it sends its readings to our AWS AppSync API. Triggers then analyze the data so a dashboard can be presented to the pet owner. Let's focus on representing the interactions between the sensor and the backend data store.

As a prerequisite, let's first create two DynamoDB tables; **locationReadings** will store sensor location readings and **temperatureReadings** will store sensor temperature readings. Both tables happen to share the same primary key structure: `sensorId` (`String`) being the partition key, and `timestamp` (`String`) the sort key.

Let's use the following GraphQL schema:

```
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 - Recording Sensor Readings

Our sensors need to be able to send their readings once they connect to the internet. The GraphQL field `Mutation.recordReadings` is the API they will use to do so. Let's attach a resolver to bring our API to life.

Select **Attach** next to the `Mutation.recordReadings` field. On the next screen, pick the same `BatchTutorial` data source created at the beginning of the tutorial.

Let's add the following request mapping template:

### Request Mapping Template

```
## 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)
  }
}
```

As you can see, the `BatchPutItem` operation allows us to specify multiple tables.

Let's use the following response mapping template.

### Response Mapping Template

```
## 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)
```

With batch operations, there can be both errors and results returned from the invocation. In that case, we're free to do some extra error handling.

**Note:** The use of `$utils.appendError()` is similar to the `$util.error()`, with the major distinction that it doesn't interrupt the evaluation of the mapping template. Instead, it signals there was an error with the field, but allows the template to be evaluated and consequently return data back to the caller. We recommend you use `$utils.appendError()` when your application needs to return partial results.

Save the resolver and navigate to the **Queries** page of the AWS AppSync console. Let's send some sensor readings!

Execute the following mutation:

```
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
    }
  }
}
```

We sent 10 sensor readings in one mutation, with readings split up across two tables. Use the DynamoDB console to validate that data shows up in both the **locationReadings** and **temperatureReadings** tables.

## BatchDeleteItem - Deleting Sensor Readings

Similarly, we would also need to delete batches of sensor readings. Let's use the `Mutation.deleteReadings` GraphQL field for this purpose. Select **Attach** next to the `Mutation.recordReadings` field. On the next screen, pick the same `BatchTutorial` data source created at the beginning of the tutorial.

Let's use the following request mapping template.

### Request Mapping Template

```
## 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)
  }
}
```

The response mapping template is the same as the one we used for `Mutation.recordReadings`.

### Response Mapping Template

```
## 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)
```

Save the resolver and navigate to the **Queries** page of the AWS AppSync console. Now, let's delete a couple of sensor readings!

Execute the following mutation:

```
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  
  }  
}
```

Validate through the DynamoDB console that these two readings have been deleted from the **locationReadings** and **temperatureReadings** tables.

## BatchGetItem - Retrieve Readings

Another common operation for our Pet Health app would be to retrieve the readings for a sensor at a specific point in time. Let's attach a resolver to the `Query.getReadings` GraphQL field on our schema. Select **Attach**, and on the next screen pick the same `BatchTutorial` data source created at the beginning of the tutorial.

Let's add the following request mapping template.

### Request Mapping Template

```
## 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  
    }  
  }  
}
```

Note that we are now using the **BatchGetItem** operation.

Our response mapping template is going to be a little different because we chose to return a `SensorReading` list. Let's map the invocation result to the desired shape.

### Response Mapping Template

```
## 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)
```

Save the resolver and navigate to the **Queries** page of the AWS AppSync console. Now, let's retrieve sensor readings!

Execute the following query:

```
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
    }
  }
}
```

We have successfully demonstrated the use of DynamoDB batch operations using AWS AppSync.

## Error Handling

In AWS AppSync, data source operations can sometimes return partial results. Partial results is the term we will use to denote when the output of an operation is comprised of some data and an error. Because error handling is inherently application specific, AWS AppSync gives you the opportunity to handle errors in the response mapping template. The resolver invocation error, if present, is available from the context as `$ctx.error`. Invocation errors always include a message and a type, accessible as properties `$ctx.error.message` and `$ctx.error.type`. During the response mapping template invocation, you can handle partial results in three ways:

1. swallow the invocation error by just returning data
2. raise an error (using `$util.error(...)`) by stopping the response mapping template evaluation, which won't return any data.
3. append an error (using `$util.appendError(...)`) and also return data

Let's demonstrate each of the three points above with DynamoDB batch operations!

## DynamoDB Batch operations

With DynamoDB batch operations, it is possible that a batch partially completes. That is, it is possible that some of the requested items or keys are left unprocessed. If AWS AppSync is unable to complete a batch, unprocessed items and an invocation error will be set on the context.

We will implement error handling using the `Query.getReadings` field configuration from the `BatchGetItem` operation from the previous section of this tutorial. This time, let's pretend that while executing the `Query.getReadings` field, the `temperatureReadings` DynamoDB table ran out of provisioned throughput. DynamoDB raised a **ProvisionedThroughputExceededException** at the second attempt by AWS AppSync to process the remaining elements in the batch.

The following JSON represents the serialized context after the DynamoDB batch invocation but before the response mapping template was evaluated.

```
{
  "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": []
}
```

A few things to note on the context:

- the invocation error has been set on the context at `$ctx.error` by AWS AppSync, and the error type has been set to **DynamoDB:ProvisionedThroughputExceededException**.
- results are mapped per table under `$ctx.result.data`, even though an error is present
- keys that were left unprocessed are available at `$ctx.result.data.unprocessedKeys`. Here, AWS AppSync was unable to retrieve the item with key (`sensorId:1, timestamp:2018-02-01T17:21:05.000+08:00`) because of insufficient table throughput.

**Note:** For `BatchPutItem`, it is `$ctx.result.data.unprocessedItems`. For `BatchDeleteItem`, it is `$ctx.result.data.unprocessedKeys`.

Let's handle this error in three different ways.

## 1. Swallowing the invocation error

Returning data without handling the invocation error effectively swallows the error, making the result for the given GraphQL field always successful.

The response mapping template we write is familiar and only focuses on the result data.

Response mapping template:

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

GraphQL response:

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

No errors will be added to the error response as only data was acted on.

## 2. Raising an error to abort the template execution

When partial failures should be treated as complete failures from the client's perspective, you can abort the template execution to prevent returning data. The `$util.error(...)` utility method achieves exactly this behavior.

Response mapping template:

```
## 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 response:

```
{
  "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. (...)"
  }
]
}

```

Even though some results might have been returned from the DynamoDB batch operation, we chose to raise an error such that the `getReadings` GraphQL field is null and the error has been added to the GraphQL response *errors* block.

### 3. Appending an error to return both data and errors

In certain cases, to provide a better user experience, applications can return partial results and notify their clients of the unprocessed items. The clients can decide to either implement a retry or translate the error back to the end user. The `$util.appendError(...)` is the utility method that enables this behavior by letting the application designer append errors on the context without interfering with the evaluation of the template. After evaluating the template, AWS AppSync will process any context errors by appending them to the errors block of the GraphQL response.

Response mapping template:

```

#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)

```

We forwarded both the invocation error and `unprocessedKeys` element inside the errors block of the GraphQL response. The `getReadings` field also return partial data from the **locationReadings** table as you can see in the response below.

GraphQL response:

```

{
  "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. (...)"
  }
]
}
```

## Tutorial: DynamoDB Transaction Resolvers

AWS AppSync supports using Amazon DynamoDB transaction operations across one or more tables in a single region. Supported operations are `TransactGetItems` and `TransactWriteItems`. By using these features in AWS AppSync, you can perform tasks such as:

- Pass a list of keys in a single query and return the results from a table
- Read records from one or more tables in a single query
- Write records in transaction to one or more tables in an all-or-nothing way
- Execute transactions when some conditions are satisfied

### Permissions

Like other resolvers, you need to create a data source in AWS AppSync and either create a role or use an existing one. Because transaction operations require different permissions on DynamoDB tables, you need to grant the configured role permissions for read or write actions:

```
{
  "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:** Roles are tied to data sources in AWS AppSync, and resolvers on fields are invoked against a data source. Data sources configured to fetch against DynamoDB only have one table specified, to keep configuration simple. Therefore, when performing a transaction operation against multiple tables in a single resolver, which is a more advanced task, you must grant the role on that data source access to any tables the resolver will interact with. This would be done in the **Resource** field in the IAM policy above. Configuration of the transaction calls against the tables is done in the resolver template, which we describe below.

## Data Source

For the sake of simplicity, we'll use the same data source for all the resolvers used in this tutorial. On the **Data sources** tab, create a new DynamoDB data source and name it **TransactTutorial**. The table name can be anything because table names are specified as part of the request mapping template for transaction operations. We will give the table name `empty`.

We'll have two tables called **savingAccounts** and **checkingAccounts**, both with `accountNumber` as partition key, and a **transactionHistory** table with `transactionId` as partition key.

For this tutorial, any role with the following inline policy will work. Replace `region` and `accountId` with your region and account ID:

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:dynamodb:region:accountId:table/savingAccounts",
        "arn:aws:dynamodb:region:accountId:table/savingAccounts/*",
        "arn:aws:dynamodb:region:accountId:table/checkingAccounts",
        "arn:aws:dynamodb:region:accountId:table/checkingAccounts/*",
        "arn:aws:dynamodb:region:accountId:table/transactionHistory",
        "arn:aws:dynamodb:region:accountId:table/transactionHistory/*"
      ]
    }
  ]
}

```

## Transactions

For this example, the context is a classic banking transaction, where we'll use `TransactWriteItems` to:

- Transfer money from saving accounts to checking accounts
- Generate new transaction records for each transaction

And then we'll use `TransactGetItems` to retrieve details from saving accounts and checking accounts.

We define our GraphQL schema as follows:

```
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 - Populate Accounts

In order to transfer money between accounts, we need to populate the table with the details. We'll use the GraphQL operation `Mutation.populateAccounts` to do so.

In the Schema section click on **Attach** next to the `Mutation.populateAccounts` operation. On the next screen, select the same `TransactTutorial` data source.

Now use the following request mapping template:

### Request Mapping Template

```
#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)
}
```

And the following response mapping template:



## Response Mapping Template

```
#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)
```

Save the resolver and navigate to the **Queries** section of the AWS AppSync console to populate the accounts.

Execute the following mutation:

```
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
    }
  }
}
```

We populated 3 saving accounts and 3 checking accounts in one mutation.

Use the DynamoDB console to validate that data shows up in both the **savingAccounts** and **checkingAccounts** tables.

## TransactWriteItems - Transfer Money

Attach a resolver to the **transferMoney** mutation with the following **Request Mapping Template**. Note the values of amounts, savingAccountNumbers, and checkingAccountNumbers are the same.

```
#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)
}

```

```
}
```

We will have 3 banking transactions in a single `TransactWriteItems` operation. Use the following **Response Mapping Template**:

```
#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)
```

Now navigate to the **Queries** section of the AWS AppSync console and execute the **transferMoney** mutation as follows:

```
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
    }
  }
}
```

We sent 2 banking transactions in one mutation. Use the DynamoDB console to validate that data shows up in the **savingAccounts**, **checkingAccounts**, and **transactionHistory** tables.

## TransactGetItems - Retrieve Accounts

In order to retrieve the details from saving accounts and checking accounts in a single transactional request we'll attach a resolver to the `query.getAccounts` GraphQL operation on our schema. Select

**Attach**, and on the next screen pick the same `TransactTutorial` data source created at the beginning of the tutorial. Configure the templates as follows:

### Request Mapping Template

```
#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)
}
```

### Response Mapping Template

```
#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)
```

Save the resolver and navigate to the **Queries** sections of the AWS AppSync console. In order to retrieve the saving accounts and checking accounts, execute the following query:

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
```

```
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
      username
      balance
    }
  }
}
```

We have successfully demonstrated the use of DynamoDB transactions using AWS AppSync.

## Tutorial: HTTP Resolvers

AWS AppSync enables you to use supported data sources (that is, AWS Lambda, Amazon DynamoDB, Amazon OpenSearch Service, or Amazon Aurora) to perform various operations, in addition to any arbitrary HTTP endpoints to resolve GraphQL fields. After your HTTP endpoints are available, you can connect to them using a data source. Then, you can configure a resolver in the schema to perform GraphQL operations such as queries, mutations, and subscriptions. This tutorial walks you through some common examples.

In this tutorial you use a REST API (created using Amazon API Gateway and Lambda) with an AWS AppSync GraphQL endpoint.

### One-Click Setup

If you want to automatically set up a GraphQL endpoint in AWS AppSync with an HTTP endpoint configured (using Amazon API Gateway and Lambda), you can use the following AWS CloudFormation template :



### Creating a REST API

You can use the following AWS CloudFormation template to set up a REST endpoint that works for this tutorial:



The AWS CloudFormation stack performs the following steps:

1. Sets up a Lambda function that contains your business logic for your microservice.
2. Sets up an API Gateway REST API with the following endpoint/method/content type combination:

API Resource Path	HTTP Method	Supported Content Type
/v1/users	POST	application/xml

API Resource Path	HTTP Method	Supported Content Type
/v1/users/1	GET	application/json
/v1/users/1	PUT	application/json
/v1/users/1	DELETE	application/json

## Creating Your GraphQL API

To create the GraphQL API in AWS AppSync:

- Open the AWS AppSync console and choose **Create API**.
- For the API name, type `UserData`.
- Choose **Custom schema**.
- Choose **Create**.

The AWS AppSync console creates a new GraphQL API for you using the API key authentication mode. You can use the console to set up the rest of the GraphQL API and run queries on it for the remainder of this tutorial.

## Creating a GraphQL Schema

Now that you have a GraphQL API, let's create a GraphQL schema. From the schema editor in the AWS AppSync console, make sure you schema matches the following schema:

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

type Mutation {
  addUser(userInput: UserInput!): User
  deleteUser(id: ID!): User
}

type Query {
  getUser(id: ID): 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
}
```

## Configure Your HTTP Data Source

To configure your HTTP data source, do the following:

- On the **DataSources** tab, choose **New**, and then type a friendly name for the data source (for example, HTTP).
- In **Data source type**, choose **HTTP**.
- Set the endpoint to the API Gateway endpoint that is created. Make sure that you don't include the stage name as part of the endpoint.

**Note:** At this time only public endpoints are supported by AWS AppSync.

**Note:** For more information about the certifying authorities that are recognized by the AWS AppSync service, see [Certificate Authorities \(CA\) Recognized by AWS AppSync for HTTPS Endpoints \(p. 387\)](#).

## Configuring Resolvers

In this step, you connect the http data source to the **getUser** query.

To set up the resolver:

- Choose the **Schema** tab.
- In the **Data types** pane on the right under the **Query** type, find the **getUser** field and choose **Attach**.
- In **Data source name**, choose **HTTP**.
- In **Configure the request mapping template**, paste the following code:

```
{
  "version": "2018-05-29",
  "method": "GET",
  "params": {
    "headers": {
      "Content-Type": "application/json"
    }
  },
  "resourcePath": $util.toJson("/v1/users/${ctx.args.id}")
}
```

- In **Configure the response mapping template**, paste the following code:

```
## 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
```

- Choose the **Query** tab, and then run the following query:

```
query{
  getUser(id:1){
    firstname
  }
}
```

```
        username
      }
    }
  }
}
```

This should return the following response:

```
{
  "data": {
    "getUser": {
      "id": "1",
      "username": "nadia"
    }
  }
}
```

- Choose the **Schema** tab.
- In the **Data types** pane on the right under **Mutation**, find the **addUser** field and choose **Attach**.
- In **Data source name**, choose **HTTP**.
- In **Configure the request mapping template**, paste the following code:

```
#set($xml = "<User>")
#foreach ($mapEntry in $ctx.args.userInput.entrySet())
  #set($xml = "$xml<$mapEntry.key>$mapEntry.value</$mapEntry.key>")
#end
#set($xml = "$xml</User>")
{
  "version": "2018-05-29",
  "method": "POST",
  "params": {
    "headers": {
      "Content-Type": "application/xml"
    },
    "body": "$xml"
  },
  "resourcePath": "/v1/users"
}
```

- In **Configure the response mapping template**, paste the following code:

```
## return the body
#if($ctx.result.statusCode == 200)
  ##if response is 200
  ## Because the response is of type XML, we are going to convert
  ## the result body as a map and only get the User object.
  $utils.toJson($utils.xml.toMap($ctx.result.body).User)
#else
  ##if response is not 200, append the response to error block.
  $utils.appendError($ctx.result.body, "$ctx.result.statusCode")
#end
```

- Choose the **Query** tab, and then run the following query:

```
mutation{
  addUser(userInput:{
    id:"2",
    username:"shaggy"
  })
}
```



```
  }}{
    id
    username
  }
}
```

This should return the following response:

```
{
  "data": {
    "getUser": {
      "id": "2",
      "username": "shaggy"
    }
  }
}
```

## Invoking AWS Services

You can use HTTP resolvers to set up a GraphQL API interface for AWS services. HTTP requests to AWS must be signed with the [Signature Version 4 process](#) so that AWS can identify who sent them. AWS AppSync calculates the signature on your behalf when you associate an IAM role with the HTTP data source.

You provide two additional components to invoke AWS services with HTTP resolvers:

- An IAM role with permissions to call the AWS service APIs
- Signing configuration in the data source

For example, if you want to call the [ListGraphqlApis operation](#) with HTTP resolvers, you first [create an IAM role \(p. 26\)](#) that AWS AppSync assumes with the following policy attached:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "appsync:ListGraphqlApis"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

Next, create the HTTP data source for AWS AppSync. In this example, you call AWS AppSync in the US West (Oregon) Region. Set up the following HTTP configuration in a file named `http.json`, which includes the signing region and service name:

```
{
  "endpoint": "https://appsync.us-west-2.amazonaws.com/",
  "authorizationConfig": {
    "authorizationType": "AWS_IAM",
    "awsIamConfig": {
      "signingRegion": "us-west-2",
      "signingServiceName": "appsync"
    }
  }
}
```

```
}
```

Then, use the AWS CLI to create the data source with an associated role as follows:

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

When you attach a resolver to the field in the schema, use the following request mapping template to call AWS AppSync:

```
{
  "version": "2018-05-29",
  "method": "GET",
  "resourcePath": "/v1/apis"
}
```

When you run a GraphQL query for this data source, AWS AppSync signs the request using the role you provided and includes the signature in the request. The query returns a list of AWS AppSync GraphQL APIs in your account in that AWS Region.

## Tutorial: Aurora Serverless

AWS AppSync provides a data source for executing SQL commands against Amazon Aurora Serverless clusters which have been enabled with a Data API. You can use AppSync resolvers to execute SQL statements against the Data API with GraphQL queries, mutations, and subscriptions.

### Create cluster

Before adding an RDS data source to AppSync you must first enable a Data API on an Aurora Serverless cluster and **configure a secret** using AWS Secrets Manager. You can create an Aurora Serverless cluster first with AWS CLI:

```
aws rds create-db-cluster --db-cluster-identifier http-endpoint-test --master-username
  USERNAME \
--master-user-password COMPLEX_PASSWORD --engine aurora --engine-mode serverless \
--region us-east-1
```

This will return an ARN for the cluster.

Create a Secret via the AWS Secrets Manager Console or also via the CLI with an input file such as the following using the USERNAME and COMPLEX\_PASSWORD from the previous step:

```
{
  "username": "USERNAME",
  "password": "COMPLEX_PASSWORD"
}
```

Pass this as a parameter to the AWS CLI:

```
aws secretsmanager create-secret --name HttpRDSSecret --secret-string file:///creds.json --
region us-east-1
```

This will return an ARN for the secret.

**Note the ARN** of your Aurora Serverless cluster and Secret for later use in the AppSync console when creating a data source.

## Enable Data API

You can enable the Data API on your cluster by [following the instructions in the RDS documentation](#). The Data API must be enabled before adding as an AppSync data source.

## Create database and table

Once you have enabled your Data API you can ensure it works with the `aws rds-data execute-statement` command in the AWS CLI. This will ensure that your Aurora Serverless cluster is configured correctly before adding it to your AppSync API. First create a database called `TESTDB` with the `--sql` parameter like so:

```
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"
```

If this runs without error, add a table with the `create table` command:

```
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"
```

If everything has run without issue you can move forward to adding the cluster as a data source in your AppSync API.

## GraphQL schema

Now that your Aurora Serverless Data API is up and running with a table, we will create a GraphQL schema and attach resolvers for performing mutations and subscriptions. Create a new API in the AWS AppSync console and navigate to the **Schema** page, and enter the following:

```
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
}

```

**Save** your schema and navigate to the **Data Sources** page and create a new data source. Select **Relational database** for the Data source type, and provide a friendly name. Use the database name that you created in the last step, as well as the **Cluster ARN** that you created it in. For the **Role** you can either have AppSync create a new role or create one with a policy similar to the below:

```

{
  "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:*"
      ]
    }
  ]
}

```

```
]
}
```

Note there are two **Statements** in this policy which you are granting role access. The first **Resource** is your Aurora Serverless cluster and the second is your AWS Secrets Manager ARN. You will need to provide **BOTH** ARNs in the AppSync data source configuration before clicking **Create**.

## Configuring Resolvers

Now that we have a valid GraphQL schema and an RDS data source, we can attach resolvers to the GraphQL fields on our schema. Our API will offer the following capabilities:

1. create a pet via the *Mutation.createPet* field
2. update a pet via the *Mutation.updatePet* field
3. delete a pet via the *Mutation.deletePet* field
4. get a single pet via the *Query.getPet* field
5. list all pets via the *Query.listPets* field
6. list pets in a price range via the *Query.listPetsByPriceRange* field

### Mutation.createPet

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `createPet(input: CreatePetInput!): Pet`. Choose your RDS data source. In the **request mapping template** section, add the following template:

```
#set($id=$utils.autoId())
{
  "version": "2018-05-29",
  "statements": [
    "insert into Pets VALUES ('$id', '$ctx.args.input.type', $ctx.args.input.price)",
    "select * from Pets WHERE id = '$id'"
  ]
}
```

The SQL statements will execute sequentially, based on the order in the **statements** array. The results will come back in the same order. Since this is a mutation, we run a *select* statement after the *insert* to retrieve the committed values in order to populate the GraphQL response mapping template.

In the **response mapping template** section, add the following template:

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

Because the *statements* has two SQL queries, we need to specify the second result in the matrix that comes back from the database with: `$utils.rds.toJsonString($ctx.result)[1][0]`.

### Mutation.updatePet

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `updatePet(input: UpdatePetInput!): Pet`. Choose your RDS data source. In the **request mapping template** section, add the following template:

```
{
  "version": "2018-05-29",
  "statements": [
```

```
        $util.toJson("update Pets set type='$ctx.args.input.type', price=
$ctx.args.input.price WHERE id='$ctx.args.input.id'"),
        $util.toJson("select * from Pets WHERE id = '$ctx.args.input.id'")
    ]
}
```

In the **response mapping template** section, add the following template:

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

## Mutation.deletePet

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `deletePet(input: DeletePetInput!): Pet`. Choose your RDS data source. In the **request mapping template** section, add the following template:

```
{
  "version": "2018-05-29",
  "statements": [
    $util.toJson("select * from Pets WHERE id='$ctx.args.input.id'"),
    $util.toJson("delete from Pets WHERE id='$ctx.args.input.id'")
  ]
}
```

In the **response mapping template** section, add the following template:

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

## Query.getPet

Now that the mutations are created for your schema, we will connect the three queries to showcase how to get individual items, lists, and apply SQL filtering. From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `getPet(id: ID!): Pet`. Choose your RDS data source. In the **request mapping template** section, add the following template:

```
{
  "version": "2018-05-29",
  "statements": [
    $util.toJson("select * from Pets WHERE id='$ctx.args.id'")
  ]
}
```

In the **response mapping template** section, add the following template:

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

## Query.listPets

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `getPet(id: ID!): Pet`. Choose your RDS data source. In the **request mapping template** section, add the following template:

```
{
  "version": "2018-05-29",
```

```
"statements": [
  "select * from Pets"
]
}
```

In the **response mapping template** section, add the following template:

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

## Query.listPetsByPriceRange

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `getPet(id: ID!): Pet`. Choose your RDS data source. In the **request mapping template** section, add the following template:

```
{
  "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)
  }
}
```

In the **response mapping template** section, add the following template:

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

## Run mutations

Now that you have configured all of your resolvers with SQL statements and connected your GraphQL API to your Serverless Aurora Data API, you can begin performing mutations and queries. In AWS AppSync console, choose the **Queries** tab and enter the following to create a Pet:

```
mutation add {
  createPet(input : { type:fish, price:10.0 }){
    id
    type
    price
  }
}
```

The response should contain the *id*, *type*, and *price* like so:

```
{
  "data": {
    "createPet": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "type": "fish",
      "price": "10.0"
    }
  }
}
```

You can modify this item by running the *updatePet* mutation:

```
mutation update {
  updatePet(input : {
    id:"c6fedbbe-57ad-4da3-860a-ffe8d039882a",
    type:bird,
    price:50.0
  }){
    id
    type
    price
  }
}
```

Note that we used the *id* which was returned from the *createPet* operation earlier. This will be a unique value for your record as the resolver leveraged `$util.autoId()`. You could delete a record in a similar manner:

```
mutation {
  deletePet(input : {id:ID_PLACEHOLDER}){
    id
    type
    price
  }
}
```

Create a few records with the first mutation with different values for *price* and then run some queries.

## Run Queries

Still in the **Queries** tab of the console, use the following statement to list all of the records you've created:

```
query allpets {
  listPets {
    id
    type
    price
  }
}
```

This is nice but let's leverage the SQL *WHERE* predicate that had where `price > :MIN` and `price < :MAX` in our mapping template for *Query.listPetsByPriceRange* with the following GraphQL query:

```
query {
  listPetsByPriceRange(min:1, max:11) {
    id
    type
    price
  }
}
```

You should only see records with a *price* over \$1 or less than \$10. Finally, you can perform queries to retrieve individual records as follows:

```
query {
  getPet(id:ID_PLACEHOLDER){
    id
  }
}
```



```

    type
    price
  }
}

```

## Input Sanitization

We strongly encourage the developers to sanitize the arguments of GraphQL operations. One way to do this is to provide input specific validation steps in the request mapping template before execution of a SQL statement against your Data API. Let's see how we can modify the request mapping template of the `listPetsByPriceRange` example. Instead of relying solely on the user input you can do the following:

```

#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)
  }
}

```

Another way to protect against rogue input when executing resolvers against your Data API is to use prepared statements together with stored procedure and parameterized inputs. For example, in the resolver for `listPets` define the following procedure that executes the `select` as a prepared statement:

```

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

```

This can be created in your Aurora Serverless Instance using the following execute sql command:

```

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"

```

The resulting resolver code for `listPets` is simplified since we now simply call the stored procedure. At a minimum, any string input should have single quotes [escaped \(p. 146\)](#).

```

#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("'", ""))
  }
}
```

## Escaping strings

Single quotes represent the start and end of string literals in an SQL statement, eg. 'some string value'. To allow string values with one or more single quote characters ( ' ) to be used within a string, each must be replaced with two single quotes ( ' ' ). For example, if the input string is Nadia 's dog, you would escape it for the SQL statement like

```
update Pets set type='Nadia's dog' WHERE id='1'
```

# Tutorial: Pipeline Resolvers

AWS AppSync provides a simple way to wire a GraphQL field to a single data source through unit resolvers. However, executing a single operation might not be enough. Pipeline resolvers offer the ability to serially execute operations against data sources. Create functions in your API and attach them to a pipeline resolver. Each function execution result is piped to the next until no function is left to execute. With pipeline resolvers you can now build more complex workflows directly in AWS AppSync. In this tutorial, you build a simple pictures viewing app, where users can post and view pictures posted by their friends.

## One-Click Setup

If you want to automatically set up the GraphQL endpoint in AWS AppSync with all the resolvers configured and the necessary AWS resources, you can use the following AWS CloudFormation template :



This stack creates the following resources in your account:

- IAM Role for AWS AppSync to access the resources in your account
- 2 DynamoDB tables
- 1 Amazon Cognito user pool
- 2 Amazon Cognito user pool groups
- 3 Amazon Cognito user pool users
- 1 AWS AppSync API

At the end of the AWS CloudFormation stack creation process you receive one email for each of the three Amazon Cognito users that were created. Each email contains a temporary password that you use to log

in as an Amazon Cognito user to the AWS AppSync console. Save the passwords for the remainder of the tutorial.

## Manual Setup

If you prefer to manually go through a step-by-step process through the AWS AppSync console, follow the setup process below.

### Setting Up Your Non AWS AppSync Resources

The API communicates with two DynamoDB tables: a **pictures** table that stores pictures and a **friends** table that stores relationships between users. The API is configured to use Amazon Cognito user pool as authentication type. The following AWS CloudFormation stack sets up these resources in the account.



At the end of the AWS CloudFormation stack creation process you receive one email for each of the three Amazon Cognito users that were created. Each email contains a temporary password that you use to log in as an Amazon Cognito user to the AWS AppSync console. Save the passwords for the remainder of the tutorial.

### Creating Your GraphQL API

To create the GraphQL API in AWS AppSync:

1. Open the AWS AppSync console and choose **Build From Scratch** and choose **Start**.
2. Set the name of the API to `AppSyncTutorial-PicturesViewer`.
3. Choose **Create**.

The AWS AppSync console creates a new GraphQL API for you using the API key authentication mode. You can use the console to set up the rest of the GraphQL API and run queries against it for the rest of this tutorial.

### Configuring The GraphQL API

You need to configure the AWS AppSync API with the Amazon Cognito user pool that you just created.

1. Choose the **Settings** tab.
2. Under the **Authorization Type** section, choose *Amazon Cognito User Pool*.
3. Under **User Pool Configuration**, choose **US-WEST-2** for the *AWS Region*.
4. Choose the **AppSyncTutorial-UserPool** user pool.
5. Choose **DENY** as *Default Action*.
6. Leave the **AppId client regex** field blank.
7. Choose **Save**.

The API is now set up to use Amazon Cognito user pool as its authorization type.

### Configuring Data Sources for the DynamoDB Tables

After the DynamoDB tables have been created, navigate to your AWS AppSync GraphQL API in the console and choose the **Data Sources** tab. Now, you're going to create a datasource in AWS AppSync for each of the DynamoDB tables that you just created.

1. Choose the **Data source** tab.
2. Choose **New** to create a new data source.
3. For the data source name, enter `PicturesDynamoDBTable`.
4. For data source type, choose **Amazon DynamoDB table**.
5. For region, choose **US-WEST-2**.
6. From the list of tables, choose the **AppSyncTutorial-PicturesDynamoDB** table.
7. In the **Create or use an existing role** section, choose **Existing role**.
8. Choose the role that was just created from the CloudFormation template. If you did not change the *ResourceNamePrefix*, the name of the role should be **AppSyncTutorial-DynamoDBRole**.
9. Choose **Create**.

Repeat the same process for the **friends** table, the name of the DynamoDB table should be **AppSyncTutorial-Friends** if you did not change the *ResourceNamePrefix* parameter at the time of creating the CloudFormation stack.

## Creating the GraphQL Schema

Now that the data sources are connected to your DynamoDB tables, let's create a GraphQL schema. From the schema editor in the AWS AppSync console, make sure your schema matches the following schema:

```
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!
}
```

Choose **Save Schema** to save your schema.

Some of the schema fields have been annotated with the `@aws_auth` directive. Since the API default action configuration is set to *DENY*, the API rejects all users that are not members of the groups mentioned inside the `@aws_auth` directive. For more information about how to secure your API, you can read the [Security \(p. 221\)](#) page. In this case, only admin users have access to the *Mutation.createPicture* and *Mutation.createFriendship* fields, while users that are members of either *Admins* or *Viewers* groups can access the *Query.getPicturesByOwner* field. All other users don't have access.

## Configuring Resolvers

Now that you have a valid GraphQL schema and two data sources, you can attach resolvers to the GraphQL fields on the schema. The API offers the following capabilities:

- Create a picture via the *Mutation.createPicture* field
- Create friendship via the *Mutation.createFriendship* field
- Retrieve a picture via the *Query.getPicture* field

### Mutation.createPicture

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `createPicture(input: CreatePictureInput!): Picture!`. Choose the `DynamoDBPicturesDynamoDBTable` data source. In the **request mapping template** section, add the following template:

```
#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)
}
```

In the **response mapping template** section, add the following template:

```
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

The create picture functionality is done. You are saving a picture in the **Pictures** table, using a randomly generated UUID as id of the picture, and using the Cognito username as owner of the picture.

### Mutation.createFriendship

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for `createFriendship(id: ID!, target: ID!): Boolean`. Choose the `DynamoDBFriendsDynamoDBTable` data source. In the **request mapping template** section, add the following template:

```
#set($userToFriendFriendship = { "userId" : "$ctx.args.id", "friendId":
  "$ctx.args.target" })
#set($friendToUserFriendship = { "userId" : "$ctx.args.target", "friendId":
  "$ctx.args.id" })
#set($friendsItems = [$util.dynamodb.toMapValues($userToFriendFriendship),
  $util.dynamodb.toMapValues($friendToUserFriendship)])

{
  "version" : "2018-05-29",
```

```
"operation" : "BatchPutItem",
"tables" : {
  ## Replace 'AppSyncTutorial-' default below with the ResourceNamePrefix you
  provided in the CloudFormation template
  "AppSyncTutorial-Friends": $util.toJson($friendsItems)
}
}
```

Important: In the **BatchPutItem** request template, the exact name of the DynamoDB table should be present. The default table name is *AppSyncTutorial-Friends*. If you are using the wrong table name, you get an error when AppSync tries to assume the provided role.

For the sake of simplicity in this tutorial, proceed as if the friendship request has been approved and save the relationship entry directly into the **AppSyncTutorialFriends** table.

Effectively, you're storing two items for each friendship as the relationship is bi-directional. For more details about Amazon DynamoDB best practices to represent many-to-many relationships, see [DynamoDB Best Practices](#).

In the **response mapping template** section, add the following template:

```
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type)
#end
true
```

Note: Make sure your request template contains the right table name. The default name is *AppSyncTutorial-Friends*, but your table name might differ if you changed the CloudFormation **ResourceNamePrefix** parameter.

## Query.getPicturesByOwner

Now that you have friendships and pictures, you need to provide the ability for users to view their friends' pictures. To satisfy this requirement, you need to first check that the requester is friend with the owner, and finally query for the pictures.

Because this functionality requires two data source operations, you're going to create two functions. The first function, **isFriend**, checks whether the requester and the owner are friends. The second function, **getPicturesByOwner**, retrieves the requested pictures given an owner ID. Let's look at the execution flow below for the proposed resolver on the *Query.getPicturesByOwner* field:

1. Before mapping template: Prepare the context and field input arguments.
2. **isFriend** function: Checks whether the requester is the owner of the picture. If not, it checks whether the requester and owner users are friends by doing a DynamoDB GetItem operation on the friends table.
3. **getPicturesByOwner** function: Retrieves pictures from the Pictures table using a DynamoDB Query operation on the *owner-index* Global Secondary Index.
4. After mapping template: Maps picture result so DynamoDB attributes map correctly to the expected GraphQL type fields.

Let's first create the functions.

### isFriend Function

1. Choose the **Functions** tab.
2. Choose **Create Function** to create a function.

3. For the data source name, enter `FriendsDynamoDBTable`.
4. For the function name, enter `isFriend`.
5. Inside the request mapping template text area, paste the following template:

```
#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. Inside the response mapping template text area, paste the following template:

```
#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. Choose **Create Function**.

Result: You've created the `isFriend` function.

#### [getPicturesByOwner function](#)

1. Choose the **Functions** tab.
2. Choose **Create Function** to create a function.
3. For the data source name, enter `PicturesDynamoDBTable`.
4. For the function name, enter `getPicturesByOwner`.
5. Inside the request mapping template text area, paste the following template:

```
{
    "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. Inside the response mapping template text area, paste the following template:

```
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type)
#end

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

7. Choose **Create Function**.

Result: You've created the **getPicturesByOwner** function. Now that the functions have been created, attach a pipeline resolver to the *Query.getPicturesByOwner* field.

From the schema editor in the AWS AppSync console, on the right side choose **Attach Resolver** for *Query.getPicturesByOwner(id: ID!): [Picture]*. On the following page, choose the **Convert to pipeline resolver** link that appears underneath the data source drop-down list. Use the following for the before mapping template:

```
#set($result = { "owner": $ctx.args.id, "callerId": $ctx.identity.username })
$util.toJson($result)
```

In the **after mapping template** section, use the following:

```
#foreach($picture in $ctx.result.items)
  ## prepend "src://" to picture.src property
  #set($picture['src'] = "src://${picture['src']}")
#end
$util.toJson($ctx.result.items)
```

Choose **Create Resolver**. You have successfully attached your first pipeline resolver. On the same page, add the two functions you created previously. In the functions section, choose **Add A Function** and then choose or type the name of the first function, **isFriend**. Add the second function by following the same process for the **getPicturesByOwner** function. Make sure the **isFriend** function appears first in the list followed by the **getPicturesByOwner** function. You can use the up and down arrows to rearrange to order of execution of the functions in the pipeline.

Now that the pipeline resolver is created and you've attached the functions, let's test the newly created GraphQL API.

## Testing Your GraphQL API

First, you need to populate pictures and friendships by executing a few mutations using the admin user you created. On the left side of the AWS AppSync console, choose the **Queries** tab.

### createPicture Mutation

1. In AWS AppSync console, choose the **Queries** tab.
2. Choose **Login With User Pools**.



3. On the modal, enter the Cognito Sample Client ID that was created by the CloudFormation stack for example, `37solo6mmhh7k4v63cqdfgdg5d`).
4. Enter the user name you passed as parameter to the CloudFormation stack. Default is **nadia**.
5. Use the temporary password that was sent to the email you provided as parameter to the CloudFormation stack (for example, *UserPoolUserEmail*).
6. Choose Login. You should now see the button renamed to **Logout nadia**, or whatever user name you chose when creating the CloudFormation stack (that is, *UserPoolUsername*).

Let's send a few *createPicture* mutations to populate the pictures table. Execute the following GraphQL query inside the console:

```
mutation {
  createPicture(input:{
    owner: "nadia"
    src: "nadia.jpg"
  }) {
    id
    owner
    src
  }
}
```

The response should look like below:

```
{
  "data": {
    "createPicture": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "owner": "nadia",
      "src": "nadia.jpg"
    }
  }
}
```

Let's add a few more pictures:

```
mutation {
  createPicture(input:{
    owner: "shaggy"
    src: "shaggy.jpg"
  }) {
    id
    owner
    src
  }
}
```

```
mutation {
  createPicture(input:{
    owner: "rex"
    src: "rex.jpg"
  }) {
    id
    owner
    src
  }
}
```

You've added three pictures using **nadia** as the admin user.

## createFriendship Mutation

Let's add a friendship entry. Execute the following mutations in the console.

Note: You must still be logged in as the admin user (the default admin user is **nadia**).

```
mutation {
  createFriendship(id: "nadia", target: "shaggy")
}
```

The response should look like:

```
{
  "data": {
    "createFriendship": true
  }
}
```

**nadia** and **shaggy** are friends. **rex** is not friends with anybody.

## getPicturesByOwner Query

For this step, log in as the **nadia** user using Cognito User Pools, using the credentials set up in the beginning of this tutorial. As **nadia**, retrieve the pictures owned by **shaggy**.

```
query {
  getPicturesByOwner(id: "shaggy") {
    id
    owner
    src
  }
}
```

Since **nadia** and **shaggy** are friends, the query should return the corresponding picture.

```
{
  "data": {
    "getPicturesByOwner": [
      {
        "id": "05a16fba-cc29-41ee-a8d5-4e791f4f1079",
        "owner": "shaggy",
        "src": "src://shaggy.jpg"
      }
    ]
  }
}
```

Similarly, if **nadia** attempts to retrieve her own pictures, it also succeeds. The pipeline resolver has been optimized to avoid running the **isFriend** GetItem operation in that case. Try the following query:

```
query {
  getPicturesByOwner(id: "nadia") {
    id
    owner
    src
  }
}
```

```
}
```

If you enable logging on your API (in the **Settings** pane), set the debug level to **ALL**, and run the same query again, it returns logs for the field execution. By looking at the logs, you can determine whether the **isFriend** function returned early at the **Request Mapping Template** stage:

```
{
  "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
}
```

The *earlyReturnedValue* key represents the data that was returned by the *#return* directive.

Finally, even though **rex** is a member of the **Viewers** Cognito UserPool Group, and because **rex** isn't friends with anybody, he won't be able to access any of the pictures owned by **shaggy** or **nadia**. If you log in as **rex** in the console and execute the following query:

```
query {
  getPicturesByOwner(id: "nadia") {
    id
    owner
    src
  }
}
```

You get the following unauthorized error:

```
{
  "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"
  }
]
```

You have successfully implemented complex authorization using pipeline resolvers.

## Tutorial: Delta Sync

Client applications in AWS AppSync store data by caching GraphQL responses locally to disk in a mobile/web application. Versioned data sources and Sync operations give customers the ability to perform the sync process using a single resolver. This allows clients to hydrate their local cache with results from one base query that might have a lot of records, and then receive only the data altered since their last query (the *delta updates*). By allowing clients to perform the base hydration of the cache with an initial request and incremental updates in another, you can move the computation from your client application to the backend. This is substantially more efficient for client applications that frequently switch between online and offline states.

To implement Delta Sync, the Sync query uses the Sync operation on a versioned data source. When an AWS AppSync mutation changes an item in a versioned data source, a record of that change will be stored in the *Delta* table as well. You can choose to use different *Delta* tables (e.g. one per type, one per domain area) for other versioned data sources or a single *Delta* table for your API. AWS AppSync recommends against using a single *Delta* table for multiple APIs to avoid the collision of primary keys.

In addition, Delta Sync clients can also receive a subscription as an argument, and then the client coordinates subscription reconnects and writes between offline to online transitions. Delta Sync performs this by automatically resuming subscriptions (including exponential backoff and retry with jitter through different network error scenarios), and storing events in a queue. The appropriate delta or base query is then run before merging any events from the queue, and finally processing subscriptions as normal.

Documentation for client configuration options, including the Amplify DataStore, is available on the [Amplify Framework website](#). This documentation outlines how to set up versioned DynamoDB data sources and Sync operations to work with the Delta Sync client for optimal data access.

## One-Click Setup

To automatically set up the GraphQL endpoint in AWS AppSync with all the resolvers configured and the necessary AWS resources, use this AWS CloudFormation template:



This stack creates the following resources in your account:

- 2 DynamoDB tables (Base and Delta)
- 1 AWS AppSync API with API key

- 1 IAM Role with policy for DynamoDB tables

Two tables are used to partition your sync queries into a second table that acts as a journal of missed events when the clients were offline. To keep the queries efficient on the delta table, [Amazon DynamoDB TTLs](#) are used to automatically groom the events as necessary. The TTL time is configurable for your needs on the data source (you might want this as 1hour, 1day, etc.).

## Schema

To demonstrate Delta Sync, the sample application creates a *Posts* schema backed by a *Base* and *Delta* table in DynamoDB. AWS AppSync automatically writes the mutations to both tables. The sync query pulls records from the *Base* or *Delta* table as appropriate, and a single subscription is defined to show how clients can leverage this in their reconnection logic.

```
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
}
```

The GraphQL schema is standard, but a couple things are worth calling out before moving forward. First, all of the mutations automatically first write to the *Base* table and then to the *Delta* table. The *Base* table is the central source of truth for state while the *Delta* table is your journal. If you don't pass in the `lastSync: AWSTimestamp`, the `syncPosts` query runs against the *Base* table and hydrates the cache as well as running at periodic times as a *global catchup process* for edge cases when clients are offline longer than your configured TTL time in the *Delta* table. If you do pass in the `lastSync: AWSTimestamp`, the `syncPosts` query runs against your *Delta* table and is used by clients to retrieve changed events since they were last offline. Amplify clients automatically pass the `lastSync: AWSTimestamp` value, and persist to disk appropriately.

The `_deleted` field on *Post* is used for **DELETE** operations. When clients are offline and records are removed from the *Base* table, this attribute notifies clients performing synchronization to evict items from their local cache. In cases where clients are offline for longer periods of time and the item has been removed before the client can retrieve this value with a Delta Sync query, the global catch-up event in the base query (configurable in the client) runs and removes the item from the cache. This field is marked optional because it only returns a value when running a sync query that has deleted items present.

## Mutations

For all of the mutations, AWS AppSync does a standard Create/Update/Delete operation in the *Base* table and also records the change in the *Delta* table automatically. You can reduce or extend the time to keep records by modifying the `DeltaSyncTableTTL` value on the data source. For organizations with a high velocity of data, it may make sense to keep this short. Alternatively, if your clients are offline for longer periods of time, it might be prudent to keep this longer.

## Sync Queries

The *base query* is a DynamoDB Sync operation without a `lastSync` value specified. For many organizations, this works because the base query only runs on startup and at a periodic basis thereafter.

The *delta query* is a DynamoDB Sync operation with a `lastSync` value specified. The *delta query* executes whenever the client comes back online from an offline state (as long as the base query periodic

time hasn't triggered to run). Clients automatically track the last time they successfully ran a query to sync data.

When a delta query is run, the query's resolver uses the `ds_pk` and `ds_sk` to query only for the records that have changed since the last time the client performed a sync. The client stores the appropriate GraphQL response.

For more information on executing Sync Queries, see the [Sync Operation documentation \(p. 205\)](#).

## Example

Let's start first by calling a `createPost` mutation to create an item:

```
mutation create {
  createPost(input: {author: "Nadia", title: "My First Post", content: "Hello World"}) {
    id
    author
    title
    content
    _version
    _lastChangedAt
    _deleted
  }
}
```

The return value of this mutation will look as follows:

```
{
  "data": {
    "createPost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
      "author": "Nadia",
      "title": "My First Post",
      "content": "Hello World",
      "_version": 1,
      "_lastChangedAt": 1574469356331,
      "_deleted": null
    }
  }
}
```

If you examine the contents of the *Base* table, you will see a record that looks like:

```
{
  "_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"
  }
}
```

If you examine the contents of the *Delta* table, you will see a record that looks like:

```
{
  "_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"
  }
}
```

Now we can simulate a *Base* query that a client will run to hydrate its local data store using a `syncPosts` query like:

```
query baseQuery {
  syncPosts(limit: 100, lastSync: null, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
      _lastChangedAt
    }
    startedAt
    nextToken
  }
}
```

The return value of this *Base* query will look as follows:

```
{
  "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
  }
}
```

We'll save the `startedAt` value later to simulate a *Delta* query, but first we need to make a change to our table. Let's use the `updatePost` mutation to modify our existing Post:

```
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
  }
}
```

The return value of this mutation will look as follows:

```
{
  "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
    }
  }
}
```

If you examine the contents of the *Base* table now, you should see the updated item:

```
{
  "_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"
  }
}
```

If you examine the contents of the *Delta* table now, you should see two records:

1. A record when the item was created
2. A record for when the item was updated.

The new item will look like:

```
{
  "_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"
  }
}
```

Now we can simulate a *Delta* query to retrieve modifications that occurred when a client was offline. We will use the *startedAt* value returned from our *Base* query to make the request:

```
query delta {
  syncPosts(limit: 100, lastSync: 1574469602238, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
    }
    startedAt
    nextToken
  }
}
```

The return value of this *Delta* query will look as follows:

```
{
  "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
    }
  }
}
```

# Real-time data

## GraphQL schema subscription directives

Subscriptions in AWS AppSync are invoked as a response to a mutation. This means that you can make any data source in AWS AppSync real time by specifying a GraphQL schema directive on a mutation.

The AWS Amplify client libraries automatically handle subscription connection management. The libraries use pure WebSockets as the network protocol between the client and service.

**Note:** To control authorization at connection time to a subscription, you can use AWS Identity and Access Management (IAM), AWS Lambda, Amazon Cognito identity pools, or Amazon Cognito user pools for field-level authorization. For fine-grained access controls on subscriptions, you can attach resolvers to your subscription fields and perform logic using the identity of the caller and AWS AppSync data sources. For more information, see [Authorization and Authentication \(p. 221\)](#).

Subscriptions are triggered from mutations and the mutation selection set is sent to subscribers.

The following example shows how to work with GraphQL subscriptions. It doesn't specify a data source because the data source could be Lambda, Amazon DynamoDB, or Amazon OpenSearch Service.

To get started with subscriptions, you must add a subscription entry point to your schema as follows:

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

Suppose you have a blog post site, and you want to subscribe to new blogs and changes to existing blogs. To do this, add the following Subscription definition to your schema:

```
type Subscription {
  addedPost: Post
  updatedPost: Post
  deletedPost: Post
}
```

Suppose further that you have the following mutations:

```
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!
}
```

You can make these fields real time by adding an `@aws_subscribe(mutations: ["mutation_field_1", "mutation_field_2"])` directive for each of the subscriptions you want to receive notifications for, as follows:

```
type Subscription {
  addedPost: Post
  @aws_subscribe(mutations: ["addPost"])
  updatedPost: Post
  @aws_subscribe(mutations: ["updatePost"])
  deletedPost: Post
  @aws_subscribe(mutations: ["deletePost"])
}
```

Because the `@aws_subscribe(mutations: ["", ..., ""])` takes an array of mutation inputs, you can specify multiple mutations, which initiate a subscription. If you're subscribing from a client, your GraphQL query might look like the following:

```
subscription NewPostSub {
  addedPost {
    __typename
    version
    title
    content
    author
    url
  }
}
```

This subscription query is needed for client connections and tooling.

With the pure WebSockets client, selection set filtering is done per client, as each client can define its own selection set. In this case, the subscription selection set must be a subset of the mutation selection set. For example, a subscription `addedPost{author title}` linked to the mutation `addPost(...){id author title url version}` receives only the author and title of the post. It does not receive the other fields. However, if the mutation lacked the author in its selection set, the subscriber would get a null value for the author field (or an error in case the author field is defined as required/not-null in the schema).

The subscription selection set is essential when using pure WebSockets. If a field is not explicitly defined in the subscription, then AWS AppSync doesn't return the field.

In the previous example, the subscriptions didn't have arguments. Suppose that your schema looks like the following:

```
type Subscription {
  updatedPost(id:ID! author:String): Post
  @aws_subscribe(mutations: ["updatePost"])
}
```

In this case, your client defines a subscription as follows:

```
subscription UpdatedPostSub {
  updatedPost(id:"XYZ", author:"ABC") {
    title
    content
  }
}
```

The return type of a subscription field in your schema must match the return type of the corresponding mutation field. In the previous example, this was shown as both `addPost` and `addedPost` returned as a type of `Post`.

To set up subscriptions on the client, see [Building a Client App \(p. 50\)](#).

## Using subscription arguments

An important part of using GraphQL subscriptions is understanding when and how to use arguments. You can make subtle changes to modify how and when to notify clients about mutations that have occurred. To do this, see the sample schema from [Launch a Sample Schema \(p. 3\)](#), which creates "Events" and "Comments". For this sample schema, the following mutation occurs:

```
type Mutation {
  createEvent(
    name: String!,
    when: String!,
    where: String!,
    description: String!
  ): Event
  deleteEvent(id: ID!): Event
  commentOnEvent(eventId: ID!, content: String!, createdAt: String!): Comment
}
```

In the default sample, clients can subscribe to comments when a specific eventId argument is passed through:

```
type Subscription {
  subscribeToEventComments(eventId: String!): Comment
  @aws_subscribe(mutations: ["commentOnEvent"])
}
```

However, if you want to enable clients to subscribe to a single event OR all events, you can make this argument optional by removing the exclamation point (!) from the subscription prototype:

```
subscribeToEventComments(eventId: String): Comment
```

With this change, clients that omitted this argument get comments for all events. Additionally, if you want clients to explicitly subscribe to all comments for all events, you should remove the argument as follows:

```
subscribeToEventComments: Comment
```

Use these for comments on one or more events. If you want to know about all events that are created, you could do the following:

```
type Subscription {
  subscribeToNewEvents: Event
  @aws_subscribe(mutations: ["createEvent"])
}
```

Multiple arguments can also be passed. For example, if you want to get notified of new events at a specific place and time, you could do the following:

```
type Subscription {
  subscribePlaceDate(where: String! when: String!): Event
  @aws_subscribe(mutations: ["createEvent"])
}
```

```
}
```

As a result, the client application can now do the following:

```
subscription myplaces {
  subscribePlaceDate(where: "Seattle" when: "Saturday"){
    id
    name
    description
  }
}
```

## Argument null value has meaning

When making a subscription query in AWS AppSync, a null argument value will filter the results differently than omitting the argument entirely.

Let's go back to the events app sample where we could create events and post comments on events. See the sample schema from [Launch a Sample Schema \(p. 3\)](#).

Let's modify our schema to include a new `location` field, on the `Comment` field, that describes where the comment was sent from. The value could be a set of coordinates or a place. See the following schema, stripped down for brevity:

```
type Comment {
  # The id of the comment's parent event.
  eventId: ID!
  # A unique identifier for the comment.
  commentId: String!
  # The comment's content.
  content: String
  # Location where the comment was made
  location: String
}

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

type Mutation {
  commentOnEvent(eventId: ID!, location: String, content: String): Comment
}

type Subscription {
  subscribeToEventComments(eventId: String!, location: String, content: String): Comment
    @aws_subscribe(mutations: ["commentOnEvent"])
}
```

Note the new optional field `Comment.location`.

Now, say that we want notifications for all the comments as they are posted for a particular event. In that case, we would write the following subscription:

```
subscribeToEventComments(eventId: "1") {
  eventId
```

```
    commentId
    location
    content
  }
```

If we instead add the field argument `location: null` to the preceding subscription:

```
subscribeToEventComments(eventId: "1" location: null) {
  eventId
  commentId
  location
  content
}
```

We would now be asking a different question. This subscription now registers the client to get notified of all the comments that have **not** provided a location for a particular event.

#### Note

**As of January 1, 2022, MQTT over WebSockets is no longer available as a protocol for GraphQL subscriptions in AWS AppSync APIs. Pure WebSockets is the only protocol supported in AWS AppSync.**

Clients based on the AWS AppSync SDK or the Amplify libraries, released after November 2019, automatically use pure WebSockets by default. Upgrading the clients to the latest version allows them to use AWS AppSync's pure WebSockets engine.

Pure WebSockets come with a larger payload size (240 KB), a wider variety of client options, and improved CloudWatch metrics. For more information on using pure WebSocket clients, see [Building a real-time WebSocket client \(p. 178\)](#).

## Creating generic pub/sub APIs powered by serverless WebSockets

Some applications only require simple WebSocket APIs where clients listen to a specific channel or topic. Generic JSON data with no specific shape or strongly typed requirements can be pushed to clients listening to one of these channels in a pure and simple publish-subscribe (pub/sub) pattern.

Use AWS AppSync to implement simple pub/sub WebSocket APIs with little to no GraphQL knowledge in minutes by automatically generating GraphQL code on both the API backend and the client sides.

### Create and configure pub-sub APIs

To get started, do the following:

1. Sign in to the AWS Management Console and open the [AppSync console](#).
  - In the **Dashboard**, choose **Create API**.
2. On the next screen, choose **Create a generic real-time API**.
3. Enter a friendly name for your pub/sub API.
4. Choose **Create** and then wait for a couple of minutes. A new pre-configured AWS AppSync pub/sub API will be created in your AWS account.

The API uses AWS AppSync's built-in local resolvers (for more information about using local resolvers, see [Tutorial: Local Resolvers](#) in the *AWS AppSync Developer Guide*) to manage multiple temporary pub/



sub channels and WebSocket connections, which automatically delivers and filters data to subscribed clients based only on the channel name. API calls are authorized with an API key.

After the API is deployed, you are presented with a couple of extra steps to generate client code and integrate it with your client application. For an example on how to quickly integrate a client, this guide will use a simple React web application.

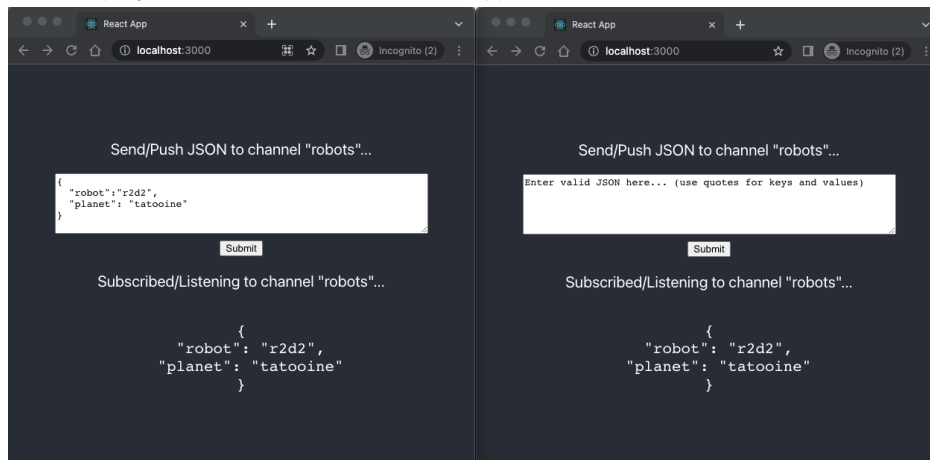
1. On your local machine, create a boilerplate React app using [NPM](#):

```
$ npm start
```

2. To test sending and receiving real-time data, open two browser windows and access `localhost:3000`. The sample application is configured to send generic JSON data to a hard-coded channel named `robots`.
3. In one of the browser windows, enter the following JSON blob in the text box then click **Submit**:

```
{
  "robot": "r2d2",
  "planet": "tatooine"
}
```

Both browser instances are subscribed to the `robots` channel and receive the published data in real time, displayed at the bottom of the web application:



All necessary GraphQL API code, including the schema, resolvers, and operations are automatically generated to enable a generic pub/sub use case. On the backend, data is published to AWS AppSync's real-time endpoint with a GraphQL mutation such as the following:

```
mutation PublishData {
  publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
    data
    name
  }
}
```

Subscribers access the published data sent to the specific temporary channel with a related GraphQL subscription:

```
subscription SubscribeToData {
  subscribe(name: "channel") {
    name
  }
}
```

```
    data
  }
}
```

## Implementing pub-sub APIs into existing applications

In case you just need to implement a real-time feature in an existing application, this generic pub/sub API configuration can be easily integrated into any application or API technology. While there are advantages in using a single API endpoint to securely access, manipulate, and combine data from one or more data sources in a single network call with GraphQL, there's no need to convert or rebuild an existing REST-based application from scratch in order to take advantage of AWS AppSync's real-time capabilities. For instance, you could have an existing CRUD workload in a separate API endpoint with clients sending and receiving messages or events from the existing application to the generic pub/sub API for real-time and pub/sub purposes only.

## Enhanced subscription filtering

In AWS AppSync, you can define and enable business logic for data filtering on the backend directly in the GraphQL API subscription resolvers by using filters that support additional logical operators. You can configure these filters, unlike the subscription arguments that are defined on the subscription query in the client. For more information about using subscription arguments, see [Using subscription arguments \(p. 166\)](#). For a list of operators, see [Resolver mapping template utility reference \(p. 286\)](#).

For the purpose of this document, we divide real-time data filtering into the following categories:

- **Basic filtering** - Filtering based on client-defined arguments in the subscription query.
- **Enhanced filtering** - Filtering based on logic defined centrally in the AWS AppSync service backend.

The following sections explain how to configure enhanced subscription filters and show their practical use.

## Defining subscriptions in your GraphQL schema

To use enhanced subscription filters, you must define the subscription in the GraphQL schema then define the enhanced filter using a filtering extension. To illustrate how enhanced subscription filtering works in AWS AppSync, use the following GraphQL schema, which defines a ticket management system API, as an example:

```
type Mutation {
  createTicket(input: TicketInput): Ticket
}

enum Priority {
  none
  lowest
  low
  medium
  high
  highest
}

type Query {
  getTicket(id: ID!): Ticket
}
```

```
type Subscription {
  onSpecialTicketCreated: Ticket
    @aws_subscribe(mutations: ["createTicket"])
  onGroupTicketCreated(group: String!): Ticket
    @aws_subscribe(mutations: ["createTicket"])
}

type Ticket {
  id: ID
  content: String
  severity: Int
  group: String
  createdAt: AWSDateTime
  priority: Priority
  status: String
  plusOnes: Int
  category: String
  type: String
  item: String
}

input TicketInput {
  content: String
  group: String
  severity: Int
  priority: Priority
  status: String
  category: String
  type: String
  item: String
}

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

### Note

Enhanced filters are enabled in the GraphQL resolver's response mapping template in a given subscription. For more information, see [Resolver mapping template overview \(p. 264\)](#).

In the preceding example, you use the Subscription to define the subscription filter requirements. In the Subscription, you define the onSpecialTicketCreated and onGroupTicketCreated subscriptions. Assume that you are implementing the enhanced filter for the first subscription, onSpecialTicketCreated. Its main requirement is to filter clients subscribed to the onSpecialTicketCreated subscription associated with the createTicket mutation.

To implement the behavior of the enhanced filter, you must use the `$extensions.setSubscriptionFilter()` method to define a filter expression evaluated against published data from a GraphQL mutation that the subscribed clients might be interested in. For more information about the filtering extensions, see [Resolver mapping template utility reference \(p. 286\)](#).

The following section explains how to use filtering extensions to implement enhanced filters.

## Creating enhanced subscription filters using filtering extensions

Enhanced filters are written in JSON in the response mapping template of the subscription's resolvers. Filters can be grouped together in a list called a `filterGroup`. Filters are defined using at least one rule, each with fields, operators, and values. In the following example, assume that you're defining the filter

for `onSpecialTicketCreated`. Using `$extensions.setSubscriptionFilter()`, multiple rules in a filter are evaluated using AND logic, while multiple filters in a filter group are evaluated using OR logic:

```
## Response Mapping Template - onSpecialTicketCreated subscription

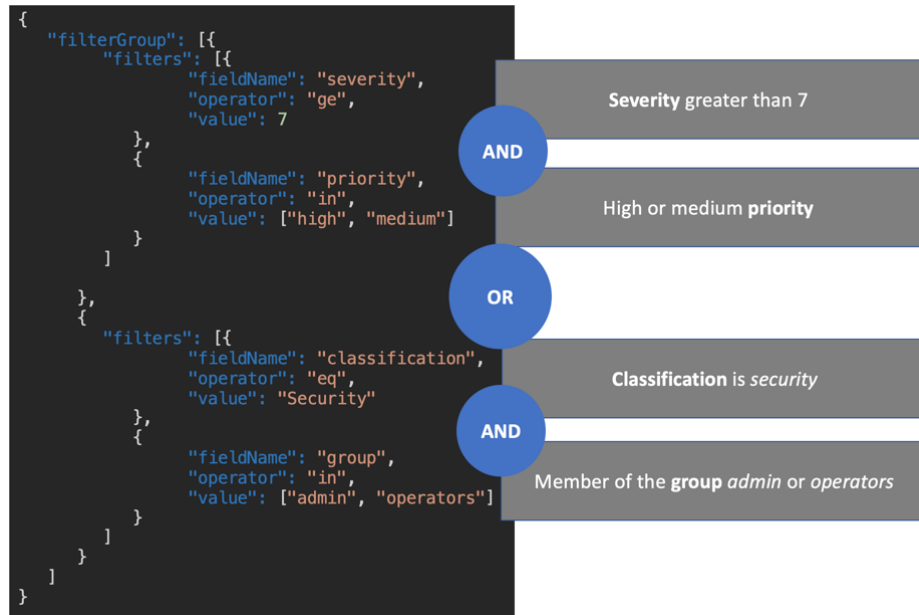
$extensions.setSubscriptionFilter({
  "filterGroup": [
    {
      "filters" : [
        {
          "fieldName" : "severity",
          "operator" : "ge",
          "value" : 7
        },
        {
          "fieldName" : "priority",
          "operator" : "in",
          "value" : ["high", "medium"]
        }
      ]
    },
    {
      "filters" : [
        {
          "fieldName" : "classification",
          "operator" : "eq",
          "value" : "Security"
        },
        {
          "fieldName" : "group",
          "operator" : "in",
          "value" : ["admin", "operators"]
        }
      ]
    }
  ]
})

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

Based on the filters defined in the preceding example, important tickets are automatically pushed to subscribed API clients if a ticket is created with:

- priority level high or medium
- AND
- severity level ge (greater than or equal to) 7
- OR
- classification ticket set to Security
- AND
- group assignment set to admin or operators

You can still manually query lower-priority tickets. However, AWS AppSync filters newly created tickets, and they aren't sent in real time to connected WebSocket clients subscribed to `onSpecialTicketCreated`.



Filters defined in the subscription resolver (enhanced filtering) take precedence over filtering based only on subscription arguments (basic filtering). For more information about using subscription arguments, see [Using subscription arguments \(p. 166\)](#).

If an argument is defined and required in the GraphQL schema of the subscription, filtering based on the given argument takes place only if the argument is defined as a rule in the resolver's `$extensions.setSubscriptionFilter()` method. However, if there are no `$extensions` filtering methods in the subscription resolver, arguments defined in the client are used only for basic filtering. You can't use basic filtering and enhanced filtering concurrently.

You can use the [\\$context variable \(p. 279\)](#) in the subscription's filter extension logic to access certain information. For example, an attribute retrieved by a `GetItem` operation from Amazon DynamoDB that's defined in the same subscription request mapping template (`$context.result.severity`), user identity (`$context.identity.sub`), or, if necessary, specific arguments from the request (`$context.args.userId`).

Now assume that you want to implement the enhanced filter behavior for `onGroupTicketCreated`, the other subscription in the example GraphQL schema. The `onGroupTicketCreated` subscription requires a mandatory group name as an argument. When created, tickets are automatically assigned a pending status, and a second filter on status is also configured:

```
## Response Mapping Template - onGroupTicketCreated subscription

$extensions.setSubscriptionFilter({
  "filterGroup": [
    {
      "filters" : [
        {
          "fieldName" : "group",
          "operator" : "eq",
          "value" : $ctx.args.group
        },
        {
          "fieldName" : "status",
          "operator" : "eq",
          "value" : "pending"
        }
      ]
    }
  ]
})
```

```
    }  
  ]  
})  
  
$util.toJson($context.result)
```

When data is published using a mutation like in the following example:

```
mutation TicketCreate{  
  createTicket(input: {description: "This is a ticket.", group: "admin", classification:  
    "Security", priority: "high", severity: 10}) {  
    id  
    classification  
    group  
    description  
    priority  
    severity  
  }  
}
```

Subscribed clients listen for the data to be automatically pushed via WebSockets as soon as a ticket is created with the `createTicket` mutation:

```
subscription onSpecialTicketCreated{  
  onSpecialTicketCreated {  
    id  
    classification  
    group  
    description  
    priority  
    severity  
  }  
}
```

Clients can be subscribed without arguments because the filtering logic is implemented in the AWS AppSync service with enhanced filtering, which simplifies the client code. Clients receive data only if the defined filter criteria is met.

## Defining enhanced filters from the client

You can use basic filtering in GraphQL with [subscriptions arguments \(p. 166\)](#). The client that makes the call in the subscription query defines the arguments' values. When enhanced filters are enabled in an AWS AppSync subscription resolver with the `$extensions` filtering, backend filters defined in the resolver take precedence and priority.

Configure dynamic, client-defined enhanced filters using a `filter` argument in the subscription. When you configure these filters, you must update the GraphQL schema to reflect the new argument:

```
...  
type Subscription {  
  onSpecialTicketCreated(filter: String!): Ticket!  
    @aws_subscribe(mutations: ["createTicket"])  
}  
...
```

The client can then send a subscription query like in the following example:

```
subscription onSpecialTicketCreated($filter: String!) {
```

```
onSpecialTicketCreated(filter: $filter) {  
  id  
  classification  
  group  
  description  
  priority  
  severity  
}  
}
```

You can configure the query variable like the following example:

```
{"filter" : "{\"severity\":{\"le\":\"2\"}}\"}
```

The `$util.transform.toSubscriptionFilter()` resolver utility can be implemented in the subscription response mapping template to apply the filter defined in the subscription argument for each client:

```
## Response Mapping Template - onSpecialTicketCreated subscription  
  
$extensions.setSubscriptionFilter($util.transform.toSubscriptionFilter($util.parseJson($ctx.args.filter)  
$util.toJson($context.result)
```

With this strategy, clients can define their own filters that use enhanced filtering logic and additional operators. Filters are assigned when a given client invokes the subscription query in a secure WebSocket connection. For more information about the transform utility for enhanced filtering, including the format of the `filter` query variable payload, see [Resolver mapping template utility reference \(p. 286\)](#).

## Unsubscribing WebSocket connections using filters

In AWS AppSync, you can forcibly unsubscribe and close (invalidate) a WebSocket connection from a connected client based on specific filtering logic. This is useful in authorization-related scenarios such as when you remove a user from a group.

Subscription invalidation occurs in response to a payload defined in a mutation. We recommend that you treat mutations used to invalidate subscription connections as administrative operations in your API and scope permissions accordingly by limiting their use to an admin user, group, or backend service. For example, using schema authorization directives such as `@aws_auth(cognito_groups: [ "Administrators" ])` or `@aws_iam`. For more information, see [Using Additional Authorization Modes \(p. 228\)](#).

Invalidation filters use the same syntax and logic as [enhanced subscription filters \(p. 170\)](#). Define these filters using the following methods:

- `$extensions.invalidateSubscriptions()` – Defined in the GraphQL resolver's response mapping templates of a mutation.
- `$extensions.setSubscriptionInvalidationFilter()` – Defined in the GraphQL resolver's response mapping templates of the subscriptions linked to the mutation.

For more information about invalidation filtering extensions, see [Resolver mapping template utility reference \(p. 286\)](#).

## Using subscription invalidation

To see how subscription invalidation works in AWS AppSync, use the following GraphQL schema:

```
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"])
}

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

Define an invalidation filter in the `removeUserFromGroup` mutation response mapping template as shown in the following:

```
## Response Mapping Template - removeUserFromGroup mutation

$extensions.invalidateSubscriptions({
  "subscriptionField": "onGroupMessageCreated",
  "payload": {
    "userId": $ctx.args.userId
    "groupId": $ctx.args.groupId
  }
})
$util.toJson($context.result)
```

When the mutation is invoked, the data defined in the `payload` object is used to unsubscribe the subscription defined in `subscriptionField`. An invalidation filter is also defined in the `onGroupMessageCreated` subscription's response mapping template.

If the `$extensions.invalidateSubscriptions()` payload contains an ID that matches the IDs from the subscribed client as defined in the filter, the corresponding subscription is unsubscribed. In addition, the WebSocket connection is closed:

```
## Response Mapping Template - onGroupMessageCreated subscription

$extensions.setSubscriptionFilter({
  "filterGroup": [
    {
      "filters" : [
        {
          "fieldName" : "groupId",
```



```

        "operator" : "eq",
        "value" : $ctx.args.groupId
      }
    ]
  }
}
})

$extensions.setSubscriptionInvalidationFilter({
  "filterGroup": [
    {
      "filters" : [
        {
          "fieldName" : "userId",
          "operator" : "eq",
          "value" : $ctx.args.userId
        },
        {
          "fieldName" : "groupId",
          "operator" : "eq",
          "value" : $ctx.args.groupId
        }
      ]
    }
  ]
})

$util.toJson($context.result)

```

Note that the subscription response template can have both subscription filters and invalidation filters defined at the same time.

For example, assume that client A subscribes a new user with the ID `user-1` to the group with the ID `group-1` using the following subscription request:

```
onGroupMessageCreated(userId : "user-1", groupId : "group-1"){...}
```

AWS AppSync runs the subscription resolver, which generates subscription and invalidation filters as defined in the preceding `onGroupMessageCreated` response mapping template. For client A, the subscription filters allow data to be sent only to `group-1`, and the invalidation filters are defined for both `user-1` and `group-1`.

Now assume that client B subscribes a user with the ID `user-2` to a group with the ID `group-2` using the following subscription request:

```
onGroupMessageCreated(userId : "user-2", groupId : "group-2"){...}
```

AWS AppSync runs the subscription resolver, which generates subscription and invalidation filters. For client B, the subscription filters allow data to be sent only to `group-2`, and the invalidation filters are defined for both `user-2` and `group-2`.

Next, assume that a new group message with the ID `message-1` is created using a mutation request, like in the following example:

```
createGroupMessage(id: "message-1", groupId :
  "group-1", message: "test message"){...}
```

Subscribed clients matching the defined filters automatically receive the following data payload via WebSockets:

```
{
```

```
"data": {
  "onGroupMessageCreated": {
    "id": "message-1",
    "groupId": "group-1",
    "message": "test message",
  }
}
```

Client A receives the message because the filtering criteria match the defined subscription filter. However, client B doesn't receive the message, as the user is not part of *group-1*. Also, the request doesn't match the subscription filter defined in the subscription resolver.

Finally, assume that *user-1* is removed from *group-1* using the following mutation request:

```
removeUserFromGroup(userId: "user-1", groupId : "group-1"){...}
```

The mutation initiates a subscription invalidation as defined in its `$extensions.invalidateSubscriptions()` resolver response mapping template method. AWS AppSync then unsubscribes client A and closes its WebSocket connection. Client B is unaffected, as the invalidation payload defined in the mutation doesn't match its user or group.

When AWS AppSync invalidates a connection, the client receives a message confirming that they are unsubscribed:

```
{
  "message": "Subscription complete."
}
```

## Using context variables in subscription invalidation filters

As with enhanced subscription filters, you can use the [\\$context variable \(p. 279\)](#) in the subscription invalidation filter extension to access certain data. For example, an attribute retrieved by a `GetItem` operation from Amazon DynamoDB that's defined in the same subscription request mapping template (`$context.result.severity`), user identity (`$context.identity.claims.group`) or, if necessary, specific arguments from the request (`$context.args.userId`).

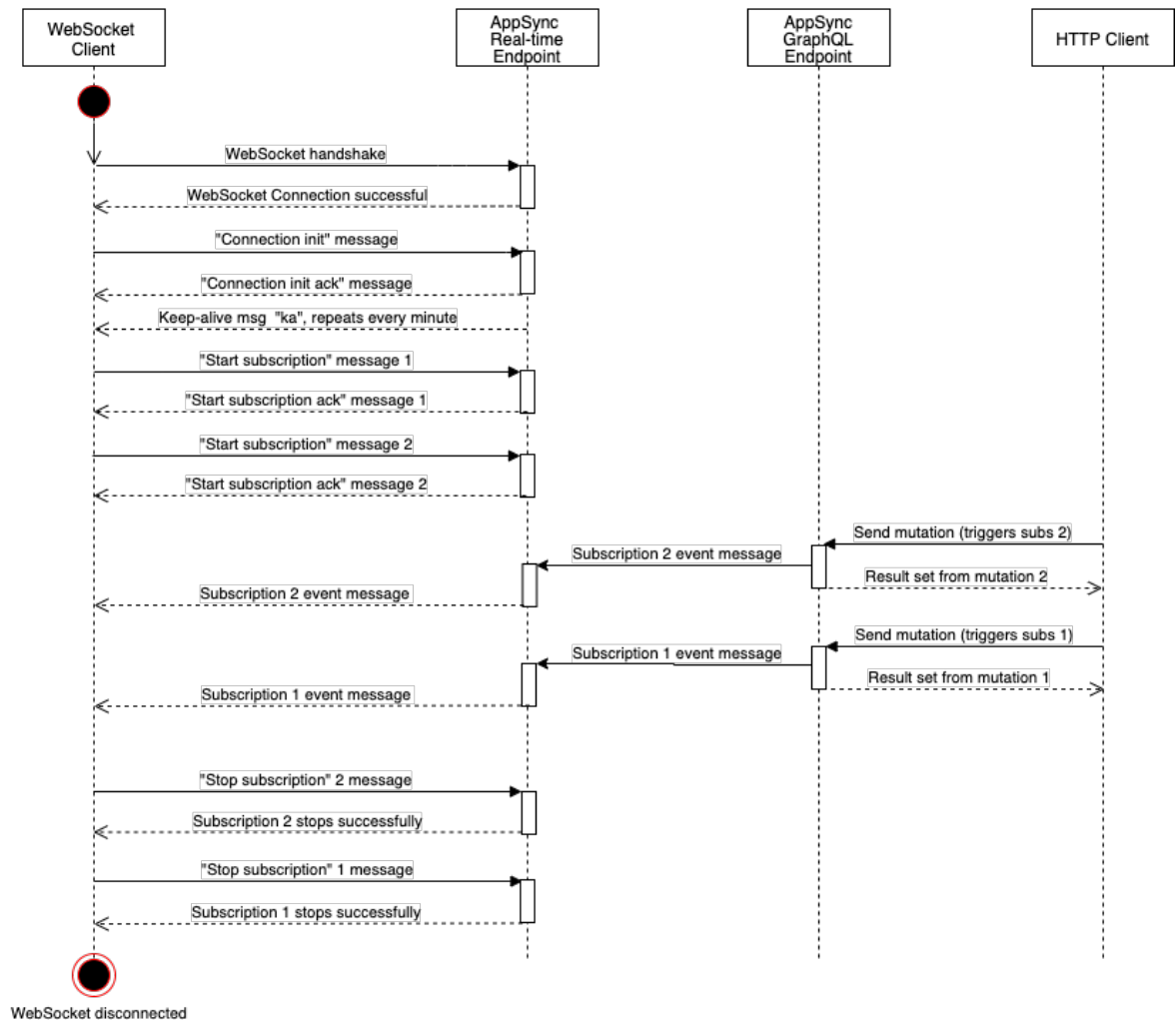
It's possible to configure an email address as the invalidation payload in the mutation, then match it against the email attribute or claim from a subscribed user authorized with Amazon Cognito user pools or OpenID Connect. The invalidation filter defined in the `$extensions.setSubscriptionInvalidationFilter()` subscription invalidator checks if the email address set by the mutation's `$extensions.invalidateSubscriptions()` payload matches the email address retrieved from the user's JWT token in `$context.identity.claims.email`, initiating the invalidation.

## Building a real-time WebSocket client

### Real-time WebSocket client implementation for GraphQL subscriptions

The following sequence diagram and steps show the real-time subscriptions workflow between the WebSocket client, HTTP client, and AWS AppSync.

AWS AppSync Developer Guide  
Real-time WebSocket client  
implementation for GraphQL subscriptions



1. The client establishes a WebSocket connection with the AWS AppSync real-time endpoint. If there is a network error, the client should do a jittered exponential backoff. For more information, see [Exponential Backoff And Jitter](#) on the AWS Architecture Blog.
2. After successfully establishing the WebSocket connection, the client sends a `connection_init` message.
3. The client waits for a `connection_ack` message from AWS AppSync. This message includes a `connectionTimeoutMs` parameter, which is the maximum wait time in milliseconds for a "ka" (keep-alive) message.
4. AWS AppSync sends "ka" messages periodically. The client keeps track of the time that it received each "ka" message. If the client doesn't receive a "ka" message within `connectionTimeoutMs` milliseconds, the client should close the connection.
5. The client registers the subscription by sending a `start subscription` message. A single WebSocket connection supports multiple subscriptions, even if they are in different authorization modes.
6. The client waits for AWS AppSync to send `start_ack` messages to confirm successful subscriptions. If there is an error, AWS AppSync returns a `"type": "error"` message.
7. The client listens for subscription events, which are sent after a corresponding mutation is called. Queries and mutations are usually sent through `https://` to the AWS AppSync GraphQL endpoint. Subscriptions flow through the AWS AppSync real-time endpoint using the secure WebSocket (`wss://`).

8. The client unregisters the subscription by sending a `stop` subscription message.
9. After unregistering all subscriptions and checking that there are no messages transferring through the WebSocket, the client can disconnect from the WebSocket connection.

## Handshake details to establish the WebSocket connection

To connect and initiate a successful handshake with AWS AppSync, a WebSocket client needs the following:

- The AWS AppSync real-time endpoint
- A query string that contains header and payload parameters:
  - `header`: Contains information relevant to the AWS AppSync endpoint and authorization. This is a base64-encoded string from a stringified JSON object. The JSON object content varies depending on the authorization mode.
  - `payload`: Base64-encoded string of payload.

With these requirements, a WebSocket client can connect to the URL, which contains the real-time endpoint with the query string, using `graphql-ws` as the WebSocket protocol.

## Discovering the real-time endpoint from the GraphQL endpoint

The AWS AppSync GraphQL endpoint and the AWS AppSync real-time endpoint are slightly different in protocol and domain. You can retrieve the GraphQL endpoint using the AWS Command Line Interface (AWS CLI) command `aws appsync get-graphql-api`.

### AWS AppSync GraphQL endpoint:

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

### AWS AppSync real-time endpoint:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

Applications can connect to the AWS AppSync GraphQL endpoint (`https://`) using any HTTP client for queries and mutations. Applications can connect to the AWS AppSync real-time endpoint (`wss://`) using any WebSocket client for subscriptions.

With custom domain names, you can interact with both endpoints using a single domain. For example, if you configure `api.example.com` as your custom domain, you can interact with your GraphQL and real-time endpoints using these URLs:

### AWS AppSync custom domain GraphQL endpoint:

```
https://api.example.com/graphql
```

### AWS AppSync custom domain real-time endpoint:

```
wss://api.example.com/graphql/realtime
```

## Header parameter format based on AWS AppSync API authorization mode

The format of the header object used in the connection query string varies depending on the AWS AppSync API authorization mode. The `host` field in the object refers to the AWS AppSync GraphQL endpoint, which is used to validate the connection even if the `wss://` call is made against the real-time endpoint. To initiate the handshake and establish the authorized connection, the payload should be an empty JSON object.

### API key

Header contents:

- `"host": <string>`: The host for the AWS AppSync GraphQL endpoint or your custom domain name.
- `"x-api-key": <string>`: The API key configured for the AWS AppSync API.

Example:

```
{
  "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
  "x-api-key": "da2-12345678901234567890123456"
}
```

Payload content:

```
{}
```

Request URL:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJ0b3N0IjoizXhhbXBsZTEyMzQ1Njc4OTAwMDAuYXBwc3luYy1hcGkudXMtZWZzdC0xLmFtYXpvc3cy5jb20iLCJ4LWFT
```

## Amazon Cognito user pools and OpenID Connect (OIDC)

Header contents:

- `"Authorization": <string>`: A JWT access token.
- `"host": <string>`: The host for the AWS AppSync GraphQL endpoint or your custom domain name.

Example:

```
{
  "Authorization": "eyJ0b3N0IjoizXhhbXBsZTEyMzQ1Njc4OTAwMDAuYXBwc3luYy1hcGkudXMtZWZzdC0xLmFtYXpvc3cy5jb20iLCJ4LWFT",
  "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

Payload content:

```
{}
```

Request URL:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?  
header=eyJBdXRob3JpemF0aW9uIjoizXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVXTkhoc1Fq
```

## IAM

Header contents include the following:

- "accept": "application/json, text/javascript": A constant <string> parameter.
- "content-encoding": "amz-1.0": A constant <string> parameter.
- "content-type": "application/json; charset=UTF-8": A constant <string> parameter.
- "host": <string>: This is the host for the AWS AppSync GraphQL endpoint. - "x-amz-date": <string>: The timestamp must be in UTC and in the following ISO 8601 format: YYYYMMDD'THHMMSS'Z'. For example, 20150830T123600Z is a valid timestamp. Do not include milliseconds in the timestamp. For more information, see [Handling dates in Signature Version 4](#) in the *AWS General Reference*.
- "X-Amz-Security-Token": <string>: The AWS session token, which is required when using temporary security credentials. For more information, see [Using temporary credentials with AWS resources](#) in the *IAM User Guide*.
- "Authorization": <string>: Signature Version 4 (SigV4) signing information for the AWS AppSync endpoint. For more information on the signing process, see [Task 4: Add the signature to the HTTP request](#) in the *AWS General Reference*.

The SigV4 signing HTTP request includes a canonical URL, which is the AWS AppSync GraphQL endpoint with /connect appended. The service endpoint AWS Region is same Region where you're using the AWS AppSync API, and the service name is 'appsync'. The HTTP request to sign is the following:

```
{  
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql/connect",  
  data: "{}",  
  method: "POST",  
  headers: {  
    "accept": "application/json, text/javascript",  
    "content-encoding": "amz-1.0",  
    "content-type": "application/json; charset=UTF-8",  
  }  
}
```

Example:

```
{  
  "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":  
  "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXR0ZWFEXAMPLECwRQIgAh97Cljq7wOPL8KsxP3YtDuyC/9hAj8PhJ7Fvf38SgoCIQDhJEXA  
  +
```

AWS AppSync Developer Guide  
Header parameter format based on  
AWS AppSync API authorization mode

```
+pEagWCveZUjKE0zyUhBEXAMPLEjj/////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8w
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUDAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtW+9zF7NaMMSe07wN2gG2tH0eKMEEXAMPLEQX+sMbyTQo8iepP9PZOz1ZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnG1SLNVx7SMcfovkFDqQamm
+88y1OwWAEYK7qcocX6Z7GGcaYuIfGpaX2MCCeLeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj+BwDg3ht2CrNvpwjVYl1j9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNCfKNCg3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYefwzexjKrV4mWiURg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTLokgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMckcCGFnwBNFLmfnbpNqT6rUBxxs3X5ntX9dHVTs
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUGCH2TfQBJ+MLMVVpggqJsPKt582caFKARiFivO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+tOLnh1YPfSLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOkTgeYQsSJdhSk6XxXThrWL9EnwBCXDkICMqndtAxyyM9
+WgtPtKOOWeDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA=="
"Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
Signature=83EXAMPLEbccc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}
```

Payload content:

```
{}
```

Request URL:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJEXAMPLEHgiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoEXAMPLEEv
```

To sign the request using a custom domain:

```
{
  url: "https://api.example.com/graphql/connect",
  data: "{}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

Example:

```
{
  "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":
    "AgEXAMPLEZ2luX2VjEAOaDmFwLXNvdXR0ZWZEXAMPLEcwrQIGah97Cljq7wOPL8KsxP3YtDuyC/9hAj8PhJ7Fvf38SgoCIQDhJEXA
+
+pEagWCveZUjKE0zyUhBEXAMPLEjj/////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8w
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUDAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtW+9zF7NaMMSe07wN2gG2tH0eKMEEXAMPLEQX+sMbyTQo8iepP9PZOz1ZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnG1SLNVx7SMcfovkFDqQamm
+88y1OwWAEYK7qcocX6Z7GGcaYuIfGpaX2MCCeLeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj+BwDg3ht2CrNvpwjVYl1j9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
```

```
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/  
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNCfKNCg3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa  
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/  
tT13yrmPd5QYefwzexjKrV4mWiuRg8NTHYSZJUaeyCwTom80VFUJXG  
+GYTUyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCkcCGFnwBNFLmfnbpNqT6rUBxxs3X5ntX9d8HVtS  
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUGCH2TfqBj+MLMVVvpqgJsPKt582caFKArIFivO  
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h  
+tOLnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOKtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqdnAxyyM9  
+WgtPtK0OweDlCaRs3R2qXcbNgVhleMk4IWNf8D1695AenU1LwHjOJLkCjxgNFiwAFEPH9aEXAMPLExA==",  
  "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXX/20200401/  
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-  
encoding;content-type;host;x-amz-date;x-amz-security-token,  
  Signature=83EXAMPLEebcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"  
}
```

Payload content:

```
{}
```

Request URL:

```
wss://api.example.com/graphql?  
header=eyJEXAMPLEHQiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoEXAMPLEE
```

**Note:** One WebSocket connection can have multiple subscriptions (even with different authentication modes). One way to implement this is to create a WebSocket connection for the first subscription and then close it when the last subscription is unregistered. You can optimize this by waiting a few seconds before closing the WebSocket connection, in case the app is subscribed immediately after the last subscription is unregistered. For a mobile app example, when changing from one screen to another, on *unmounting* event it stops a subscription, and on *mounting* event it starts a different subscription.

## Lambda authorization

Header contents:

- "Authorization": <string>: The value that is passed as authorizationToken.
- "host": <string>: The host for the AWS AppSync GraphQL endpoint or your custom domain name.

Example:

```
{  
  
  "Authorization": "M0UzQzM1MkQtMkI0Ni00OTZCLUI1NkQtMUM0MTQ0QjVBRTczCkI1REEzRTIxLTk5NzItNDJENi1BQjMwLTFCN  
  "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"  
}
```

Payload content:

```
{}
```

Request URL:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?  
header=eyJBdXRob3JpemF0aW9uIjoizXlkcmFXUWlPaUpqYkc1eGIzQTVlVzVNSZA5UVVlYSxJNVEpIV0VGTFNYQmllVTVXTkhoc1Fq
```



## Real-time WebSocket operation

After initiating a successful WebSocket handshake with AWS AppSync, the client must send a subsequent message to connect to AWS AppSync for different operations. These messages require the following data:

- `type`: The type of the operation.
- `id`: A unique identifier for the subscription. We recommend using a UUID for this purpose.
- `payload`: The associated payload, depending on the operation type.

The `type` field is the only required field; the `id` and `payload` fields are optional.

### Sequence of events

To successfully initiate, establish, register, and process the subscription request, the client must step through the following sequence:

1. Initialize connection (`connection_init`)
2. Connection acknowledgment (`connection_ack`)
3. Subscription registration (`start`)
4. Subscription acknowledgment (`start_ack`)
5. Processing subscription (`data`)
6. Subscription unregistration (`stop`)

### Connection init message

After a successful handshake, the client must send the `connection_init` message to start communicating with the AWS AppSync real-time endpoint. Without this step, all other messages are ignored. The message is a string obtained by stringifying the following JSON object as follows:

```
{ "type": "connection_init" }
```

### Connection acknowledge message

After sending the `connection_init` message, the client must wait for the `connection_ack` message. All messages sent before receiving `connection_ack` are ignored. The message should read as follows:

```
{
  "type": "connection_ack",
  "payload": {
    // Time in milliseconds waiting for ka message before the client should terminate the
    // WebSocket connection
    "connectionTimeoutMs": 300000
  }
}
```

### Keep-alive message

In addition to the connection acknowledgment message, the client periodically receives keep-alive messages. If the client doesn't receive a keep-alive message within the connection timeout period, the client should close the connection. AWS AppSync keeps sending these messages and servicing the

registered subscriptions until it shuts down the connection automatically (after 24 hours). Keep-alive messages are heartbeats and do not need the client to acknowledge them.

```
{ "type": "ka" }
```

## Subscription registration message

After the client receives a `connection_ack` message, the client can send subscription registration messages to AWS AppSync. This type of message is a stringified JSON object that contains the following fields:

- `"id"`: `<string>`: The ID of the subscription. This ID must be unique for each subscription, otherwise the server returns an error indicating that the subscription ID is duplicated.
- `"type"`: `"start"`: A constant `<string>` parameter.
- `"payload"`: `<Object>`: An object that contains the information relevant to the subscription.
  - `"data"`: `<string>`: A stringified JSON object that contains a GraphQL query and variables.
    - `"query"`: `<string>`: A GraphQL operation.
    - `"variables"`: `<Object>`: An object that contains the variables for the query.
  - `"extensions"`: `<Object>`: An object that contains an authorization object.
- `"authorization"`: `<Object>`: An object that contains the fields required for authorization.

## Authorization object for subscription registration

The same rules in the [Header parameter format based on AWS AppSync API authorization mode \(p. 181\)](#) section apply for the authorization object. The only exception is for IAM, where the SigV4 signature information is slightly different. For more details, see the IAM example.

Example using Amazon Cognito user pools:

```
{
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\n onCreateMessage {\n __typename\n message\n }\n }\",\"variables\":{\"}}",
    "extensions": {
      "authorization": {
        "Authorization":
        "eyJEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJEXAMPLEBieU5WNHhsQjhPVW9YMnM2WldvPSIsImFsZyI6IEXAMPLEEn0.eyJzdWIiOiJEXAMPLEqTCtrYeboUJ4luRSTPXAnewNeEXAMPLE14C6sfq05t00fOMpiUwj9k19gtNCCMqoSsjtQoUweFnH4JYa5EXAMPLEVxOyQE04G7jQrt5RWvW7yQU3sNQRLEXAMPLEcd0yufBiCYs3dfQxTTdvR1B6Wz6CD78lfNeKqfzzUn2beMoup2h6EXAMPLE4ow8cUPUPvGODzRtHNMBwSk",
        "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
      }
    }
  },
  "type": "start"
}
```

Example using IAM:

```
{
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\n onCreateMessage {\n __typename\n message\n }\n }\",\"variables\":{\"}}",
    "extensions": {
```

```
    "authorization": {
      "accept": "application/json, text/javascript",
      "content-type": "application/json; charset=UTF-8",
      "X-Amz-Security-Token":
        "AgEXAMPLEZ2luX2VjEAAoDmFWLXNvdXR0ZWFEXAMPLEcwRQIGAh97Cljq7wOPL8KsxP3YtDuyC/9hAaj8PhJ7Fvf38SgoCIQDhJEXA
+
+pEagWCveZUjKE0zyUhBEXAMPLEjJ/////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8w
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9Psw0NoFy3vLBUDAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtWR+9zF7NaMMmSe07wN2gG2tH0eKMEEXAMPLEQX+sMbytQo8iepP9PZOzLZsSfb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWnNGLSNVx7SMcfovkFDQqamm
+88y1OwwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNw64S9Nvj+BwDg3ht2CrNvpwjVYlJ9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8Ays7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNCfKNCg3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEFwzexjKrV4mWiuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCKcCGFnwBNFLmfmbpNqT6rUBxxs3X5ntX9d8HvtS
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUGCH2TfQbj+MLMVVvpqgJsPKt582caFKarIFivO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+tOLnh1YPFsLQ88anib/7TTC8k9DSBTq0ASE8R2GbSEsm09qbbMwgEaYUhOKtGeyQsSjdhSk6XxxThrWL9EnwBCXDkICMqndtAxyyM9
+WgtPtKOoweDlCaRs3R2qXcbNgVhleMk4IwnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA==",
      "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
      Signature=b90131a61a7c4318e1c35ead5dbfdeb46339a7585bbdbecaeaff51f4022eb1fd",
      "content-encoding": "amz-1.0",
      "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
      "x-amz-date": "20200401T001010Z"
    }
  },
  "type": "start"
}
```

Example using a custom domain name:

```
{
  "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"
}
```

The SigV4 signature does not need `/connect` to be appended to the URL, and the JSON stringified GraphQL operation replaces `data`. The following is an example of a SigV4 signature request:

```
{
  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",
```

```
}  
}
```

## Subscription acknowledgment message

After sending the subscription start message, the client should wait for AWS AppSync to send the `start_ack` message. The `start_ack` message indicates that the subscription is successful.

Subscription acknowledgment example:

```
{  
  "type": "start_ack",  
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69"  
}
```

## Error message

If connection init or subscription registration fails, or if a subscription is ended from the server, the server sends an error message to the client:

- `"type": "error"`: A constant `<string>` parameter.
- `"id": <string>`: The ID of the corresponding registered subscription, if relevant.
- `"payload" <Object>`: An object that contains the corresponding error information.

Example:

```
{  
  "type": "error",  
  "payload": {  
    "errors": [  
      {  
        "errorType": "LimitExceededError",  
        "message": "Rate limit exceeded"  
      }  
    ]  
  }  
}
```

## Processing data messages

When a client submits a mutation, AWS AppSync identifies all of the subscribers interested in it and sends a `"type": "data"` message to each using the corresponding subscription `id` from the `"start"` subscription operation. The client is expected to keep track of the subscription `id` that it sends so that when it receives a data message, the client can match it with the corresponding subscription.

- `"type": "data"`: A constant `<string>` parameter.
- `"id": <string>`: The ID of the corresponding registered subscription.
- `"payload" <Object>`: An object that contains the subscription information.

Example:

```
{  
  "type": "data",
```

```
{
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": {
      "onCreateMessage": {
        "__typename": "Message",
        "message": "test"
      }
    }
  }
}
```

## Subscription unregistration message

When the app wants to stop listening to the subscription events, the client should send a message with the following stringified JSON object:

- "type": "stop": A constant <string> parameter.
- "id": <string>: The ID of the subscription to unregister.

Example:

```
{
  "type": "stop",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

AWS AppSync sends back a confirmation message with the following stringified JSON object:

- "type": "complete": A constant <string> parameter.
- "id": <string>: The ID of the unregistered subscription.

After the client receives the confirmation message, it receives no more messages for this particular subscription.

Example:

```
{
  "type": "complete",
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

## Disconnecting the WebSocket

Before disconnecting, to avoid data loss, the client should have the necessary logic to check that no operation is currently in place through the WebSocket connection. All subscriptions should be unregistered before disconnecting from the WebSocket.

# Configuration and settings

AWS AppSync enables you to:

- Cache data that's requested often but unlikely to change from request to request. This can reduce the load on your resolvers. For more information, see [the section called "Caching" \(p. 190\)](#).
- Version GraphQL objects to handle and avoid conflict among multiple clients. For more information, see [the section called "Conflict Detection and Sync" \(p. 196\)](#).

Additionally, AWS AppSync includes the following standard AWS tools for logging, monitoring, and tracing:

- [Logging in AWS CloudTrail \(p. 217\)](#)
- [Monitoring with Amazon CloudWatch \(p. 205\)](#)
- [Tracing with AWS X-Ray \(p. 215\)](#)

## Caching

AWS AppSync's server-side data caching capabilities make data available in a high speed, in-memory cache, improving performance and decreasing latency. This reduces the need to directly access data sources. Caching is available for both unit and pipeline resolvers.

Refer to this section for help defining the desired behavior of server-side caching in your AWS AppSync API.

## Instance Types

AWS AppSync hosts Amazon ElastiCache for Redis instances in the same AWS account and AWS Region as your AWS AppSync API.

The following ElastiCache for Redis instance types are available:

### **small**

1 vCPU, 1.5 GiB RAM, low to moderate network performance

### **medium**

2 vCPU, 3 GiB RAM, low to moderate network performance

### **large**

2 vCPU, 12.3 GiB RAM, up to 10 Gigabit network performance

### **xlarge**

4 vCPU, 25.05 GiB RAM, up to 10 Gigabit network performance

### **2xlarge**

8 vCPU, 50.47 GiB RAM, up to 10 Gigabit network performance

### **4xlarge**

16 vCPU, 101.38 GiB RAM, up to 10 Gigabit network performance

### 8xlarge

32 vCPU, 203.26 GiB RAM, 10 Gigabit network performance (not available in all Regions)

### 12xlarge

48 vCPU, 317.77 GiB RAM, 10 Gigabit network performance

#### Note

Historically, you specified a specific instance type (such as `t2.medium`). As of July 2020, these legacy instance types continue to be available, but their use is deprecated and discouraged. We recommend that you use the generic instance types described here.

## Caching behavior

The following are the behaviors related to caching:

### None

No server-side caching.

### Full request caching

If the data is not in the cache, it is retrieved from the data source and populates the cache until the time to live (TTL) expiration. All subsequent requests to your API are returned from the cache. This means that data sources aren't contacted directly unless the TTL expires. In this setting, we use the contents of the `$context.arguments` and `$context.identity` maps as caching keys.

### Per-resolver caching

With this setting, each resolver must be explicitly opted in for it to cache responses. You can specify a TTL and caching keys on the resolver. Caching keys that you can specify are the top-level maps `$context.arguments`, `$context.source`, and `$context.identity`, and/or string fields from these maps. The TTL value is mandatory, but the caching keys are optional. If you don't specify any caching keys, the defaults are the contents of the `$context.arguments`, `$context.source`, and `$context.identity` maps.

For example, you could use the following combinations:

- `$context.arguments` and `$context.source`
- `$context.arguments` and `$context.identity.sub`
- `$context.arguments.id` or `$context.arguments.InputType.id`
- `$context.source.id` and `$context.identity.sub`
- `$context.identity.claims.username`

When you specify only a TTL and no caching keys, the behavior of the resolver is the same as full request caching.

### Cache time to live

This setting defines the amount of time to store cached entries in memory. The maximum TTL is 3,600 seconds (1 hour), after which entries are automatically deleted.

## Cache encryption

Cache encryption comes in the following two flavors. These are similar to the settings that ElastiCache for Redis allows. You can enable the encryption settings only when first enabling caching for your AWS AppSync API.

- Encryption in transit – Requests between AWS AppSync, the cache, and data sources (except insecure HTTP data sources) are encrypted at the network level. Because there is some processing needed to encrypt and decrypt the data at the endpoints, in-transit encryption can impact performance.
- Encryption at rest – Data saved to disk from memory during swap operations are encrypted at the cache instance. This setting also impacts performance.

To invalidate cache entries, you can make a flush cache API call using either the AWS AppSync console or the AWS Command Line Interface (AWS CLI).

For more information, see the [ApiCache](#) data type in the AWS AppSync API Reference.

## Cache eviction

When you set up AWS AppSync's server-side caching, you can configure a maximum TTL. This value defines the amount of time that cached entries are stored in memory. In situations where you must remove specific entries from your cache, you can use AWS AppSync's `evictFromApiCache` extensions utility in your resolver's request or response mapping template. (For example, when your data in your data sources have changed, and your cache entry is now stale.) To evict an item from the cache, you must know its key. For this reason, if you must evict items dynamically, we recommend using per-resolver caching and explicitly defining a key to use to add entries to your cache.

## Evicting a cache entry

To evict an item from the cache, use the `evictFromApiCache` extensions utility. Specify the type name and field name, then provide an object of key-value items to build the key of the entry that you want to evict. In the object, each key represents a valid entry from the `$context` object that is used in the cached resolver's `cachingKey` list. Each value is the actual value used to construct the value of the key. You must put the items in the object in the same order as the caching keys in the cached resolver's `cachingKey` list.

For example, see the following schema:

```
type Note{
  id: ID!
  title: String
  content: String!
}

type Query {
  getNote(id: ID!): Note
}

type Mutation {
  updateNote(id: ID!, content: String!): Note
}
```

In this example, you can enable per-resolver caching, then enable it for the `getNote` query. Then, you can configure the caching key to consist of `[$context.arguments.id]`.

When you try to get a `Note`, to build the cache key, AWS AppSync performs a lookup in its server-side cache using the `id` argument of the `getNote` query.

When you update a `Note`, you must evict the entry for the specific note to make sure that the next request fetches it from the backend data source. To do this, you must create a request mapping template.

The following example shows one way to handle the eviction using this method:



```
#set($cachingKeys = {})  
$util.qr($cachingKeys.put("context.arguments.id", $context.arguments.id))  
$extensions.evictFromApiCache("Query", "getNote", $cachingKeys)  
  
{  
  "version" : "2017-02-28",  
  "operation" : "UpdateItem",  
  "key" : { "id" : "$context.arguments.id" },  
  "update" : {  
    "expression" : "SET #content = :content",  
    "expressionNames": { "#content" : "content" }  
    "expressionValues": {  
      ":content" : $util.dynamodb.toDynamoDBJson($context.arguments.content)  
    }  
  }  
}
```

Alternatively, you can also handle the eviction in the response mapping template:

```
#set($cachingKeys = {})  
$util.qr($cachingKeys.put("context.arguments.id", $context.arguments.id))  
$extensions.evictFromApiCache("Query", "getNote", $cachingKeys)  
  
$util.toJson($context.result)
```

When the `updateNote` mutation is processed, AWS AppSync tries to evict the entry. If an entry is successfully cleared, the response contains an `apiCacheEntriesDeleted` value in the `extensions` object that shows how many entries were deleted:

```
"extensions": {  "apiCacheEntriesDeleted": 1}
```

## Evicting a cache entry based on identity

You can create caching keys based on multiple values from the `$context` object.

For example, take the following schema that uses Amazon Cognito user pools as the default auth mode and is backed by an Amazon DynamoDB data source:

```
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  
}
```

The `Note` object types are saved in a DynamoDB table. The table has a composite key that uses the Amazon Cognito user name as the primary key and the `id` (a slug) of the `Note` as the partition key. This is a multi-tenant system that allows multiple users to host and update their private `Note` objects, which are never shared.

Since this is a read-heavy system, the `getNote` query is cached using per-resolver caching, with the caching key composed of `[$context.identity.username, $context.arguments.id]`. When a

Note is updated, you can evict the entry for that specific Note. You must add the components in the object in the same order that they are specified in your resolver's `cachingKeys` list.

The following example shows this:

```
#set($cachingKeys = {})  
$util.qr($cachingKeys.put("context.identity.username", $context.identity.username))  
$util.qr($cachingKeys.put("context.arguments.id", $context.arguments.id))  
$extensions.evictFromApiCache("Query", "getNote", $cachingKeys)  
  
{  
  "version" : "2017-02-28",  
  "operation" : "UpdateItem",  
  "key" : {  
    "username": "$context.identity.username",  
    "slug" : "$context.arguments.id"  
  },  
  "update" : {  
    "expression" : "SET #content = :content",  
    "expressionNames": { "#content" : "content" }  
    "expressionValues": {  
      ":content" : $util.dynamodb.toDynamoDBJson($context.arguments.content)  
    }  
  }  
}
```

A backend system can also update the Note and evict the entry. For example, take this mutation:

```
type Mutation {  
  updateNoteFromBackend(id: ID!, content: String!, username: ID!): Note @aws_iam  
}
```

You can evict the entry, but add the components of the caching key to the `cachingKeys` object.

In the following example, the eviction occurs in the response mapping template of the resolver:

```
#set($cachingKeys = {})  
$util.qr($cachingKeys.put("context.identity.username", $context.arguments.username))  
$util.qr($cachingKeys.put("context.arguments.id", $context.arguments.id))  
$extensions.evictFromApiCache("Query", "getNote", $cachingKeys)  
  
$utils.toJson($context.result)
```

In cases where your backend data has been updated outside of AWS AppSync, you can evict an item from the cache by calling a mutation that uses a `NONE` data source.

## Configuring custom domain names

With AWS AppSync, you can use custom domain names to configure a single, memorable domain that works for both your GraphQL and real-time APIs.

In other words, you can utilize simple and memorable endpoint URLs with domain names of your choice by creating custom domain names that you associate with the AWS AppSync APIs in your account.

When you configure an AWS AppSync API, two endpoints are provisioned:

### AWS AppSync GraphQL endpoint:

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

#### AWS AppSync real-time endpoint:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

With custom domain names, you can interact with both endpoints using a single domain. For example, if you configure `api.example.com` as your custom domain, you can interact with both your GraphQL and real-time endpoints using these URLs:

#### AWS AppSync custom domain GraphQL endpoint:

```
https://api.example.com/graphql
```

#### AWS AppSync custom domain real-time endpoint:

```
wss://api.example.com/graphql/realtime
```

#### Note

AWS AppSync APIs support only TLS 1.2 and TLS 1.3 for custom domain names.

## Registering and configuring a domain name

To set up custom domain names for your AWS AppSync APIs, you must have a registered internet domain name. You can register an internet domain using Amazon Route 53 domain registration or a third-party domain registrar of your choice. For more information about Route 53, see [What is Amazon Route 53?](#) in the *Amazon Route 53 Developer Guide*.

An API's custom domain name can be the name of a subdomain or the root domain (also known as the "zone apex") of a registered internet domain. After you create a custom domain name in AWS AppSync, you must create or update your DNS provider's resource record to map to your API endpoint. Without this mapping, API requests bound for the custom domain name cannot reach AWS AppSync.

## Creating a custom domain name in AWS AppSync

Creating a custom domain name for an AWS AppSync API sets up an Amazon CloudFront distribution. You must set up a DNS record to map the custom domain name to the CloudFront distribution domain name. This mapping is required to route API requests that are bound for the custom domain name AWS AppSync through the mapped CloudFront distribution. You must also provide a certificate for the custom domain name.

To set up the custom domain name or to update its certificate, you must have permission to update CloudFront distributions and describe the AWS Certificate Manager (ACM) certificate that you plan to use. To grant these permissions, attach the following AWS Identity and Access Management (IAM) policy statement to an IAM user, group, or role in your account:

```
{
  "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 AppSync supports custom domain names by leveraging Server Name Indication (SNI) on the CloudFront distribution. For more information about using custom domain names on a CloudFront distribution, including the required certificate format and the maximum certificate key length, see [Using HTTPS with CloudFront](#) in the *Amazon CloudFront Developer Guide*.

To set up a custom domain name as the API's hostname, the API owner must provide an SSL/TLS certificate for the custom domain name. To provide a certificate, do one of the following:

- Request a new certificate in ACM, or import a certificate issued by a third-party certificate authority into ACM in the `us-east-1` AWS Region (US East (N. Virginia)). For more information about ACM, see [What is AWS Certificate Manager?](#) in the *AWS Certificate Manager User Guide*.
- Provide an IAM server certificate. For more information, see [Managing server certificates in IAM](#) in the *IAM User Guide*.

## Wildcard custom domain names in AWS AppSync

AWS AppSync supports wildcard custom domain names. To configure a wildcard custom domain name, specify a wildcard character (\*) as the first subdomain of a custom domain. This represents all possible subdomains of the root domain. For example, the wildcard custom domain name `*.example.com` results in subdomains such as `a.example.com`, `b.example.com`, and `c.example.com`. All these subdomains route to the same domain.

To use a wildcard custom domain name in AWS AppSync, you must provide a certificate issued by ACM containing a wildcard name that can protect several sites in the same domain. For more information, see [ACM certificate characteristics](#) in the *AWS Certificate Manager User Guide*.

# Conflict Detection and Sync

## Versioned Data Sources

AWS AppSync currently supports versioning on DynamoDB data sources. Conflict Detection, Conflict Resolution, and Sync operations require a `Versioned` data source. When you enable versioning on a data source, AWS AppSync will automatically:

- Enhance items with object versioning metadata.
- Record changes made to items with AWS AppSync mutations to a *Delta* table.
- Maintain deleted items in the *Base* table with a “tombstone” for a configurable amount of time.

## Versioned Data Source Configuration

When you enable versioning on a DynamoDB data source, you specify the following fields:

### **BaseTableTTL**

The number of minutes to retain deleted items in the *Base* table with a “tombstone” - a metadata field indicating that the item has been deleted. You can set this value to 0 if you want items to be removed immediately when they are deleted. This field is required.

**DeltaSyncTableName**

The name of the table where changes made to items with AWS AppSync mutations are stored. This field is required.

**DeltaSyncTableTTL**

The number of minutes to retain items in the *Delta* table. This field is required.

## Delta Sync Table

AWS AppSync currently supports Delta Sync Logging for mutations using `PutItem`, `UpdateItem`, and `DeleteItem` DynamoDB operations.

When an AWS AppSync mutation changes an item in a versioned data source, a record of that change will be stored in a *Delta* table that is optimized for incremental updates. You can choose to use different *Delta* tables (e.g. one per type, one per domain area) for other versioned data sources or a single *Delta* table for your API. AWS AppSync recommends against using a single *Delta* table for multiple APIs to avoid the collision of primary keys.

The schema required for this table is as follows:

**ds\_pk**

A string value that is used as the partition key. It is constructed by concatenating the *Base* data source name and the ISO8601 format of the date the change occurred. (e.g. `Comments:2019-01-01`)

**ds\_sk**

A string value that is used as the sort key. It is constructed by concatenating the ISO8601 format of the time the change occurred, the primary key of the item, and the version of the item. The combination of these fields guarantees uniqueness for every entry in the *Delta* table (e.g. for a time of 09:30:00 and an ID of 1a and version of 2, this would be 09:30:00:1a:2)

**\_ttl**

A numeric value that stores the timestamp, in epoch seconds, when an item should be removed from the *Delta* table. This value is determined by adding the `DeltaSyncTableTTL` value configured on the data source to the moment when the change occurred. This field should be configured as the DynamoDB TTL Attribute.

The IAM role configured for use with the *Base* table must also contain permission to operate on the *Delta* table. In this example, the permissions policy for a *Base* table called `Comments` and a *Delta* table called `ChangeLog` is displayed:

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

## Versioned Data Source Metadata

AWS AppSync manages metadata fields on `Versioned` data sources on your behalf. Modifying these fields yourself may cause errors in your application or data loss. These fields include:

### **`_version`**

A monotonically increasing counter that is updated any time that a change occurs to an item.

### **`_lastChangedAt`**

A numeric value that stores the timestamp, in epoch milliseconds, when an item was last modified.

### **`_deleted`**

A Boolean “tombstone” value that indicates that an item has been deleted. This can be used by applications to evict deleted items from local data stores.

### **`_ttl`**

A numeric value that stores the timestamp, in epoch seconds, when an item should be removed from the underlying data source.

### **`ds_pk`**

A string value that is used as the partition key for *Delta* tables.

### **`ds_sk`**

A string value that is used as the sort key for *Delta* tables.

These metadata fields will impact the overall size of items in the underlying data source. AWS AppSync recommends reserving *500 bytes + Max Primary Key Size* of storage for versioned data source metadata when designing your application. To use this metadata in client applications, include the `_version`, `_lastChangedAt`, and `_deleted` fields on your GraphQL types and in the selection set for mutations.

## Conflict Detection and Resolution

When concurrent writes happen with AWS AppSync, you can configure Conflict Detection and Conflict Resolution strategies to handle updates appropriately. Conflict Detection determines if the mutation is in conflict with the actual written item in the data source. Conflict Detection is enabled by setting the value in the `SyncConfig` for the `conflictDetection` field to `VERSION`.

Conflict Resolution is the action that is taken in the event that a conflict is detected. This is determined by setting the Conflict Handler field in the `SyncConfig`. There are three Conflict Resolution strategies:

- `OPTIMISTIC_CONCURRENCY`
- `AUTOMERGE`
- `LAMBDA`

Each of these Conflict Resolution strategies are detailed in depth below.

Versions are automatically incremented by AppSync during write operations and should not be modified by clients or outside of a resolver configured with a version-enabled data source. Doing so will change the consistency behavior of the system and could result in data loss.

## Optimistic Concurrency

Optimistic Concurrency is a conflict resolution strategy that AWS AppSync provides for versioned data sources. When the conflict resolver is set to Optimistic Concurrency, if an incoming mutation is detected to have a version that differs from the actual version of the object, the conflict handler will simply reject the incoming request. Inside the GraphQL response, the existing item on the server that has the latest version will be provided. The client is then expected to handle this conflict locally and retry the mutation with the updated version of the item.

## Automerge

Automerge provides developers an easy way to configure a conflict resolution strategy without writing client-side logic to manually merge conflicts that were unable to be handled by other strategies. Automerge adheres to a strict rule set when merging data to resolve conflicts. The tenets of Automerge revolve around the underlying data type of the GraphQL field. They are as follows:

- Conflict on a scalar field: GraphQL scalar or any field that is not a collection (i.e List, Set, Map). Reject the incoming value for the scalar field and select the value existing in the server.
- Conflict on a list: GraphQL type and database type are lists. Concatenate the incoming list with the existing list in the server. The list values in the incoming mutation will be appended to the end of the list in the server. Duplicate values will be retained.
- Conflict on a set: GraphQL type is a list and database type is a Set. Apply a set union using incoming the set and the existing set in the server. This adheres to the properties of a Set, meaning no duplicate entries.
- When an incoming mutation adds a new field to the item, merge that on to the existing item.
- Conflict on a map: When the underlying data type in the database is a Map (i.e. key-value document), apply the above rules as it parses and processes each property of the Map.

Automerge is designed to automatically detect, merge, and retry requests with an updated version, absolving the client from needing to manually merge any conflicting data.

To show an example of how Automerge handles a Conflict on a Scalar type. We will use the following record as our starting point.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 4
}
```

Now an incoming mutation might be attempting to update the item but with an older version since the client has not synchronized with the server yet. That looks like this:

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 55,
  "_version" : 2
}
```

Notice the outdated version of 2 in the incoming request. During this flow, Automerge will merge the data by rejecting the 'jersey' field update to '55' and keep the value at '5' resulting in the following image of the item being saved in the server.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 5 # version is incremented every time automerge performs a merge that is
  stored on the server.
}
```

Given the state of the item shown above at version 5, now suppose an incoming mutation that attempts to mutate the item with the following image:

```
{
  "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
}
```

There are three points of interest in the incoming mutation. The name, a scalar, has been changed but two new fields “interests”, a Set, and “points”, a List, have been added. In this scenario, a conflict will be detected due to the version mismatch. Automerge adheres to its properties and rejects the name change due to it being a scalar and add on the non-conflicting fields. This results in the item that is saved in the server to appear as follows.

```
{
  "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
}
```

With the updated image of the item with version 6, now suppose an incoming mutation (with another version mismatch) tries to transform the item to the following:

```
{
  "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
}
```

Here we observe that the incoming field for “interests” has one duplicate value that exists in the server and two new values. In this case, since the underlying data type is a Set, Automerge will combine the values existing in the server with the ones in the incoming request and strip out any duplicates. Similarly there is a conflict on the “points” field where there is one duplicate value and one new value. But since the underlying data type here is a List, Automerge will simply append all values in the incoming request to the end of the values already existing in the server. The resulting merged image stored on the server would appear as follows:

```
{
  "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
}
```

Now let's assume the item stored in the server appears as follows, at version 8.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "stats": {
    "ppg": "35.4",
    "apg": "6.3"
  }
  "_version" : 8
}
```

But an incoming request tries to update the item with the following image, once again with a version mismatch:

```
{
  "id" : 1,
  "name" : "Nadia",
  "stats": {
    "ppg": "25.7",
    "rpg": "6.9"
  }
  "_version" : 3
}
```

Now in this scenario, we can see that the fields that already exist in the server are missing (interests, points, jersey). In addition, the value for "ppg" within the map "stats" is being edited, a new value "rpg" is being added, and "apg" is omitted. Automerge preserve the fields that have been omitted (note: if fields are intended to be removed, then the request must be tried again with the matching version), and so they will not be lost. It will also apply the same rules to fields within maps and therefore the change to "ppg" will be rejected whereas "apg" is preserved and "rpg", a new field, is added on. The resulting item stored in the server will now appear as:

```
{
  "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

Conflict Resolution options:

- **RESOLVE:** Replace the existing item with new item supplied in response payload. You can only retry the same operation on a single item at a time. Currently supported for `DynamoDB PutItem` & `UpdateItem`.
- **REJECT:** Reject the mutation and returns an error with the existing item in the GraphQL response. Currently supported for `DynamoDB PutItem`, `UpdateItem`, & `DeleteItem`.
- **REMOVE:** Remove the existing item. Currently supported for `DynamoDB DeleteItem`.

### The Lambda Invocation Request

The AWS AppSync DynamoDB resolver invokes the Lambda function specified in the `LambdaConflictHandlerArn`. It uses the same `service-role-arn` configured on the data source. The payload of the invocation has the following structure:

```
{
  "newItem": { ... },
  "existingItem": {... },
  "arguments": { ... },
  "resolver": { ... },
  "identity": { ... }
}
```

The fields are defined as follows:

#### **newItem**

The preview item, if the mutation succeeded.

#### **existingItem**

The item currently resided in DynamoDB table.

#### **arguments**

The arguments from the GraphQL mutation.

#### **resolver**

Information about the AWS AppSync resolver.

#### **identity**

Information about the caller. This field is set to null, if access with API key.

Example payload:

```
{
  "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": "AIDAAAAAAAAAAAAAAAAAAAA",
    "userArn": "arn:aws:iam::123456789012:user/appsync"
  }
}
```

### The Lambda Invocation Response

For `PutItem` and `UpdateItem` conflict resolution

**RESOLVE** the mutation. The response must be in the following format.

```
{
  "action": "RESOLVE",
  "item": { ... }
}
```

The `item` field represents an object that will be used to replace the existing item in the underlying data source. The primary key and sync metadata will be ignored if included in `item`.

**REJECT** the mutation. The response must be in the following format.

```
{
  "action": "REJECT"
}
```

For `DeleteItem` conflict resolution

**REMOVE** the item. The response must be in the following format.

```
{
  "action": "REMOVE"
}
```

**REJECT** the mutation. The response must be in the following format.

```
{
  "action": "REJECT"
}
```

The example Lambda function below checks who makes the call and the resolver name. If it is made by `jeffTheAdmin`, **REMOVE** the object for `DeletePost` resolver or **RESOLVE** the conflict with new item for `Update/Put` resolvers. If not, the mutation is **REJECT**.

```
exports.handler = async (event, context, callback) => {
  console.log("Event: " + JSON.stringify(event));
```

```
// Business logic goes here.
var response;
if ( event.identity.user == "jeffTheAdmin" ) {
    let resolver = event.resolver.field;

    switch(resolver) {
        case "deletePost":
            response = {
                "action" : "REMOVE"
            }
            break;

        case "updatePost":
        case "createPost":
            response = {
                "action" : "RESOLVE",
                "item": event.newItem
            }
            break;
        default:
            response = { "action" : "REJECT" };
    }
} else {
    response = { "action" : "REJECT" };
}

console.log("Response: " + JSON.stringify(response));
return response;
}
```

## Errors

### **ConflictUnhandled**

Conflict detection finds a version mismatch and the conflict handler rejects the mutation.

Example: Conflict resolution with an Optimistic Concurrency conflict handler. Or, Lambda conflict handler returned with `REJECT`.

### **ConflictError**

An internal error occurs when trying to resolve a conflict.

Example: Lambda conflict handler returned a malformed response. Or, cannot invoke Lambda conflict handler because the supplied resource `LambdaConflictHandlerArn` is not found.

### **MaxConflicts**

Max retry attempts were reached for conflict resolution.

Example: Too many concurrent requests on the same object. Before the conflict is resolved, the object is updated to a new version by another client.

### **BadRequest**

Client tries to update metadata fields (`_version`, `_ttl`, `_lastChangedAt`, `_deleted`).

Example: Client tries to update `_version` of an object with an update mutation.

### **DeltaSyncWriteError**

Failed to write delta sync record.

Example: Mutation succeeded, but an internal error occurred when trying to write to the delta sync table.

### InternalFailure

An internal error occurred.

## CloudWatch Logs

If an AWS AppSync API has enabled CloudWatch Logs with the logging settings set to Field-Level Logs enabled and log-level for the Field-Level Logs set to ALL, then AWS AppSync will emit Conflict Detection and Resolution information to the log group. For information about the format of the log messages, see the [documentation for Conflict Detection and Sync Logging \(p. 211\)](#).

## Sync Operations

Versioned data sources support Sync operations that allow you to retrieve all the results from a DynamoDB table and then receive only the data altered since your last query (the delta updates). When AWS AppSync receives a request for a Sync operation, it uses the fields specified in the request to determine if the *Base* table or the *Delta* table should be accessed.

- If the `lastSync` field is not specified, a Scan on the *Base* table is performed.
- If the `lastSync` field is specified, but the value is before the `current moment - DeltaSyncTTL`, a Scan on the *Base* table is performed.
- If the `lastSync` field is specified, and the value is on or after the `current moment - DeltaSyncTTL`, a Query on the *Delta* table is performed.

AWS AppSync returns the `startedAt` field to the response mapping template for all Sync operations. The `startedAt` field is the moment, in epoch milliseconds, when the Sync operation started that you can store locally and use in another request. If a pagination token was included in the request, this value will be the same as the one returned by the request for the first page of results.

For information about the format for Sync mapping templates, see [the mapping template reference \(p. 336\)](#).

## Monitoring and logging

To monitor your AWS AppSync GraphQL API and help debug issues related to requests, you can turn on logging to Amazon CloudWatch Logs.

## Setup and configuration

To turn on automatic logging on a GraphQL API, use the AWS AppSync console.

1. Sign in to the [AWS AppSync console](#).
2. On the **APIs** page, choose the name of a GraphQL API.
3. On your API's homepage, in the navigation pane, choose **Settings**.
4. Under **Logging**, do the following:
  - a. Turn on **Enable Logs**.
  - b. (Optional) For detailed request-level logging, select the check box under **Include verbose content**.
  - c. (Optional) Under **Field resolver log level**, choose your preferred field-level logging level (**None**, **Error**, or **All**).

- d. Under **Create or use an existing role**, choose **New role** to create a new AWS Identity and Access Management (IAM) that allows AWS AppSync to write logs to CloudWatch. Or, choose **Existing role** to select the Amazon Resource Name (ARN) of an existing IAM role in your AWS account.
5. Choose **Save**.

## Manual IAM role configuration

If you choose to use an existing IAM role, the role must grant AWS AppSync the required permissions to write logs to CloudWatch. To configure this manually, you must provide a service role ARN so that AWS AppSync can assume the role when writing the logs.

In the [IAM console](#), create a new policy with the name `AWSAppSyncPushToCloudWatchLogsPolicy` that has the following definition:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "*"
    }
  ]
}
```

Next, create a new role with the name `AWSAppSyncPushToCloudWatchLogsRole`, and attach the newly created policy to the role. Edit the trust relationship for this role to the following:

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

Copy the role ARN and use it when setting up logging for an AWS AppSync GraphQL API.

## CloudWatch metrics

You can use CloudWatch metrics to monitor and provide alerts about specific events that can result in HTTP status codes or from latency. The following metrics are emitted.

### **4XXError**

Errors resulting from requests that are not valid due to an incorrect client configuration. Typically, these errors happen anywhere outside of GraphQL processing. For example, these errors can occur when the request includes an incorrect JSON payload or an incorrect query, when the service is throttled, or when the authorization settings are misconfigured.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of these errors.

#### **5XXError**

Errors encountered during the running of a GraphQL query. For example, this can occur when invoking a query for an empty or incorrect schema. It can also occur when the Amazon Cognito user pool ID or AWS Region is not valid. Alternatively, this could also happen if AWS AppSync encounters an issue during processing of a request.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of these errors.

#### **Latency**

The time between when AWS AppSync receives a request from a client and when it returns a response to the client. This doesn't include the network latency encountered for a response to reach the end devices.

**Unit:** *Millisecond*. Use the Average statistic to evaluate expected latencies.

#### **Requests**

The number of requests (queries + mutations) that all APIs in your account have processed, by Region.

**Unit:** *Count*. The number of all requests processed in a particular Region.

#### **TokensConsumed**

Tokens are allocated to `Requests` based on the amount of resources (processing time and memory used) that a `Request` consumes. Usually, each `Request` consumes one token. However, a `Request` that consumes large amounts of resources is allocated additional tokens as needed.

**Unit:** *Count*. The number of tokens allocated to requests processed in a particular Region.

## Real-time subscriptions

All metrics are emitted in one dimension: **GraphQLAPIId**. This means that all metrics are coupled with GraphQL API IDs. The following metrics are related to GraphQL subscriptions over pure WebSockets:

#### **ConnectSuccess**

The number of successful WebSocket connections to AWS AppSync. It is possible to have connections without subscriptions.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the successful connections.

#### **ConnectClientError**

The number of WebSocket connections that were rejected by AWS AppSync because of client-side errors. This could imply that the service is throttled or that the authorization settings are misconfigured.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the client-side connection errors.

#### **ConnectServerError**

The number of errors that originated from AWS AppSync while processing connections. This usually happens when an unexpected server-side issue occurs.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the server-side connection errors.

#### **DisconnectSuccess**

The number of successful WebSocket disconnections from AWS AppSync.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the successful disconnections.

**DisconnectClientError**

The number of client errors that originated from AWS AppSync while disconnecting WebSocket connections.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the disconnection errors.

**DisconnectServerError**

The number of server errors that originated from AWS AppSync while disconnecting WebSocket connections.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the disconnection errors.

**SubscribeSuccess**

The number of subscriptions that were successfully registered to AWS AppSync through WebSocket. It's possible to have connections without subscriptions, but it's not possible to have subscriptions without connections.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the successful subscriptions.

**SubscribeClientError**

The number of subscriptions that were rejected by AWS AppSync because of client-side errors. This can occur when a JSON payload is incorrect, the service is throttled, or the authorization settings are misconfigured.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the client-side subscription errors.

**SubscribeServerError**

The number of errors that originated from AWS AppSync while processing subscriptions. This usually happens when an unexpected server-side issue occurs.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the server-side subscription errors.

**UnsubscribeSuccess**

The number of unsubscribe requests that were successfully processed.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the successful unsubscribe requests.

**UnsubscribeClientError**

The number of unsubscribe requests that were rejected by AWS AppSync because of client-side errors.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the client-side unsubscribe request errors.

**UnsubscribeServerError**

The number of errors that originated from AWS AppSync while processing unsubscribe requests. This usually happens when an unexpected server-side issue occurs.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the server-side unsubscribe request errors.

**PublishDataMessageSuccess**

The number of subscription event messages that were successfully published.



**Unit:** *Count*. Use the Sum statistic to get the total of the subscription event messages were successfully published.

**PublishDataMessageClientError**

The number of subscription event messages that failed to publish because of client-side errors.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the client-side publishing subscription events errors.

**PublishDataMessageServerError**

The number of errors that originated from AWS AppSync while publishing subscription event messages. This usually happens when an unexpected server-side issue occurs.

**Unit:** *Count*. Use the Sum statistic to get the total occurrences of the server-side publishing subscription events errors.

**PublishDataMessageSize**

The size of subscription event messages published.

**Unit:** *Bytes*.

**ActiveConnections**

The number of concurrent WebSocket connections from clients to AWS AppSync in 1 minute.

**Unit:** *Count*. Use the Sum statistic to get the total opened connections.

**ActiveSubscriptions**

The number of concurrent subscriptions from clients in 1 minute.

**Unit:** *Count*. Use the Sum statistic to get the total active subscriptions.

**ConnectionDuration**

The amount of time that the connection stays open.

**Unit:** *Milliseconds*. Use the Average statistic to evaluate connection duration.

**InvalidationSuccess**

The number of subscriptions successfully invalidated (unsubscribed) by a mutation with `$extensions.invalidateSubscriptions()`.

**Unit:** *Count*. Use the Sum statistic to retrieve the total number of subscriptions that were successfully unsubscribed.

## CloudWatch Logs

You can configure two types of logging on any new or existing GraphQL API: request-level and field-level.

### Request-level logs

When request-level logging (**Include verbose content**) is configured, the following information is logged:

- The number of tokens consumed
- The request and response HTTP headers
- The GraphQL query that is running in the request

- The overall operation summary
- New and existing GraphQL subscriptions that are registered

## Field-level logs

When field-level logging is configured, the following information is logged:

- Generated request mapping with source and arguments for each field
- The transformed response mapping for each field, which includes the data as a result of resolving that field
- Tracing information for each field

If you turn on logging, AWS AppSync manages the CloudWatch Logs. The process includes creating log groups and log streams, and reporting to the log streams with these logs.

When you turn on logging on a GraphQL API and make requests, AWS AppSync creates a log group and log streams under the log group. The log group is named following the `/aws/appsync/apis/{graphql_api_id}` format. Within each log group, the logs are further divided into log streams. These are ordered by **Last Event Time** as logged data is reported.

Every log event is tagged with the `x-amzn-RequestId` of that request. This helps you filter log events in CloudWatch to get all logged information about that request. You can get the RequestId from the response headers of every GraphQL AWS AppSync request.

The field-Level logging is configured with the following log levels:

- **None** - No field-level logs are captured.
- **Error** - **Logs the following information only for the fields that are in error:**
  - The error section in the server response
  - Field-level errors
  - The generated request/response functions that got resolved for error fields
- **All** - **Logs the following information for all fields in the query:**
  - Field-level tracing information
  - The generated request/response functions that got resolved for each field

## Benefits of monitoring

You can use logging and metrics to identify, troubleshoot, and optimize your GraphQL queries. For example, these will help you debug latency issues using the tracing information that is logged for each field in the query. To demonstrate this, suppose you are using one or more resolvers nested in a GraphQL query. A sample field operation in CloudWatch Logs might look similar to the following:

```
{
  "path": [
    "singlePost",
    "authors",
    0,
    "name"
  ],
  "parentType": "Post",
  "returnType": "String!",
  "fieldName": "name",
  "startOffset": 416563350,
  "duration": 11247
}
```

```
}

```

This might correspond to a GraphQL schema, similar to the following:

```
type Post {
  id: ID!
  name: String!
  authors: [Author]
}

type Author {
  id: ID!
  name: String!
}

type Query {
  singlePost(id:ID!): Post
}
```

In the preceding log results, **path** shows a single item in your data returned from running a query named `singlePost()`. In this example, it's representing the **name** field at the first index (0). The **startOffset** gives an offset from the start of the GraphQL query operation. The **duration** is the total time to resolve the field. These values can be useful to troubleshoot why data from a particular data source might be running slower than expected, or if a specific field is slowing down the entire query. For example, you might choose to increase provisioned throughput for an Amazon DynamoDB table, or remove a specific field from a query that is causing the overall operation to perform poorly.

As of May 8, 2019, AWS AppSync generates log events as fully structured JSON. This can help you use log analytics services such as CloudWatch Logs Insights and Amazon OpenSearch Service to understand the performance of your GraphQL requests and usage characteristics of your schema fields. For example, you can easily identify resolvers with large latencies that may be the root cause of a performance issue. You can also identify the most and least frequently used fields in your schema and assess the impact of deprecating GraphQL fields.

## Conflict detection and sync logging

If an AWS AppSync API has logging to CloudWatch Logs configured with the **Field resolver log level** set to **All**, then AWS AppSync emits conflict detection and resolution information to the log group. This provides granular insight into how the AWS AppSync API responded to a conflict. To help you interpret the response, the following information is provided in the logs:

**conflictType**

Details whether a conflict occurred due to a version mismatch or the customer-supplied condition.

**conflictHandlerConfigured**

States the conflict handler configured on the resolver at the time of the request.

**message**

Provides information on how the conflict was detected and resolved.

**syncAttempt**

The number of tries the server attempted in order to synchronize the data before ultimately rejecting the request.

**data**

If the conflict handler configured is `Automerge`, this field is populated to show what decision `Automerge` took for each field. Actions provided can be:

- **REJECTED** - When Automerge rejects the incoming field value in favor of the value in the server.
- **ADDED** - When Automerge adds on the incoming field due to no pre-existing value in the server.
- **APPENDED** - When Automerge appends the incoming values to the values for the List that exists in the server.
- **MERGED** - When Automerge merges the incoming values to the values for the Set that exists in the server.

## Using token counts to optimize your requests

Requests that consume less than or equal to 1,500 KB-seconds of memory and vCPU time are allocated one token. Requests with resource consumption greater than 1,500 KB-seconds receive additional tokens. For example, if a request consumes 3,350 KB-seconds, AWS AppSync allocates three tokens (rounded up to the next integer value) to the request. By default, AWS AppSync allocates a maximum of 2,000 request tokens per second to the APIs in your account, per AWS Region. If your APIs each use an average of two tokens per second, you'll be limited to 1,000 requests per second. If you need more tokens per second than the allotted amount, you can submit a request to increase the default quota for the rate of request tokens. For more information, see [AWS AppSync endpoints and quotas](#) in the *AWS General Reference guide* and [Requesting a quota increase](#) in the *Service Quotas User Guide*.

A high per-request token count could indicate that there's an opportunity to optimize your requests and improve the performance of your API. Factors that can increase your per-request token count include:

- The size and complexity of your GraphQL schema.
- The complexity of request and response mapping templates.
- The number of resolver invocations per request.
- The amount of data returned from resolvers.
- The latency of downstream data sources.
- Schema and query designs that require successive data source calls (as opposed to parallel or batched calls).
- Logging configuration, particularly field-level and verbose log content.

### Note

In addition to AWS AppSync metrics and logs, clients can access the number of tokens consumed in a request via the response header `x-amzn-appsync-TokensConsumed`.

## Log type reference

### RequestSummary

- **requestId**: Unique identifier for the request.
- **graphqlAPIId**: ID of the GraphQL API making the request.
- **statusCode**: HTTP status code response.
- **latency**: End-to-end latency of the request, in nanoseconds, as an integer.

```
{
  "logType": "RequestSummary",
  "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
  "graphqlAPIId": "pmo28inf75eepg63qxq4ekoeg4",
  "statusCode": 200,
  "latency": 242000000
}
```

## ExecutionSummary

- **requestId**: Unique identifier for the request.
- **graphqlAPIId**: ID of the GraphQL API making the request.
- **startTime**: The start timestamp of GraphQL processing for the request, in RFC 3339 format.
- **endTime**: The end timestamp of GraphQL processing for the request, in RFC 3339 format.
- **duration**: The total elapsed GraphQL processing time, in nanoseconds, as an integer.
- **version**: The schema version of the ExecutionSummary.
- **parsing**:
  - **startOffset**: The start offset for parsing, in nanoseconds, relative to the invocation, as an integer.
  - **duration**: The time spent parsing, in nanoseconds, as an integer.
- **validation**:
  - **startOffset**: The start offset for validation, in nanoseconds, relative to the invocation, as an integer.
  - **duration**: The time spent performing validation, in nanoseconds, as an integer.

```
{
  "duration": 217406145,
  "logType": "ExecutionSummary",
  "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
  "startTime": "2019-01-01T06:06:18.956Z",
  "endTime": "2019-01-01T06:06:19.174Z",
  "parsing": {
    "startOffset": 49033,
    "duration": 34784
  },
  "version": 1,
  "validation": {
    "startOffset": 129048,
    "duration": 69126
  },
  "graphqlAPIId": "pmo28inf75eepg63qxq4ekoeg4"
}
```

## Tracing

- **requestId**: Unique identifier for the request.
- **graphqlAPIId**: ID of the GraphQL API making the request.
- **startOffset**: The start offset for field resolution, in nanoseconds, relative to the invocation, as an integer.
- **duration**: The time spent resolving the field, in nanoseconds, as an integer.
- **fieldName**: The name of the field being resolved.
- **parentType**: The parent type of the field being resolved.
- **returnType**: The return type of the field being resolved.
- **path**: A list of path segments, starting at the root of the response and ending with the field being resolved.
- **resolverArn**: The ARN of the resolver used for field resolution. Might not be present on nested fields.

```
{
```

```
"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"
}
```

## Analyzing your logs with CloudWatch Logs Insights

The following are examples of queries you can run to get actionable insights into the performance and health of your GraphQL operations. These examples are available as sample queries in the CloudWatch Logs Insights console. In the [CloudWatch console](#), choose **Logs Insights**, select the AWS AppSync log group for your GraphQL API, and then choose **AWS AppSync queries** under **Sample queries**.

The following query returns the top 10 GraphQL requests with maximum tokens consumed:

```
filter @message like "Tokens Consumed"
| parse @message "*" Tokens Consumed: "*" as requestId, tokens
| sort tokens desc
| display requestId, tokens
| limit 10
```

The following query returns the top 10 resolvers with maximum latency:

```
fields resolverArn, duration
| filter logType = "Tracing"
| limit 10
| sort duration desc
```

The following query returns the most frequently invoked resolvers:

```
fields ispresent(resolverArn) as isRes
| stats count() as invocationCount by resolverArn
| filter isRes and logType = "Tracing"
| limit 10
| sort invocationCount desc
```

The following query returns resolvers with the most errors in mapping templates:

```
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
```

The following query returns resolver latency statistics:

```
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
```

The following query returns field latency statistics:

```
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
```

The results of CloudWatch Logs Insights queries can be exported to CloudWatch dashboards.

## Analyze your logs with OpenSearch Service

You can search, analyze, and visualize your AWS AppSync logs with Amazon OpenSearch Service to identify performance bottlenecks and root causes of operational issues. You can identify resolvers with the maximum latency and errors. In addition, you can use OpenSearch Dashboards to create dashboards with powerful visualizations. OpenSearch Dashboards is an open source data visualization and exploration tool available in OpenSearch Service. Using OpenSearch Dashboards, you can continuously monitor the performance and health of your GraphQL operations. For example, you can create dashboards to visualize the P90 latency of your GraphQL requests and drill down into the P90 latencies of each resolver.

When using OpenSearch Service, use **"cwl"** as the filter pattern to search OpenSearch indexes. OpenSearch Service indexes the logs streamed from CloudWatch Logs with a prefix of **"cwl-"**. To differentiate AWS AppSync API logs from other CloudWatch logs sent to OpenSearch Service, we recommend adding an additional filter expression of `graphqlAPIID.keyword=YourGraphQLAPIID` to your search.

## Log format migration

Log events that AWS AppSync generates on or after May 8, 2019 are formatted as fully structured JSON. To analyze GraphQL requests prior to May 8, 2019, you can migrate older logs to fully structured JSON using a script available in the [GitHub Sample](#). If you need to use the log format prior to May 8, 2019, create a support ticket with the following settings: set **Type** to **Account Management** and then set **Category** to **General Account Question**.

You can also use [metric filters](#) in CloudWatch to turn log data into numerical CloudWatch metrics, so that you can graph or set an alarm on them.

## Tracing with AWS X-Ray

You can use [AWS X-Ray](#) to trace requests as they are executed in AWS AppSync. You can use X-Ray with AWS AppSync in all AWS Regions where X-Ray is available. X-Ray gives you a detailed overview of an entire GraphQL request. This enables you to analyze latencies in your APIs and their underlying resolvers and data sources. You can use an X-Ray service map to view the latency of a request, including any AWS services that are integrated with X-Ray. You can also configure sampling rules to tell X-Ray which requests to record, and at what sampling rates, according to criteria that you specify.

For more information about sampling in X-Ray, see [Configuring Sampling Rules in the AWS X-Ray Console](#).

## Setup and Configuration

You can enable X-Ray tracing for a GraphQL API through the AWS AppSync console.

1. Sign in to the AWS AppSync console.
2. Choose **Settings** from the navigation panel.
3. Under **X-Ray**, turn on **Enable X-Ray**.
4. Choose **Save**. X-Ray tracing is now enabled for your API.

If you're using the AWS CLI or AWS CloudFormation, you can also enable X-Ray tracing when you create a new AWS AppSync API, or update an existing AWS AppSync API, by setting the `xrayEnabled` property to `true`.

When X-Ray tracing is enabled for an AWS AppSync API, an AWS Identity and Access Management [service-linked role](#) is automatically created in your account with the appropriate permissions. This allows AWS AppSync to send traces to X-Ray in a secure way.

## Tracing Your API with X-Ray

### Sampling

By using sampling rules, you can control the amount of data that you record in AWS AppSync, and can modify sampling behavior on the fly without modifying or redeploying your code. For example, this rule samples requests to the GraphQL API with the API ID `3n572shhcpfokwhdnq1ogu59v6`.

- **Rule name** — `test-sample`
- **Priority** — `10`
- **Reservoir size** — `10`
- **Fixed rate** — `10`
- **Service name** — `*`
- **Service type** — `AWS::AppSync::GraphQLAPI`
- **HTTP method** — `*`
- **Resource ARN** — `arn:aws:appsync:us-west-2:123456789012:apis/3n572shhcpfokwhdnq1ogu59v6`
- **Host** — `*`

### Understanding Traces

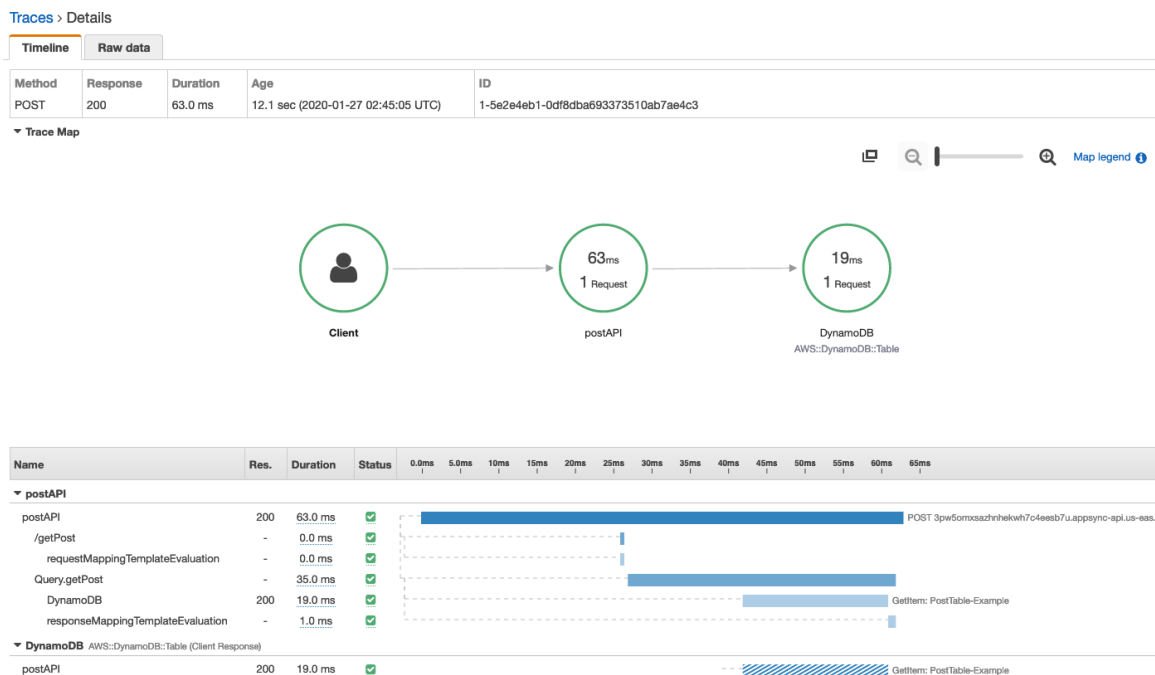
When you enable X-Ray tracing for your GraphQL API, you can use the X-Ray trace detail page to examine detailed latency information about requests made to your API. The following example shows the trace view along with the service map for this specific request. The request was made to an API called `postAPI` with a `Post` type, whose data is contained in an Amazon DynamoDB table called `PostTable-Example`.

The following trace image corresponds to the following GraphQL query:

```
query getPost {
  getPost(id: "1") {
    id
    title
  }
}
```



The resolver for the `getPost` query uses the underlying DynamoDB data source. The following trace view shows the call to DynamoDB, as well as the latencies of various parts of the query's execution:



- In the preceding image, `/getPost` represents the complete path to the element that is being resolved. In this case, because `getPost` is a field on the root `Query` type, it appears directly after the root of the path.
- `requestMappingTemplateEvaluation` represents the time spent by AWS AppSync evaluating the request mapping template for this element in the query.
- `Query.getPost` represents a type and field (in `Type.field` format). It can contain multiple subsegments, depending on the structure of the API and the request being traced.
  - `DynamoDB` represents the data source that is attached to this resolver. It contains the latency for the network call to DynamoDB to resolve the field.
  - `responseMappingTemplateEvaluation` represents the time spent by AWS AppSync evaluating the response mapping template for this element in the query.

When you view traces in X-Ray, you can get additional contextual and metadata information about the subsegments in the AWS AppSync segment by choosing the subsegments and exploring the detailed view.

For certain deeply nested or complex queries, note that the segment delivered to X-Ray by AWS AppSync can be larger than the maximum size allowed for segment documents, as defined in [AWS X-Ray Segment Documents](#). X-Ray doesn't display segments that exceed the limit.

## Logging AWS AppSync API calls using AWS CloudTrail

AWS AppSync is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or AWS service in AWS AppSync. CloudTrail captures all API calls for AWS AppSync as events. The calls captured include calls from the AWS AppSync console and from code calls to the AWS AppSync

APIs. You can use the information collected by CloudTrail to determine the request that was made to AWS AppSync, the IP address of the requester, who made the request, when the request was made, and additional details.

You can create a *trail* to enable continuous delivery of CloudTrail events to an Amazon Simple Storage Service (Amazon S3) bucket, including events for AWS AppSync. If you don't configure a trail, you can still view the most recent events in the CloudTrail console.

**Important**

Not all GraphQL actions are currently logged. AppSync does not log Query and Mutation actions to CloudTrail.

For more information about CloudTrail, see the [AWS CloudTrail User Guide](#).

## AWS AppSync information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. In the CloudTrail console in **Event history**, you can view, search, and download recent events in your AWS account. For more information, see [Viewing Events with CloudTrail Event History](#) in the *AWS CloudTrail User Guide*.

For an ongoing record of events in your AWS account, including events for AWS AppSync, create a trail. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following in the *AWS CloudTrail User Guide*:

- [Creating a Trail For Your AWS Account](#)
- [AWS Service Integrations With CloudTrail Logs](#)
- [Configuring Amazon SNS Notifications for CloudTrail](#)
- [Receiving CloudTrail Log Files from Multiple Regions](#)
- [Receiving CloudTrail Log Files from Multiple Accounts](#)

CloudTrail logs all AWS AppSync API operations. For example, calls to the `CreateGraphQLApi`, `CreateDataSource`, and `ListResolvers` APIs generate entries in the CloudTrail log files. These and other operations are documented in the [AWS AppSync API Reference](#).

Every event or log entry contains information about who generated the request. The identity information helps you determine:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see [CloudTrail userIdentity Element](#) in the *AWS CloudTrail User Guide*.

## Understanding AWS AppSync log file entries

CloudTrail delivers events as log files that contain one or more log entries. An event represents a single request from any source and includes information about the requested operation, the date and time of the operation, the request parameters, and so on. Because these log files aren't an ordered stack trace of the public API calls, they don't appear in any specific order.

The following example CloudTrail log entry demonstrates the `CreateApiKey` operation.

```
{
  "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"
  ]
}
```

The following example CloudTrail log entry demonstrates the `ListApiKeys` operation.

```
{
  "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": "11112223333"
}
]
```

The following example CloudTrail log entry demonstrates the `DeleteApiKey` operation.

```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::11112223333:user/Alice",
      "accountId": "11112223333",
      "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": "11112223333"
  }
]
```

# Authorization and Authentication

## Topics

- [API\\_KEY Authorization \(p. 222\)](#)
- [AWS\\_LAMBDA Authorization \(p. 222\)](#)
- [AWS\\_IAM Authorization \(p. 225\)](#)
- [OPENID\\_CONNECT Authorization \(p. 227\)](#)
- [AMAZON\\_COGNITO\\_USER\\_POOLS Authorization \(p. 227\)](#)
- [Using Additional Authorization Modes \(p. 228\)](#)
- [Fine-Grained Access Control \(p. 230\)](#)
- [Filtering Information \(p. 232\)](#)
- [Data source access \(p. 232\)](#)
- [Authorization Use Cases \(p. 233\)](#)
- [Using AWS WAF to protect your APIs \(p. 241\)](#)

This section describes options for configuring security and data protection for your applications.

There are five ways you can authorize applications to interact with your AWS AppSync GraphQL API. You specify which authorization type you use by specifying one of the following authorization type values in your AWS AppSync API or CLI call:

- **API\_KEY**

For using API keys.

- **AWS\_LAMBDA**

For using an AWS Lambda function.

- **AWS\_IAM**

For using AWS Identity and Access Management ([IAM](#)) permissions.

- **OPENID\_CONNECT**

For using your OpenID Connect provider.

- **AMAZON\_COGNITO\_USER\_POOLS**

For using an Amazon Cognito user pool.

These basic authorization types work for most developers. For more advanced use cases, you can add additional authorization modes through the console, the CLI, and AWS CloudFormation. For additional authorization modes, AppSync provides an authorization type that takes the values listed above (that is, `API_KEY`, `AWS_LAMBDA`, `AWS_IAM`, `OPENID_CONNECT`, and `AMAZON_COGNITO_USER_POOLS`).

When you specify `API_KEY`, `AWS_LAMBDA`, or `AWS_IAM` as the main or default authorization type, you can't specify them again as one of the additional authorization modes. Similarly, you can't duplicate `API_KEY`, `AWS_LAMBDA` or `AWS_IAM` inside the additional authorization modes. You can use multiple Amazon Cognito User Pools and OpenID Connect providers. However, you can't use duplicate Amazon Cognito User Pools or OpenID Connect providers between the default authorization mode and any of the

additional authorization modes. You can specify different clients for your Amazon Cognito User Pool or OpenID Connect provider using the corresponding configuration regular expression.

## API\_KEY Authorization

Unauthenticated APIs require more strict throttling than authenticated APIs. One way to control throttling for unauthenticated GraphQL endpoints is through the use of API keys. An API key is a hard-coded value in your application that is generated by the AWS AppSync service when you create an unauthenticated GraphQL endpoint. You can rotate API keys from the console, from the CLI, or from the [AWS AppSync API Reference](#).

API keys are configurable for up to 365 days, and you can extend an existing expiration date for up to another 365 days from that day. API Keys are recommended for development purposes or use cases where it's safe to expose a public API.

On the client, the API key is specified by the header `x-api-key`.

For example, if your `API_KEY` is `'ABC123'`, you can send a GraphQL query via `curl` as follows:

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "x-api-key:ABC123" -d '{ "query":  
  "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

## AWS\_LAMBDA Authorization

You can implement your own API authorization logic using an AWS Lambda function. You can use a Lambda function for either your primary or secondary authorizer, but there may only be one Lambda authorization function per API. When using Lambda functions for authorization, the following applies:

- An authorization token must not exceed 2048 characters
- A Lambda function must not return more than 5MB of contextual data for resolvers.

For example, if your authorization token is `'ABC123'`, you can send a GraphQL query via `curl` as follows:

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "Authorization:ABC123" -d  
'{ "query":  
  "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

Lambda functions are called before each query or mutation, but their return value is cached: repeated requests will invoke the function only once before it is cached based on the API ID and the authentication token. By default, this caching time is 300 seconds (5 minutes,) but this can be overridden at an API level or by setting the `ttlOverride` value in a function's return value.

A regular expression that validates authorization tokens before the function is called can be specified if desired. These regular expressions are used to validate that an authorization token is of the correct format before your function is called. Any request using a token which does not match this regular expression will be denied automatically.

Lambda functions used for authorization require a principal policy for `appsync.amazonaws.com` to be applied on them to allow AWS AppSync to call them. This action is done automatically in the AWS AppSync console; The AWS AppSync console does *not* remove the policy. For more information on attaching policies to Lambda functions, see [Resource-based policies](#) in the AWS Lambda Developer Guide.

The Lambda function you specify will receive an event with the following shape:

```
{
  "authorizationToken": "ExampleAUTHtoken123123123",
  "requestContext": {
    "apiId": "aaaaaa123123123example123",
    "accountId": "111122223333",
    "requestId": "f4081827-1111-4444-5555-5cf4695f339f",
    "queryString": "mutation CreateEvent {...}\n\nquery MyQuery {...}\n",
    "operationName": "MyQuery",
    "variables": {}
  }
}
```

The authorization function must return at least `isAuthorized`, a boolean indicating if the request is authorized. AWS AppSync recognizes the following keys returned from Lambda authorization functions:

`isAuthorized` (boolean, required)

A boolean value indicating if the value in `authorizationToken` is authorized to make calls to the GraphQL API.

If this value is true, execution of the GraphQL API continues. If this value is false, an `UnauthorizedException` is raised

`deniedFields` (list of string, optional)

A list of which are forcibly changed to null, even if a value was returned from a resolver.

Each item is either a fully qualified field ARN in the form of `arn:aws:appsync:us-east-1:111122223333:apis/GraphQLApiId/types/TypeName/fields/FieldName` or a short form of `TypeName.FieldName`. The full ARN form should be used when two APIs share a lambda function authorizer and there might be ambiguity between common types and fields between the two APIs.

`resolverContext` (JSON Object, optional)

A JSON object visible as `$ctx.identity.resolverContext` in resolver templates. For example, if the following structure is returned by a resolver:

```
{
  "isAuthorized":true
  "resolverContext": {
    "banana":"very yellow",
    "apple":"very green"
  }
}
```

The value of `$ctx.identity.resolverContext.apple` in resolver templates will be "very green". The `resolverContext` object only supports key-value pairs. Nested keys are not supported.

### Warning

The total size of this JSON object must not exceed 5MB.

`ttlOverride` (integer, optional)

The number of seconds that the response should be cached for. If no value is returned, the value from the API (if configured) or the default of 300 seconds (five minutes) is used. If this is 0, the response is not cached.

Lambda authorizers have a timeout of 10 seconds. We recommend designing functions to execute in the shortest amount of time as possible to scale the performance of your API.

Multiple AWS AppSync APIs can share a single authentication Lambda function. Cross account authorizer use is not permitted.

When sharing an authorization function between multiple APIs, be aware that short-form field names (*typename.fieldname*) may inadvertently hide fields. To disambiguate a field in `deniedFields`, you can specify an unambiguous field ARN in the form of `arn:aws:appsync:region:accountId:apis/GraphQLApiId/types/typeName/fields/fieldName`.

To add a Lambda function as the default authorization mode in AWS AppSync:

Console

### Adding a Lambda Authorizer with the AWS AppSync Console

1. Log into the AWS AppSync Console and navigate to the API you wish to update.
2. Navigate to the Settings page for your API.

Change the API-Level authorization to **AWS Lambda**.

3. Choose the AWS Region and Lambda ARN to authorize API calls against.

#### Note

The appropriate principal policy will be added automatically, allowing AWS AppSync to call your Lambda function.

4. Optionally, set the response TTL and token validation regular expression.

AWS CLI

1. Attach the following policy to the Lambda function being used:

```
aws lambda add-permission --function-name "my-function" --statement-id "appsync" --principal appsync.amazonaws.com --action lambda:InvokeFunction --output text
```

#### Important

If you want the policy of the function to be locked to a single GraphQL API, you can run this command:

```
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. Update your AWS AppSync API to use the given Lambda function ARN as the authorizer:

```
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

You can also include other configuration options such as the token regular expression.

The following example describes a Lambda function that demonstrates the various authentication and failure states a Lambda function can have when used as a AWS AppSync authorization mechanism:



```
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 {}
```

## AWS\_IAM Authorization

This authorization type enforces the [AWS Signature Version 4 Signing Process](#) on the GraphQL API. You can associate Identity and Access Management (IAM) access policies with this authorization type. Your application can leverage this association by using an access key (which consists of an access key ID and secret access key) or by using short-lived, temporary credentials provided by Amazon Cognito Federated Identities.

If you want a role that has access to perform all data operations:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "appsync:GraphQL"
      ],
      "Resource": [
        "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/*"
      ]
    }
  ]
}
```

```
}

```

You can find `YourGraphQLApiId` from the main API listing page in the AppSync console, directly under the name of your API. Alternatively you can retrieve it with the CLI: `aws appsync list-graphql-apis`

If you want to restrict access to just certain GraphQL operations, you can do this for the root `Query`, `Mutation`, and `Subscription` fields.

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

For example, suppose you have the following schema and you want to restrict access to getting all posts:

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

type Query {
  posts:[Post!]!
}

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

The corresponding IAM policy for a role (that you could attach to an Amazon Cognito identity pool, for example) would look like the following:

```
{
  "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 Authorization

This authorization type enforces [OpenID Connect](#) (OIDC) tokens provided by an OIDC-compliant service. Your application can leverage users and privileges defined by your OIDC provider for controlling access.

An Issuer URL is the only required configuration value that you provide to AWS AppSync (for example, `https://auth.example.com`). This URL must be addressable over HTTPS. AWS AppSync appends `/.well-known/openid-configuration` to the issuer URL and locates the OpenID configuration at `https://auth.example.com/.well-known/openid-configuration` per the [OpenID Connect Discovery](#) specification. It expects to retrieve an [RFC5785](#) compliant JSON document at this URL. This JSON document must contain a `jwks_uri` key, which points to the JSON Web Key Set (JWKS) document with the signing keys.

AWS AppSync requires the JWKS to contain JSON fields of `alg`, `key`, and `kid`.

AWS AppSync supports RS256, RS384, and RS512 as signing algorithms. Tokens issued by the provider must include the time at which the token was issued (`iat`) and may include the time at which it was authenticated (`auth_time`). You can provide TTL values for issued time (`iatTTL`) and authentication time (`authTTL`) in your OpenID Connect configuration for additional validation. If your provider authorizes multiple applications, you can also provide a regular expression (`clientId`) that is used to authorize by client ID.

To validate multiple client IDs use the pipeline operator (`|`) which is an “or” in regular expression. For example, if your OIDC application has four clients with client IDs such as `0A1S2D`, `1F4G9H`, `1J6L4B`, `6GS5MG`, to validate for only the first three client ids you would place `1F4G9H|1J6L4B|6GS5MG` in the client ID field.

## AMAZON\_COGNITO\_USER\_POOLS Authorization

This authorization type enforces OIDC tokens provided by Amazon Cognito User Pools. Your application can leverage the users and groups in your user pools and associate these with GraphQL fields for controlling access.

When using Amazon Cognito User Pools, you can create groups that users belong to. This information is encoded in a JWT token that your application sends to AWS AppSync in an authorization header when sending GraphQL operations. You can use GraphQL directives on the schema to control which groups can invoke which resolvers on a field, thereby giving more controlled access to your customers.

For example, suppose you have the following GraphQL schema:

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Query {  
  posts:[Post!]!  
}  
  
type Mutation {  
  addPost(id:ID!, title:String!):Post!  
}
```

```
...
```

If you have two groups in Amazon Cognito User Pools - bloggers and readers - and you want to restrict the readers so that they cannot add new entries, then your schema should look like this:

```
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"])  
}  
...
```

Note that you can omit the `@aws_auth` directive if you want to default to a specific grant-or-deny strategy on access. You can specify the grant-or-deny strategy in the user pool configuration when you create your GraphQL API via the console or via the following CLI command:

```
$ 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"}'
```

## Using Additional Authorization Modes

When you add additional authorization modes, you can directly configure the authorization setting at the AWS AppSync GraphQL API level (that is, the `authenticationType` field that you can directly configure on the `GraphQLApi` object) and it acts as the default on the schema. This means that any type that doesn't have a specific directive has to pass the API level authorization setting.

At the schema level, you can specify additional authorization modes using directives on the schema. You can specify authorization modes on individual fields in the schema. For example, for `API_KEY` authorization you would use `@aws_api_key` on schema object type definitions/fields. The following directives are supported on schema fields and object type definitions:

- `@aws_api_key` - To specify the field is `API_KEY` authorized.
- `@aws_iam` - To specify that the field is `AWS_IAM` authorized.
- `@aws_oidc` - To specify that the field is `OPENID_CONNECT` authorized.
- `@aws_cognito_user_pools` - To specify that the field is `AMAZON_COGNITO_USER_POOLS` authorized.
- `@aws_lambda` - To specify that the field is `AWS_LAMBDA` authorized.

You can't use the `@aws_auth` directive along with additional authorization modes. `@aws_auth` works only in the context of `AMAZON_COGNITO_USER_POOLS` authorization with no additional authorization modes. However, you can use the `@aws_cognito_user_pools` directive in place of the `@aws_auth` directive, using the same arguments. The main difference between the two is that you can specify `@aws_cognito_user_pools` on any field and object type definitions.

To understand how the additional authorization modes work and how they can be specified on a schema, let's have a look at the following schema:

```
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!
}
...
```

For this schema, assume that `AWS_IAM` is the default authorization type on the AWS AppSync GraphQL API. This means that fields that don't have a directive are protected using `AWS_IAM`. For example, that's the case for the `getPost` field on the `Query` type. Schema directives enable you to use more than one authorization mode. For example, you can have `API_KEY` configured as an additional authorization mode on the AWS AppSync GraphQL API, and you can mark a field using the `@aws_api_key` directive (for example, `getAllPosts` in this example). Directives work at the field level so you need to give `API_KEY` access to the `Post` type too. You can do this either by marking each field in the `Post` type with a directive, or by marking the `Post` type with the `@aws_api_key` directive.

To further restrict access to fields in the `Post` type you can use directives against individual fields in the `Post` type as shown following.

For example, you can add a `restrictedContent` field to the `Post` type and restrict access to it by using the `@aws_iam` directive. `AWS_IAM` authenticated requests could access `restrictedContent`, however, `API_KEY` requests wouldn't be able to access it.

```
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
}
...
```

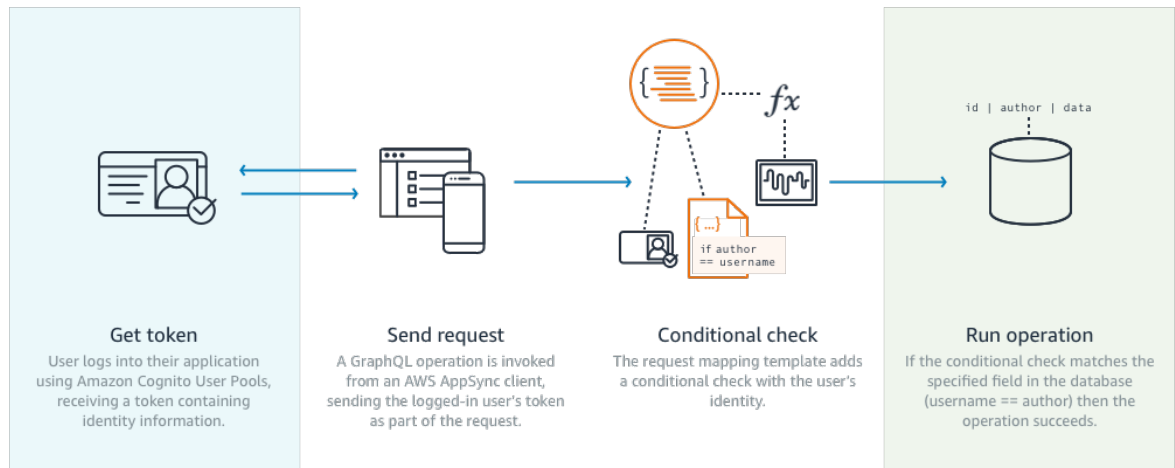
## Fine-Grained Access Control

The preceding information demonstrates how to restrict or grant access to certain GraphQL fields. If you want to set access controls on the data based on certain conditions (for example, based on the user that's making a call and whether the user owns the data) you can use mapping templates in your resolvers. You can also perform more complex business logic, which we describe in [Filtering Information \(p. 232\)](#).

This section shows how to set access controls on your data using a DynamoDB resolver mapping template.

Before proceeding any further, if you're not familiar with mapping templates in AWS AppSync, you may want to review the [Resolver Mapping Template Reference \(p. 264\)](#) and the [Resolver Mapping Template Reference for DynamoDB \(p. 321\)](#).

In the following example using DynamoDB, suppose you're using the preceding blog post schema, and only users that created a post are allowed to edit it. The evaluation process would be for the user to gain credentials in their application, using Amazon Cognito User Pools for example, and then pass these credentials as part of a GraphQL operation. The mapping template will then substitute a value from the credentials (like the username) in a conditional statement which will then be compared to a value in your database.



To add this functionality, add a GraphQL field of `editPost` as follows:

```
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!
}
```

...

The resolver mapping template for `editPost` (shown in an example at the end of this section) needs to perform a logical check against your data store to allow only the user that created a post to edit it. Since this is an edit operation, it corresponds to an `UpdateItem` in DynamoDB. You can perform a conditional check before performing this action, using context passed through for user identity validation. This is stored in an `Identity` object that has the following values:

```
{
  "accountId" : "12321434323",
  "cognitoIdentityPoolId" : "",
  "cognitoIdentityId" : "",
  "sourceIP" : "",
  "caller" : "ThisistheprincipalARN",
  "username" : "username",
  "userArn" : "Sameasabove"
}
```

To use this object in a `DynamoDBUpdateItem` call, you need to store the user identity information in the table for comparison. First, your `addPost` mutation needs to store the creator. Second, your `editPost` mutation needs to perform the conditional check before updating.

Here is an example of the request mapping template for `addPost` that stores the user identity as an `Author` column:

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "postId" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "attributeValues" : {
    "Author" : $util.dynamodb.toDynamoDBJson($context.identity.username)
    #foreach( $entry in $context.arguments.entrySet() )
      #if( $entry.key != "id" )
        , "{$entry.key}" : $util.dynamodb.toDynamoDBJson($entry.value)
      #end
    #end
  },
  "condition" : {
    "expression" : "attribute_not_exists(postId)"
  }
}
```

Note that the `Author` attribute is populated from the `Identity` object, which came from the application.

Finally, here is an example of the request mapping template for `editPost`, which only updates the content of the blog post if the request comes from the user that created the post:

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args),
  "condition" : {
    "expression" : "contains(#author, :expectedOwner)",
    "expressionNames" : {
```

```
        "#author" : "Author"
      },
      "expressionValues" : {
        ":expectedOwner" : $util.dynamodb.toDynamoDBJson($context.identity.username)
      }
    }
  }
}
```

This example uses a `PutItem` that overwrites all values rather than an `UpdateItem`, which would be a bit more verbose in an example, but the same concept applies on the condition statement block.

## Filtering Information

There may be cases where you cannot control the response from your data source, but you don't want to send unnecessary information to clients on a successful write or read to the data source. In these cases, you can filter information by using a response mapping template.

For example, suppose you don't have an appropriate index on your blog post DynamoDB table (such as an index on `Author`). You could run a `GetItem` query with the following mapping template:

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

This returns all the values responses, even if the caller isn't the author who created the post. To prevent this from happening, you can perform the access check on the response mapping template in this case as follows:

```
{
  #if($context.result["Author"] == "$context.identity.username")
    $utils.toJson($context.result);
  #end
}
```

If the caller doesn't match this check, only a null response is returned.

## Data source access

AWS AppSync communicates with data sources using Identity and Access Management ([IAM](#)) roles and access policies. If you are using an existing role, a Trust Policy needs to be added in order for AWS AppSync to assume the role. The trust relationship will look like below:

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



```
        "Action": "sts:AssumeRole"
      }
    ]
  }
}
```

It's important to scope down the access policy on the role to only have permissions to act on the minimal set of resources necessary. When using the AppSync console to create a data source and create a role, this is done automatically for you. However when using a built in sample template from the IAM console to create a role outside of the AWS AppSync console the permissions will not be automatically scoped down on a resource and you should perform this action before moving your application to production.

## Authorization Use Cases

In the [Security \(p. 221\)](#) section you learned about the different Authorization modes for protecting your API and an introduction was given on Fine Grained Authorization mechanisms to understand the concepts and flow. Since AWS AppSync allows you to perform logic full operations on data through the use of GraphQL Resolver [Mapping Templates \(p. 264\)](#), you can protect data on read or write in a very flexible manner using a combination of user identity, conditionals, and data injection.

If you're not familiar with editing AWS AppSync Resolvers, review the [programming guide \(p. 268\)](#).

### Overview

Granting access to data in a system is traditionally done through an [Access Control Matrix](#) where the intersection of a row (resource) and column (user/role) is the permissions granted.

AWS AppSync uses resources in your own account and threads identity (user/role) information into the GraphQL request and response as a [context object \(p. 279\)](#), which you can use in the resolver. This means that permissions can be granted appropriately either on write or read operations based on the resolver logic. If this logic is at the resource level, for example only certain named users or groups can read/write to a specific database row, then that "authorization metadata" must be stored. AWS AppSync does not store any data so therefore you must store this authorization metadata with the resources so that permissions can be calculated. Authorization metadata is usually an attribute (column) in a DynamoDB table, such as an **owner** or list of users/groups. For example there could be **Readers** and **Writers** attributes.

From a high level, what this means is that if you are reading an individual item from a data source, you perform a conditional `#if () ... #end` statement in the response template after the resolver has read from the data source. The check will normally be using user or group values in `$context.identity` for membership checks against the authorization metadata returned from a read operation. For multiple records, such as lists returned from a table `Scan` or `Query`, you'll send the condition check as part of the operation to the data source using similar user or group values.

Similarly when writing data you'll apply a conditional statement to the action (like a `PutItem` or `UpdateItem` to see if the user or group making a mutation has permission. The conditional again will many times be using a value in `$context.identity` to compare against authorization metadata on that resource. For both request and response templates you can also use custom headers from clients to perform validation checks.

### Reading Data

As outlined above the authorization metadata to perform a check must be stored with a resource or passed in to the GraphQL request (identity, header, etc.). To demonstrate this suppose you have the DynamoDB table below:

ID	Data	PeopleCanAccess	GroupsCanAccess	Owner
123	{my: data,...}	[Mary, Joe]	[Admins, Editors]	Nadia

The primary key is `id` and the data to be accessed is `Data`. The other columns are examples of checks you can perform for authorization. `Owner` would be a `String` while `PeopleCanAccess` and `GroupsCanAccess` would be `String Sets` as outlined in the [Resolver Mapping Template Reference for DynamoDB \(p. 321\)](#).

In the [resolver mapping template overview \(p. 264\)](#) the diagram shows how the response template contains not only the context object but also the results from the data source. For GraphQL queries of individual items, you can use the response template to check if the user is allowed to see these results or return an authorization error message. This is sometimes referred to as an “Authorization filter”. For GraphQL queries returning lists, using a `Scan` or `Query`, it is more performant to perform the check on the request template and return data only if an authorization condition is satisfied. The implementation is then:

1. `GetItem` - authorization check for individual records. Done using `#if() ... #end` statements.
2. `Scan/Query` operations - authorization check is a `"filter":{"expression":...}` statement. Common checks are equality (`attribute = :input`) or checking if a value is in a list (`contains(attribute, :input)`).

In #2 the `attribute` in both statements represents the column name of the record in a table, such as `Owner` in our above example. You can alias this with a `#` sign and use `"expressionNames":{"...}"` but it's not mandatory. The `:input` is a reference to the value you're comparing to the database attribute, which you will define in `"expressionValues":{"..."}`. You'll see these examples below.

## Use Case: Owner Can Read

Using the table above, if you only wanted to return data if `Owner == Nadia` for an individual read operation (`GetItem`) your template would look like:

```
#if($context.result["Owner"] == $context.identity.username)
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

A couple things to mention here which will be re-used in the remaining sections. First, the check uses `$context.identity.username` which will be the friendly user sign-up name if Amazon Cognito user pools is used and will be the user identity if IAM is used (including Amazon Cognito Federated Identities). There are other values to store for an owner such as the unique “Amazon Cognito identity” value, which is useful when federating logins from multiple locations, and you should review the options available in the [Resolver Mapping Template Context Reference \(p. 279\)](#).

Second, the conditional else check responding with `$util.unauthorized()` is completely optional but recommended as a best practice when designing your GraphQL API.

## Use Case: Hardcode Specific Access

```
// 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

```

## Use Case: Filtering a List of Results

In the previous example you were able to perform a check against `$context.result` directly as it returned a single item, however some operations like a scan will return multiple items in `$context.result.items` where you need to perform the authorization filter and only return results that the user is allowed to see. Suppose the `Owner` field had the Amazon Cognito IdentityID this time set on the record, you could then use the following response mapping template to filter to only show those records that the user owned:

```

#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)

```

## Use Case: Multiple People Can Read

Another popular authorization option is to allow a group of people to be able to read data. In the example below the `"filter":{"expression":...}` only returns values from a table scan if the user running the GraphQL query is listed in the set for `PeopleCanAccess`.

```

{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "limit": #if(${context.arguments.count}) $util.toJson($context.arguments.count) #else
20 #end,
    "nextToken": #if(${context.arguments.nextToken})
$util.toJson($context.arguments.nextToken) #else null #end,
    "filter":{
        "expression": "contains(#peopleCanAccess, :value)",
        "expressionNames": {
            "#peopleCanAccess": "peopleCanAccess"
        },
        "expressionValues": {
            ":value": $util.dynamodb.toDynamoDBJson($context.identity.username)
        }
    }
}

```

```
}
}
}
```

## Use Case: Group Can Read

Similar to the last use case, it may be that only people in one or more groups have rights to read certain items in a database. Use of the "expression": "contains()" operation is similar however it's a logical-OR of all the groups that a user might be a part of which needs to be accounted for in the set membership. In this case we build up a \$expression statement below for each group the user is in and then pass this to the filter:

```
#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)
    }
}
```

## Writing Data

Writing data on mutations is always controlled on the request mapping template. In the case of DynamoDB data sources, the key is to use an appropriate "condition":{"expression"...}" which performs validation against the authorization metadata in that table. In [Security \(p. 221\)](#), we provided an example you can use to check the Author field in a table. The use cases in this section explore more use cases.

## Use Case: Multiple Owners

Using the example table diagram from earlier, suppose the PeopleCanAccess list

```
{
    "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)
      }
    }
  }
}

```

## Use Case: Group Can Create New Record

```

#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
#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)
  }
}

```

## Use Case: Group Can Update Existing Record

```

#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
#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)
    }
}
```

## Public and Private Records

With the conditional filters you can also choose to mark data as private, public or some other Boolean check. This can then be combined as part of an authorization filter inside the response template. Using this check is a nice way to temporarily hide data or remove it from view without trying to control group membership.

For example suppose you added an attribute on each item in your DynamoDB table called `public` with either a value of `yes` or `no`. The following response template could be used on a `GetItem` call to only display data if the user is in a group that has access AND if that data is marked as public:

```
#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
```

The above code could also use a logical OR (`||`) to allow people to read if they have permission to a record or if it's public:

```
#if($hasPermission || $context.result.public == 'yes')
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

In general, you will find the standard operators `==`, `!=`, `&&`, and `||` helpful when performing authorization checks.

## Real-Time Data

You can apply Fine Grained Access Controls to GraphQL subscriptions at the time a client makes a subscription, using the same techniques described earlier in this documentation. You attach a resolver to the subscription field, at which point you can query data from a data source and perform conditional logic in either the request or response mapping template. You can also return additional data to the

client, such as the initial results from a subscription, as long as the data structure matches that of the returned type in your GraphQL subscription.

## Use Case: User Can Subscribe to Specific Conversations Only

A common use case for real-time data with GraphQL subscriptions is building a messaging or private chat application. When creating a chat application that has multiple users, conversations can occur between two people or among multiple people. These might be grouped into “rooms”, which are private or public. As such, you would only want to authorize a user to subscribe to a conversation (which could be one to one or among a group) for which they have been granted access. For demonstration purposes, the sample below shows a simple use case of one user sending a private message to another. The setup has two Amazon DynamoDB tables:

- Messages table: (primary key) toUser, (sort key) id
- Permissions table: (primary key) username

The Messages table stores the actual messages that get sent via a GraphQL mutation. The Permissions table is checked by the GraphQL subscription for authorization at client connection time. The example below assumes you are using the following GraphQL schema:

```
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  
}
```

Some of the standard operations, such as `createUserPermissions()`, are not covered below to illustrate the subscription resolvers, but are standard implementations of DynamoDB resolvers. Instead, we'll focus on subscription authorization flows with resolvers. To send a message from one user to another, attach a resolver to the `sendMessage()` field and select the **Messages** table data source with the following request template:

```
{  
  "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),  
  }  
}
```

In this example, we use `$context.identity.username`. This returns user information for AWS Identity and Access Management or Amazon Cognito users. The response template is a simple passthrough of `$util.toJson($ctx.result)`. Save and go back to the schema page. Then attach a resolver for the `newMessage()` subscription, using the **Permissions** table as a data source and the following request mapping template:

```
{  
  "version": "2018-05-29",  
  "operation": "GetItem",  
  "key": {  
    "username": $util.dynamodb.toDynamoDBJson($ctx.identity.username),  
  },  
}
```

Then use the following response mapping template to perform your authorization checks using data from the **Permissions** table:

```
#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
```

In this case, you're doing three authorization checks. The first ensures that a result is returned. The second ensures that the user isn't subscribing to messages that are meant for another person. The third ensures that the user is allowed to subscribe to any fields, by checking a DynamoDB attribute of `isAuthorizedForSubscriptions` stored as a `BOOL`.



To test things out, you could sign in to the AWS AppSync console using Amazon Cognito user pools and a user named “Nadia”, and then run the following GraphQL subscription:

```
subscription AuthorizedSubscription {  
  newMessage(toUser: "Nadia") {  
    id  
    toUser  
    fromUser  
    content  
  }  
}
```

If in the **Permissions** table there is a record for the username key attribute of Nadia with `isAuthorizedForSubscriptions` set to `true`, you'll see a successful response. If you try a different username in the `newMessage()` query above, an error will be returned.

## Using AWS WAF to protect your APIs

AWS WAF is a web application firewall that helps protect web applications and APIs from attacks. It allows you to configure a set of rules, called a web access control list (web ACL), that allow, block, or monitor (count) web requests based on customizable web security rules and conditions that you define. When you integrate your AWS AppSync API with AWS WAF, you gain more control and visibility into the HTTP traffic accepted by your API. To learn more about AWS WAF, see [How AWS WAF Works](#) in the AWS WAF Developer Guide.

You can use AWS WAF to protect your AppSync API from common web exploits, such as SQL injection and cross-site scripting (XSS) attacks. These could affect API availability and performance, compromise security, or consume excessive resources. For example, you can create rules to allow or block requests from specified IP address ranges, requests from CIDR blocks, requests that originate from a specific country or region, requests that contain malicious SQL code, or requests that contain malicious script.

You can also create rules that match a specified string or a regular expression pattern in HTTP headers, method, query string, URI, and the request body (limited to the first 8 KB). Additionally, you can create rules to block attacks from specific user agents, bad bots, and content scrapers. For example, you can use rate-based rules to specify the number of web requests that are allowed by each client IP in a trailing, continuously updated, 5-minute period.

To learn more about the types of rules that are supported and additional AWS WAF features, see the [AWS WAF Developer Guide](#) and the [AWS WAF API Reference](#).

### Important

AWS WAF is your first line of defense against web exploits. When AWS WAF is enabled on an API, AWS WAF rules are evaluated before other access control features, such as API key authorization, IAM policies, OIDC tokens, and Amazon Cognito user pools.

## Integrate an AppSync API with AWS WAF

You can integrate an Appsync API with AWS WAF using the AWS Management Console, the AWS CLI, AWS CloudFormation, or any other compatible client.

### To integrate an AWS AppSync API with AWS WAF

1. Create an AWS WAF web ACL. For detailed steps using the [AWS WAF Console](#), see [Creating a web ACL](#).
2. Define the rules for the web ACL. A rule or rules are defined in the process of creating the web ACL. For information about how to structure rules, see [AWS WAF rules](#). For examples of useful rules you can define for your AWS AppSync API, see [Creating rules for a web ACL \(p. 242\)](#).

3. Associate the web ACL with an AWS AppSync API. You can perform this step in the [AWS WAF Console](#) or in the [AppSync Console](#).
  - To associate the web ACL with an AWS AppSync API in the AWS WAF Console, follow the instructions for [Associating or disassociating a Web ACL with an AWS resource](#) in the AWS WAF Developer Guide.
  - To associate the web ACL with an AWS AppSync API in the AWS AppSync Console
    - a. Sign in to the AWS Management Console and open the [AppSync Console](#).
    - b. Choose the API that you want to associate with a web ACL.
    - c. In the navigation pane, choose **Settings**.
    - d. In the **Web application firewall** section, turn on **Enable AWS WAF**.
    - e. In the **Web ACL** dropdown list, choose the name of the web ACL to associate with your API.
    - f. Choose **Save** to associate the web ACL with your API.

#### Note

After you create a web ACL in the AWS WAF Console, it can take a few minutes for the new web ACL to be available. If you do not see a newly created web ACL in the **Web application firewall** menu, wait a few minutes and retry the steps to associate the web ACL with your API.

After you associate a web ACL with an AWS AppSync API, you will manage the web ACL using the AWS WAF APIs. You do not need to re-associate the web ACL with your AWS AppSync API unless you want to associate the AWS AppSync API with a different web ACL.

## Creating rules for a web ACL

Rules define how to inspect web requests and what to do when a web request matches the inspection criteria. Rules don't exist in AWS WAF on their own. You can access a rule by name in a rule group or in the web ACL where it's defined. For more information, see [AWS WAF rules](#). The following examples demonstrate how to define and associate rules that are useful for protecting an AppSync API.

### Example web ACL rule to limit request body size

The following is an example of a rule that limits the body size of requests. This would be entered into the **Rule JSON editor** when creating a web ACL in the AWS WAF Console.

```
{
  "Name": "BodySizeRule",
  "Priority": 1,
  "RuleAction": {
    "Block": {}
  },
  "Statement": {
    "SizeConstraintStatement": {
      "ComparisonOperator": "GE",
      "FieldToMatch": {
        "Body": {}
      },
      "Size": 1024,
      "TextTransformations": [
        {
          "Priority": 0,
          "Type": "NONE"
        }
      ]
    }
  }
},
```

```
"VisibilityConfig": {  
  "CloudWatchMetricsEnabled": true,  
  "MetricName": "BodySizeRule",  
  "SampledRequestsEnabled": true  
}
```

After you have created your web ACL using the preceding example rule, you must associate it with your AppSync API. As an alternative to using the AWS Management Console, you can perform this step in the AWS CLI by running the following command.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

It can take a few minutes for the changes to propagate, but after running this command, requests that contain a body larger than 1024 bytes will be rejected by AWS AppSync.

#### Note

After you create a new web ACL in the AWS WAF Console, it can take a few minutes for the web ACL to be available to associate with an API. If you run the CLI command and get a `WAFUnavailableEntityException` error, wait a few minutes and retry running the command.

### Example web ACL rule to limit requests from a single IP address

The following is an example of a rule that throttles an AppSync API to 100 requests from a single IP address in a 5 minute period. This would be entered into the **Rule JSON editor** when creating a web ACL with a rate-based rule in the AWS WAF Console.

```
{  
  "Name": "Throttle",  
  "Priority": 0,  
  "Action": {  
    "Block": {}  
  },  
  "VisibilityConfig": {  
    "SampledRequestsEnabled": true,  
    "CloudWatchMetricsEnabled": true,  
    "MetricName": "Throttle"  
  },  
  "Statement": {  
    "RateBasedStatement": {  
      "Limit": 100,  
      "AggregateKeyType": "IP"  
    }  
  }  
}
```

After you have created your web ACL using the preceding example rule, you must associate it with your AppSync API. You can perform this step in the AWS CLI by running the following command.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

### Example web ACL rule to prevent GraphQL `__schema` introspection queries to an API

The following is an example of a rule that prevents GraphQL `__schema` introspection queries to an API. Any HTTP body that includes the string `"__schema"` will be blocked. This would be entered into the **Rule JSON editor** when creating a web ACL in the AWS WAF Console.

```
{
```

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

After you have created your web ACL using the preceding example rule, you must associate it with your AppSync API. You can perform this step in the AWS CLI by running the following command.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

# Security in AWS AppSync

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The [shared responsibility model](#) describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the [AWS Compliance Programs](#). To learn about the compliance programs that apply to AWS AppSync, see [AWS Services in Scope by Compliance Program](#).
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using AWS AppSync. The following topics show you how to configure AWS AppSync to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your AWS AppSync resources.

## Topics

- [Data Protection in AWS AppSync](#) (p. 245)
- [Compliance Validation for AWS AppSync](#) (p. 246)
- [Infrastructure Security in AWS AppSync](#) (p. 247)
- [Resilience in AWS AppSync](#) (p. 247)
- [Identity and access management for AWS AppSync](#) (p. 247)
- [Logging AWS AppSync API calls with AWS CloudTrail](#) (p. 259)
- [Security best practices for AWS AppSync](#) (p. 106)

## Data Protection in AWS AppSync

The AWS [shared responsibility model](#) applies to data protection in AWS AppSync. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. This content includes the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the [Data Privacy FAQ](#). For information about data protection in Europe, see the [AWS Shared Responsibility Model and GDPR](#) blog post on the [AWS Security Blog](#).

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with AWS Identity and Access Management (IAM). That way each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We recommend TLS 1.2 or later.

- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see [Federal Information Processing Standard \(FIPS\) 140-2](#).

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form fields such as a **Name** field. This includes when you work with AWS AppSync or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

## Encryption in motion

AWS AppSync, like all AWS services, makes use of TLS1.2 and beyond for communication when using the AWS published APIs and SDKs.

Using AWS AppSync with other AWS services such as Amazon DynamoDB ensures encryption in transit: All AWS services use TLS 1.2 and beyond to communicate with one another unless otherwise specified. For resolvers that utilize Amazon EC2 or CloudFront, it is your responsibility to verify that TLS (HTTPS) is configured and secure. For information on configuring HTTPS in Amazon EC2, see [Configuring SSL/TLS on Amazon Linux 2](#) in the Amazon EC2 user guide. For information about configuring HTTPS on CloudFront, see [HTTPS in Amazon CloudFront](#) in the CloudFront user guide.

## Compliance Validation for AWS AppSync

Third-party auditors assess the security and compliance of AWS AppSync as part of multiple AWS compliance programs. AWS AppSync is compliant with SOC, PCI, HIPAA/HIPAA BAA, IRAP, C5, ENS High, OSPAR, and HITRUST CSF programs.

To learn whether AWS AppSync or other AWS services are in scope of specific compliance programs, see [AWS Services in Scope by Compliance Program](#). For general information, see [AWS Compliance Programs](#).

You can download third-party audit reports using AWS Artifact. For more information, see [Downloading Reports in AWS Artifact](#).

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- [Security and Compliance Quick Start Guides](#) – These deployment guides discuss architectural considerations and provide steps for deploying baseline environments on AWS that are security and compliance focused.
- [Architecting for HIPAA Security and Compliance Whitepaper](#) – This whitepaper describes how companies can use AWS to create HIPAA-eligible applications.

### Note

Not all AWS services are HIPAA eligible. For more information, see the [HIPAA Eligible Services Reference](#).

- [AWS Compliance Resources](#) – This collection of workbooks and guides might apply to your industry and location.

- [Evaluating Resources with Rules](#) in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- [AWS Security Hub](#) – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.
- [AWS Audit Manager](#) – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

## Infrastructure Security in AWS AppSync

As a managed service, AWS AppSync is protected by the AWS global network security procedures that are described in the [Amazon Web Services: Overview of Security Processes](#) whitepaper.

You use AWS published API calls to access AWS AppSync through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the [AWS Security Token Service](#) (AWS STS) to generate temporary security credentials to sign requests.

## Resilience in AWS AppSync

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see [AWS Global Infrastructure](#).

In addition to the AWS global infrastructure, AWS AppSync allows most resources to be defined using AWS CloudFormation templates; for an example of using AWS CloudFormation templates to declare AWS AppSync resources, see [Practical use cases for AWS AppSync Pipeline Resolvers](#) on the AWS blog and the [AWS CloudFormation User Guide](#).

## Identity and access management for AWS AppSync

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use AWS AppSync resources. IAM is an AWS service that you can use with no additional charge.

### Topics

- [Audience](#) (p. 248)
- [Authenticating with identities](#) (p. 248)
- [Managing access using policies](#) (p. 250)

- [How AWS AppSync works with IAM \(p. 251\)](#)
- [Identity-based policies for AWS AppSync \(p. 256\)](#)
- [Troubleshooting AWS AppSync identity and access \(p. 257\)](#)

## Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in AWS AppSync.

**Service user** – If you use the AWS AppSync service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more AWS AppSync features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in AWS AppSync, see [Troubleshooting AWS AppSync identity and access \(p. 257\)](#).

**Service administrator** – If you're in charge of AWS AppSync resources at your company, you probably have full access to AWS AppSync. It's your job to determine which AWS AppSync features and resources your employees should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with AWS AppSync, see [How AWS AppSync works with IAM \(p. 251\)](#).

**IAM administrator** – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to AWS AppSync. To view example AWS AppSync identity-based policies that you can use in IAM, see [Identity-based policies for AWS AppSync \(p. 256\)](#).

## Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see [Signing in to the AWS Management Console as an IAM user or root user](#) in the *IAM User Guide*.

You must be *authenticated* (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the [AWS Management Console](#), use your password with your root user email address or your IAM user name. You can access AWS programmatically using your root user or IAM users access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don't use AWS tools, you must sign the request yourself. Do this using *Signature Version 4*, a protocol for authenticating inbound API requests. For more information about authenticating requests, see [Signature Version 4 signing process](#) in the *AWS General Reference*.

Regardless of the authentication method that you use, you might also be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see [Using multi-factor authentication \(MFA\) in AWS](#) in the *IAM User Guide*.

## AWS account root user

When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the [best practice of using the root user only to create your first IAM user](#). Then



securely lock away the root user credentials and use them to perform only a few account and service management tasks.

## IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see [Managing access keys for IAM users](#) in the *IAM User Guide*. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see [When to create an IAM user \(instead of a role\)](#) in the *IAM User Guide*.

## IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by [switching roles](#). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see [Using IAM roles](#) in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an [identity provider](#). For more information about federated users, see [Federated users and roles](#) in the *IAM User Guide*.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see [How IAM roles differ from resource-based policies](#) in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
- **Principal permissions** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform both actions. To see whether an action requires additional dependent actions in a policy, see [Actions, resources, and condition keys for AWS AppSync](#) in the *Service Authorization Reference*.
- **Service role** – A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Creating a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.

- **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see [Using an IAM role to grant permissions to applications running on Amazon EC2 instances](#) in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see [When to create an IAM role \(instead of a user\)](#) in the *IAM User Guide*.

## Managing access using policies

You control access in AWS by creating policies and attaching them to IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. You can sign in as the root user or an IAM user, or you can assume an IAM role. When you then make a request, AWS evaluates the related identity-based or resource-based policies. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see [Overview of JSON policies](#) in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

## Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Creating IAM policies](#) in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see [Choosing between managed policies and inline policies](#) in the *IAM User Guide*.

## Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-

based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#) in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

## Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see [Access control list \(ACL\) overview](#) in the *Amazon Simple Storage Service Developer Guide*.

## Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see [Permissions boundaries for IAM entities](#) in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see [How SCPs work](#) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see [Session policies](#) in the *IAM User Guide*.

## Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see [Policy evaluation logic](#) in the *IAM User Guide*.

## How AWS AppSync works with IAM

Before you use IAM to manage access to AWS AppSync, learn what IAM features are available to use with AWS AppSync.

### **IAM features that you can use with AWS AppSync**

<b>IAM feature</b>	<b>AWS AppSync support</b>
<a href="#">Identity-based policies (p. 252)</a>	Yes
<a href="#">Resource-based policies (p. 252)</a>	No
<a href="#">Policy actions (p. 253)</a>	Yes
<a href="#">Policy resources (p. 253)</a>	Yes
<a href="#">Policy condition keys (p. 254)</a>	No
<a href="#">ACLs (p. 254)</a>	No
<a href="#">ABAC (tags in policies) (p. 255)</a>	Partial
<a href="#">Temporary credentials (p. 255)</a>	Yes
<a href="#">Principal permissions (p. 255)</a>	Partial
<a href="#">Service roles (p. 256)</a>	No
<a href="#">Service-linked roles (p. 256)</a>	Partial

To get a high-level view of how AWS AppSync and other AWS services work with most IAM features, see [AWS services that work with IAM](#) in the *IAM User Guide*.

## **Identity-based policies for AWS AppSync**

Supports identity-based policies	Yes
----------------------------------	-----

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Creating IAM policies](#) in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see [IAM JSON policy elements reference](#) in the *IAM User Guide*.

### **Identity-based policy examples for AWS AppSync**

To view examples of AWS AppSync identity-based policies, see [Identity-based policies for AWS AppSync \(p. 256\)](#).

## **Resource-based policies within AWS AppSync**

Supports resource-based policies	No
----------------------------------	----

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-

based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#) in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see [How IAM roles differ from resource-based policies](#) in the *IAM User Guide*.

## Policy actions for AWS AppSync

Supports policy actions	Yes
-------------------------	-----

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The **Action** element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of AWS AppSync actions, see [Actions defined by AWS AppSync](#) in the *Service Authorization Reference*.

Policy actions in AWS AppSync use the following prefix before the action:

```
appsync
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [  
    "appsync:action1",  
    "appsync:action2"  
]
```

To view examples of AWS AppSync identity-based policies, see [Identity-based policies for AWS AppSync](#) (p. 256).

## Policy resources for AWS AppSync

Supports policy resources	Yes
---------------------------	-----

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The Resource JSON policy element specifies the object or objects to which the action applies. Statements must include either a Resource or a NotResource element. As a best practice, specify a resource using its [Amazon Resource Name \(ARN\)](#). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (\*) to indicate that the statement applies to all resources.

```
"Resource": "*"

```

To see a list of AWS AppSync resource types and their ARNs, see [Resources defined by AWS AppSync](#) in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see [Actions defined by AWS AppSync](#).

To view examples of AWS AppSync identity-based policies, see [Identity-based policies for AWS AppSync](#) (p. 256).

## Policy condition keys for AWS AppSync

Supports service-specific policy condition keys	No
---	----

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The Condition element (or Condition *block*) lets you specify conditions in which a statement is in effect. The Condition element is optional. You can create conditional expressions that use [condition operators](#), such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple Condition elements in a statement, or multiple keys in a single Condition element, AWS evaluates them using a logical AND operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical OR operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see [IAM policy elements: variables and tags](#) in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see [AWS global condition context keys](#) in the *IAM User Guide*.

To see a list of AWS AppSync condition keys, see [Condition keys for AWS AppSync](#) in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see [Actions defined by AWS AppSync](#).

To view examples of AWS AppSync identity-based policies, see [Identity-based policies for AWS AppSync](#) (p. 256).

## Access control lists (ACLs) in AWS AppSync

Supports ACLs	No
---------------	----

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

## Attribute-based access control (ABAC) with AWS AppSync

Supports ABAC (tags in policies)	Partial
----------------------------------	---------

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the [condition element](#) of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

For more information about ABAC, see [What is ABAC?](#) in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see [Use attribute-based access control \(ABAC\)](#) in the *IAM User Guide*.

## Using temporary credentials with AWS AppSync

Supports temporary credentials	Yes
--------------------------------	-----

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see [AWS services that work with IAM](#) in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see [Switching to a role \(console\)](#) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see [Temporary security credentials in IAM](#).

## Cross-service principal permissions for AWS AppSync

Supports principal permissions	Partial
--------------------------------	---------

When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform



both actions. To see whether an action requires additional dependent actions in a policy, see [Actions, resources, and condition keys for AWS AppSync](#) in the *Service Authorization Reference*.

## Service roles for AWS AppSync

Supports service roles	No
------------------------	----

A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Creating a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.

### Warning

Changing the permissions for a service role might break AWS AppSync functionality. Edit service roles only when AWS AppSync provides guidance to do so.

## Service-linked roles for AWS AppSync

Supports service-linked roles	Partial
-------------------------------	---------

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see [AWS services that work with IAM](#) in the *IAM User Guide*. Find a service in the table that includes a **Yes** in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

## Identity-based policies for AWS AppSync

By default, IAM users and roles don't have permission to create or modify AWS AppSync resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform actions on the resources that they need. The administrator must then attach those policies to the IAM users or groups that require those permissions.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see [Creating IAM policies](#) in the *IAM User Guide*.

To learn the best practices for creating and configuring IAM identity-based policies, see [the section called "IAM policy best practices" \(p. 262\)](#).

### Topics

- [Using the AWS AppSync console \(p. 256\)](#)
- [Allow users to view their own permissions \(p. 257\)](#)

## Using the AWS AppSync console

To access the AWS AppSync console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the AWS AppSync resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (IAM users or roles) with that policy.



You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that you're trying to perform.

To ensure that IAM users and roles can still use the AWS AppSync console, also attach the AWS AppSync ConsoleAccess or ReadOnly AWS managed policy to the entities. For more information, see [Adding permissions to a user](#) in the *IAM User Guide*.

## Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
  "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": "*"
    }
  ]
}
```

## Troubleshooting AWS AppSync identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with AWS AppSync and IAM.

### Topics

- [I am not authorized to perform an action in AWS AppSync \(p. 258\)](#)
- [I am not authorized to perform iam:PassRole \(p. 258\)](#)
- [I want to view my access keys \(p. 258\)](#)

- [I'm an administrator and want to allow others to access AWS AppSync \(p. 259\)](#)
- [I want to allow people outside of my AWS account to access my AWS AppSync resources \(p. 259\)](#)

## I am not authorized to perform an action in AWS AppSync

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password.

The following example error occurs when the IAM user `mateojackson` tries to use the console to view details about a fictional `my-example-widget` resource, but he does not have the fictional `appsync:GetWidget` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
appsync:GetWidget on resource: my-example-widget
```

In this case, Mateo asks his administrator to update his policies to allow him to access the `my-example-widget` resource using the `appsync:GetWidget` action.

## I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password. Ask that person to update your policies to allow you to pass a role to AWS AppSync.

Some AWS services allow you to pass an existing role to that service, instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in AWS AppSync. However, the action requires the service to have permissions granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform: iam:PassRole
```

In this case, Mary asks her administrator to update her policies to allow her to perform the `iam:PassRole` action.

## I want to view my access keys

After you create your IAM user access keys, you can view your access key ID at any time. However, you can't view your secret access key again. If you lose your secret key, you must create a new access key pair.

Access keys consist of two parts: an access key ID (for example, `AKIAIOSFODNN7EXAMPLE`) and a secret access key (for example, `wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY`). Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests. Manage your access keys as securely as you do your user name and password.

### Important

Do not provide your access keys to a third party, even to help [find your canonical user ID](#). By doing this, you might give someone permanent access to your account.

When you create an access key pair, you are prompted to save the access key ID and secret access key in a secure location. The secret access key is available only at the time you create it. If you lose your secret access key, you must add new access keys to your IAM user. You can have a maximum of two access keys.

If you already have two, you must delete one key pair before creating a new one. To view instructions, see [Managing access keys](#) in the *IAM User Guide*.

## I'm an administrator and want to allow others to access AWS AppSync

To allow others to access AWS AppSync, you must create an IAM entity (user or role) for the person or application that needs access. They will use the credentials for that entity to access AWS. You must then attach a policy to the entity that grants them the correct permissions in AWS AppSync.

To get started right away, see [Creating your first IAM delegated user and group](#) in the *IAM User Guide*.

## I want to allow people outside of my AWS account to access my AWS AppSync resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether AWS AppSync supports these features, see [How AWS AppSync works with IAM](#) (p. 251).
- To learn how to provide access to your resources across AWS accounts that you own, see [Providing access to an IAM user in another AWS account that you own](#) in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see [Providing access to AWS accounts owned by third parties](#) in the *IAM User Guide*.
- To learn how to provide access through identity federation, see [Providing access to externally authenticated users \(identity federation\)](#) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see [How IAM roles differ from resource-based policies](#) in the *IAM User Guide*.

## Logging AWS AppSync API calls with AWS CloudTrail

AWS AppSync is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in AWS AppSync. CloudTrail captures API calls for AWS AppSync as events. The calls captured include calls from the AWS AppSync console and code calls to the AWS AppSync API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for AWS AppSync. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to AWS AppSync, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the [AWS CloudTrail User Guide](#).

## AWS AppSync information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in AWS AppSync, that activity is recorded in a CloudTrail event along with other AWS service events in **Event**

**history.** You can view, search, and download recent events in your AWS account. For more information, see [Viewing events with CloudTrail Event history](#).

For an ongoing record of events in your AWS account, including events for AWS AppSync, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- [Overview for creating a trail](#)
- [CloudTrail supported services and integrations](#)
- [Configuring Amazon SNS notifications for CloudTrail](#)
- [Receiving CloudTrail log files from multiple regions](#) and [Receiving CloudTrail log files from multiple accounts](#)

AWS AppSync supports logging of calls made through the AWS AppSync API. At this time, calls to your APIs, as well as calls made to resolvers are not logged by AWS AppSync into CloudTrail.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the [CloudTrail userIdentity element](#).

## Understanding AWS AppSync log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the `GetGraphQLApi` action made through the AWS AppSync console:

```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "ABCDEFXAMPLEPRINCIPAL:nikkiwolf",
    "arn": "arn:aws:sts::111122223333:assumed-role/admin/nikkiwolf",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AIDAJ45Q7YFFAREXAMPLE",
        "arn": "arn:aws:iam::111122223333:role/admin",
        "accountId": "111122223333",
        "userName": "admin"
      },
      "webIdFederationData": {},
    }
  },
  "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.metall1.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": "11112223333"
    }
  }
```

The following example shows a CloudTrail log entry that demonstrates the `CreateApiKey` action made through the AWS CLI:

```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "ABCDEFXAMPLEPRINCIPAL",
    "arn": "arn:aws:iam::11112223333:user/nikkiwolf",
    "accountId": "11112223333",
    "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": "11112223333"
}
```

# Security best practices for AWS AppSync

Securing AWS AppSync is more than simply turning on a few levers or setting up logging. The following sections discuss security best practices that vary depending on how you use the service.

## Topics

- [Understand authentication methods \(p. 262\)](#)
- [Use TLS for HTTP resolvers \(p. 262\)](#)
- [Use roles with the least permissions possible \(p. 262\)](#)
- [IAM policy best practices \(p. 262\)](#)

## Understand authentication methods

AWS AppSync provides multiple ways to authenticate your users to your GraphQL APIs. Each method has trade-offs in security, auditability, and usability.

The following common authentication methods are available:

- Amazon Cognito user pools allow your GraphQL API to use user attributes for fine-grained access control and filtering.
- API tokens have a limited lifetime and are appropriate for automated systems, such as Continuous Integration systems and integration with external APIs.
- AWS Identity and Access Management (IAM) is appropriate for internal applications managed in your AWS account.
- OpenID Connect allows you to control and federate access with the OpenID Connect protocol.

For more information on authentication and authorization in AWS AppSync, see [Authorization and Authentication \(p. 221\)](#).

## Use TLS for HTTP resolvers

When using HTTP resolvers, make sure to use TLS-secured (HTTPS) connections wherever possible. For a full list of the TLS certificates that AWS AppSync trusts, see [Certificate Authorities \(CA\) Recognized by AWS AppSync for HTTPS Endpoints \(p. 387\)](#).

## Use roles with the least permissions possible

When using resolvers such as the [DynamoDB resolver \(p. 51\)](#), use roles that provide the most restrictive view to your resources, such as your Amazon DynamoDB tables.

## IAM policy best practices

Identity-based policies are very powerful. They determine whether someone can create, access, or delete AWS AppSync resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started using AWS managed policies** – To start using AWS AppSync quickly, use AWS managed policies to give your employees the permissions they need. These policies are already available in your account and are maintained and updated by AWS. For more information, see [Get started using permissions with AWS managed policies](#) in the *IAM User Guide*.

- **Grant least privilege** – When you create custom policies, grant only the permissions required to perform a task. Start with a minimum set of permissions and grant additional permissions as necessary. Doing so is more secure than starting with permissions that are too lenient and then trying to tighten them later. For more information, see [Grant least privilege](#) in the *IAM User Guide*.
- **Enable MFA for sensitive operations** – For extra security, require IAM users to use multi-factor authentication (MFA) to access sensitive resources or API operations. For more information, see [Using multi-factor authentication \(MFA\) in AWS](#) in the *IAM User Guide*.
- **Use policy conditions for extra security** – To the extent that it's practical, define the conditions under which your identity-based policies allow access to a resource. For example, you can write conditions to specify a range of allowable IP addresses that a request must come from. You can also write conditions to allow requests only within a specified date or time range, or to require the use of SSL or MFA. For more information, see [IAM JSON policy elements: Condition](#) in the *IAM User Guide*.

# Resolver Mapping Template Reference

## Topics

- [Resolver mapping template overview \(p. 264\)](#)
- [Resolver Mapping Template Programming Guide \(p. 268\)](#)
- [Resolver mapping template context reference \(p. 279\)](#)
- [Resolver mapping template utility reference \(p. 286\)](#)
- [Resolver Mapping Template Reference for DynamoDB \(p. 321\)](#)
- [Resolver mapping template reference for RDS \(p. 370\)](#)
- [Resolver Mapping Template Reference for OpenSearch \(p. 372\)](#)
- [Resolver mapping template reference for Lambda \(p. 375\)](#)
- [Resolver mapping template reference for None data source \(p. 382\)](#)
- [Resolver Mapping Template Reference for HTTP \(p. 384\)](#)
- [Resolver mapping template changelog \(p. 416\)](#)

## Resolver mapping template overview

AWS AppSync lets you respond to GraphQL requests by performing operations on your resources. For each GraphQL field you wish to run a query or mutation on, a resolver must be attached in order to communicate with a data source. The communication is typically through parameters or operations that are unique to the data source.

Resolvers are the connectors between GraphQL and a data source. They tell AWS AppSync how to translate an incoming GraphQL request into instructions for your backend data source, and how to translate the response from that data source back into a GraphQL response. They are written in [Apache Velocity Template Language \(VTL\)](#), which takes your request as input and outputs a JSON document containing the instructions for the resolver. You can use mapping templates for simple instructions, such as passing in arguments from GraphQL fields, or for more complex instructions, such as looping through arguments to build an item before inserting the item into DynamoDB.

There are two types of resolvers in AppSync which leverage mapping templates in slightly different ways: unit resolvers and pipeline resolvers.

## Unit Resolvers

Unit Resolvers are self contained entities which include a request and response template only. Use these for simple, single operations such as listing items from a single data source.

- **Request templates:** Take the incoming request after a GraphQL operation is parsed and convert it into a request configuration for the selected data source operation.
- **Response templates:** Interpret responses from your data source and map it to the shape of the GraphQL field output type.

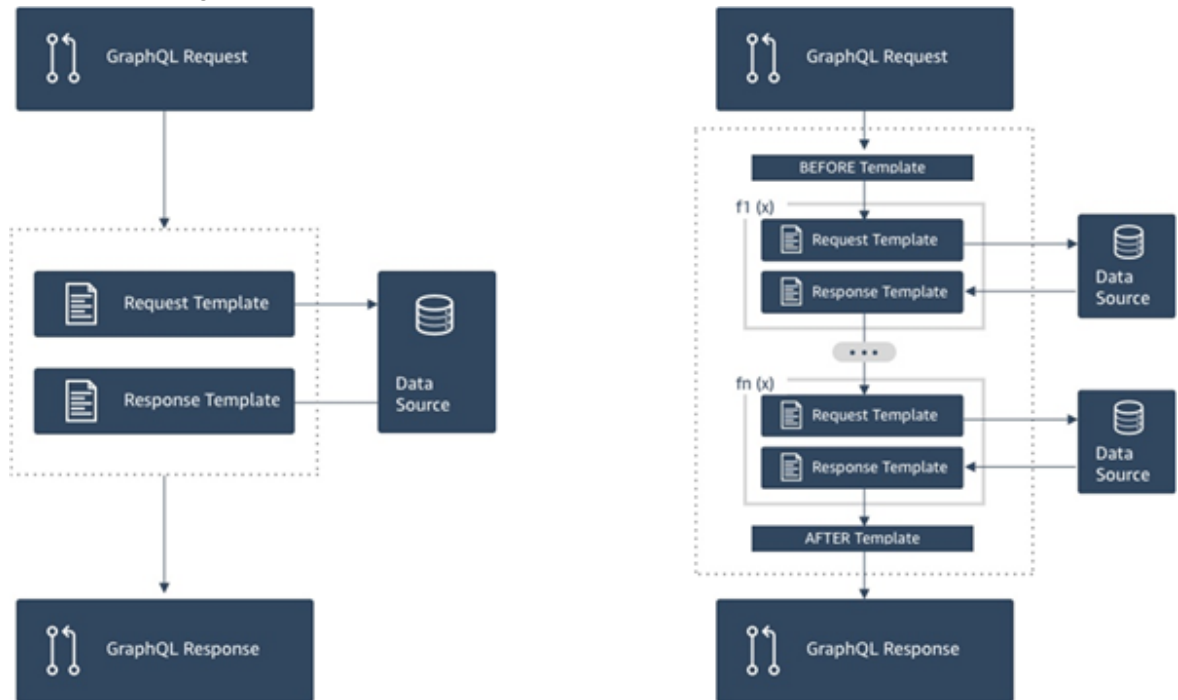


## Pipeline resolvers

Pipeline resolvers contain one or more *functions* which are performed in sequential order. Each function includes a request template and response template. A pipeline resolver also has a *before* template and an *after* template that surround the sequence of functions that the template contains. The *after* template maps to the GraphQL field output type. Pipeline resolvers differ from unit resolvers in the way that the response template maps output. A pipeline resolver can map to any output you want, including the input for another function or the *after* template of the pipeline resolver.

Pipeline resolver *functions* enable you to write common logic that you can reuse across multiple resolvers in your schema. Functions are attached directly to a data source, and like a unit resolver, contain the same request and response mapping template format.

The following diagram demonstrates the process flow of a unit resolver on the left and a pipeline resolver on the right.



Pipeline resolvers contain a superset of the functionality that unit resolvers support, and more, at the cost of a little more complexity.

## Example template

For example, suppose you have a DynamoDB data source and a **Unit** resolver on a field named `getPost(id:ID!)` that returns a `Post` type with the following GraphQL query:

```
getPost(id:1){
  id
  title
  content
}
```

Your resolver template might look like the following:

```
{
```

```
"version" : "2018-05-29",
"operation" : "GetItem",
"key" : {
  "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
}
}
```

This would substitute the `id` input parameter value of 1 for `${ctx.args.id}` and generate the following JSON:

```
{
  "version" : "2018-05-29",
  "operation" : "GetItem",
  "key" : {
    "id" : { "S" : "1" }
  }
}
```

AWS AppSync uses this template to generate instructions for communicating with DynamoDB and getting data (or performing other operations as appropriate). After the data returns, AWS AppSync runs it through an optional response mapping template, which you can use to perform data shaping or logic. For example, when we get the results back from DynamoDB, they might look like this:

```
{
  "id" : 1,
  "theTitle" : "AWS AppSync works offline!",
  "theContent-part1" : "It also has realtime functionality",
  "theContent-part2" : "using GraphQL"
}
```

You could choose to join two of the fields into a single field with the following response mapping template:

```
{
  "id" : $util.toJson($context.data.id),
  "title" : $util.toJson($context.data.theTitle),
  "content" : $util.toJson("${context.data.theContent-part1}
${context.data.theContent-part2}")
}
```

Here's how the data is shaped after the template is applied to the data:

```
{
  "id" : 1,
  "title" : "AWS AppSync works offline!",
  "content" : "It also has realtime functionality using GraphQL"
}
```

This data is given back as the response to a client as follows:

```
{
  "data": {
    "getPost": {
      "id" : 1,
      "title" : "AWS AppSync works offline!",
      "content" : "It also has realtime functionality using GraphQL"
    }
  }
}
```

Note that under most circumstances, response mapping templates are a simple passthrough of data, differing mostly if you are returning an individual item or a list of items. For an individual item the passthrough is:

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

For lists the passthrough is usually:

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

To see more examples of both unit and pipeline resolvers, see [Resolver tutorials \(p. 51\)](#).

## Evaluated mapping template deserialization rules

Mapping templates evaluate to a string. In AWS AppSync, the output string must follow a JSON structure to be valid.

Additionally, the following deserialization rules are enforced.

### Duplicate keys are not allowed in JSON objects

If the evaluated mapping template string represents a JSON object or contains an object that has duplicate keys, the mapping template returns the following error message:

```
Duplicate field 'aField' detected on Object. Duplicate JSON keys are not allowed.
```

Example of a duplicate key in an evaluated request mapping template:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "postId": "1",
    "field": "getPost" ## key 'field' has been redefined
  }
}
```

To fix this error, do not redefine keys in JSON objects.

### Trailing characters are not allowed in JSON objects

If the evaluated mapping template string represents a JSON object and contains trailing extraneous characters, the mapping template returns the following error message:

```
Trailing characters at the end of the JSON string are not allowed.
```

Example of trailing characters in an evaluated request mapping template:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "postId": "1",
  }
}
```

```
}extraneouschars
```

To fix this error, ensure evaluated templates strictly evaluate to JSON.

## Resolver Mapping Template Programming Guide

This is a cookbook-style tutorial of programming with the Apache Velocity Template Language (VTL) in AWS AppSync. If you are familiar with other programming languages such as JavaScript, C, or Java, it should be fairly straightforward.

AWS AppSync uses VTL to translate GraphQL requests from clients into a request to your data source. Then it reverses the process to translate the data source response back into a GraphQL response. VTL is a logical template language that gives you the power to manipulate both the request and the response in the standard request/response flow of a web application, using techniques such as:

- Default values for new items
- Input validation and formatting
- Transforming and shaping data
- Iterating over lists, maps, and arrays to pluck out or alter values
- Filter/change responses based on user identity
- Complex authorization checks

For example, you might want to perform a phone number validation in the service on a GraphQL argument, or convert an input parameter to upper case before storing it in DynamoDB. Or maybe you want client systems to provide a code, as part of a GraphQL argument, JWT token claim, or HTTP header, and only respond with data if the code matches a specific string in a list. These are all logical checks you can perform with VTL in AWS AppSync.

VTL allows you to apply logic using programming techniques that might be familiar. However, it is bounded to run within the standard request/response flow to ensure that your GraphQL API is scalable as your user base grows. Because AWS AppSync also supports AWS Lambda as a resolver, you can write Lambda functions in your programming language of choice (Node.js, Python, Go, Java, etc.) if you need more flexibility.

### Setup

A common technique when learning a language is to print out results (for example, `console.log(variable)` in JavaScript) to see what happens. In this tutorial, we demonstrate this by creating a simple GraphQL schema and passing a map of values to a Lambda function. The Lambda function prints out the values and then responds with them. This will enable you to understand the request/response flow and see different programming techniques.

Start by creating the following GraphQL schema:

```
type Query {
  get(id: ID, meta: String): Thing
}

type Thing {
  id: ID!
  title: String!
  meta: String
}

schema {
  query: Query
}
```

```
    query: Query
  }
```

Now create the following AWS Lambda function, using Node.js as the language:

```
exports.handler = (event, context, callback) => {
  console.log('VTL details: ', event);
  callback(null, event);
};
```

In the **Data Sources** pane of the AWS AppSync console, add this Lambda function as a new data source. Navigate back to the **Schema** page of the console and click the **ATTACH** button on the right, next to the `get(...):Thing` query. For the request template, choose the existing template from the **Invoke and forward arguments** menu. For the response template, choose **Return Lambda result**.

Open Amazon CloudWatch Logs for your Lambda function in one location, and from the **Queries** tab of the AWS AppSync console, run the following GraphQL query:

```
query test {
  get(id:123 meta:"testing"){
    id
    meta
  }
}
```

The GraphQL response should contain `id:123` and `meta:testing`, because the Lambda function is echoing them back. After a few seconds, you should see a record in CloudWatch Logs with these details as well.

## Variables

VTL uses [references](#), which you can use to store or manipulate data. There are three types of references in VTL: variables, properties, and methods. Variables have a `$` sign in front of them and are created with the `#set` directive:

```
#set($var = "a string")
```

Variables store similar types that you're familiar with from other languages, such as numbers, strings, arrays, lists, and maps. You might have noticed a JSON payload being sent in the default request template for Lambda resolvers:

```
"payload": $util.toJson($context.arguments)
```

A couple of things to notice here - first, AWS AppSync provides several convenience functions for common operations. In this example, `$util.toJson` converts a variable to JSON. Second, the variable `$context.arguments` is automatically populated from a GraphQL request as a map object. You can create a new map as follows:

```
#set( $myMap = {
  "id": $context.arguments.id,
  "meta": "stuff",
  "upperMeta" : $context.arguments.meta.toUpperCase()
} )
```

You have now created a variable named `$myMap`, which has keys of `id`, `meta`, and `upperMeta`. This also demonstrates a few things:

- `id` is populated with a key from the GraphQL arguments. This is common in VTL to grab arguments from clients.
- `meta` is hardcoded with a value, showcasing default values.
- `upperMeta` is transforming the `meta` argument using a method `.toUpperCase()`.

Put the previous code at the top of your request template and change the `payload` to use the new `$myMap` variable:

```
"payload": $util.toJson($myMap)
```

Run your Lambda function, and you can see the response change as well as this data in CloudWatch logs. As you walk through the rest of this tutorial, we will keep populating `$myMap` so you can run similar tests.

You can also set *properties\_* on your variables. These could be simple strings, arrays, or JSON:

```
#set($myMap.myProperty = "ABC")
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])
#set($myMap.jsonProperty = {
  "AppSync" : "Offline and Realtime",
  "Cognito" : "AuthN and AuthZ"
})
```

## Quiet References

Because VTL is a templating language, by default, every reference you give it will do a `.toString()`. If the reference is undefined, it prints the actual reference representation, as a string. For example:

```
#set($myValue = 5)
##Prints '5'
$myValue

##Prints '$somethingelse'
$somethingelse
```

To address this, VTL has a *quiet reference* or *silent reference* syntax, which tells the template engine to suppress this behavior. The syntax for this is `#{ }`. For example, if we changed the previous code slightly to use `#{somethingelse}`, the printing is suppressed:

```
#set($myValue = 5)
##Prints '5'
$myValue

##Nothing prints out
#{somethingelse}
```

## Calling Methods

In an earlier example, we showed you how to create a variable and simultaneously set values. You can also do this in two steps by adding data to your map as shown following:

```
#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")}

```

**HOWEVER** there is something to know about this behavior. Although the quiet reference notation `#!{ }` allows you to call methods, as above, it won't suppress the returned value of the executed method. This is why we noted `##Prints "first value"` and `##Prints true` above. This can cause errors when you're iterating over maps or lists, such as inserting a value where a key already exists, because the output adds unexpected strings to the template upon evaluation.

The workaround to this is sometimes to call the methods using a `#set` directive and ignore the variable. For example:

```

#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))

```

You might use this technique in your templates, as it prevents the unexpected strings from being printed in the template. AWS AppSync provides an alternative convenience function that offers the same behavior in a more succinct notation. This enables you to not have to think about these implementation specifics. You can access this function under `$util.quiet()` or its alias `$util.qr()`. For example:

```

#set ($myMap = {})
#set ($myList = [])

##Nothing prints out
$util.quiet($myMap.put("id", "first value"))
##Nothing prints out
$util.qr($myList.add("something"))

```

## Strings

As with many programming languages, strings can be difficult to deal with, especially when you want to build them from variables. There are some common things that come up with VTL.

Suppose you are inserting data as a string to a data source like DynamoDB, but it is populated from a variable, like a GraphQL argument. A string will have double quotation marks, and to reference the variable in a string you just need `"${ }"` (so no `#!` as in [quiet reference notation](#)). This is similar to a template literal in JavaScript: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template\\_literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals)

```

#set($firstname = "Jeff")
${myMap.put("Firstname", "${firstname}")}

```

You can see this in DynamoDB request templates, like `"author": { "S" : "${context.arguments.author}" }` when using arguments from GraphQL clients, or for automatic ID generation like `"id" : { "S" : "${util.autoId()}" }`. This means that you can reference a variable or the result of a method inside a string to populate data.

You can also use public methods of the Java [String class](#), such as pulling out a substring:

```

#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}"))
```

String concatenation is also a very common task. You can do this with variable references alone or with static values:

```
#set($s1 = "Hello")
#set($s2 = " World")

$util.qr($myMap.put("concat", "$s1$s2"))
$util.qr($myMap.put("concat2", "Second $s1 World"))
```

## Loops

Now that you have created variables and called methods, you can add some logic to your code. Unlike other languages, VTL allows only loops, where the number of iterations is predetermined. There is no `do..while` in Velocity. This design ensures that the evaluation process always terminates, and provides bounds for scalability when your GraphQL operations execute.

Loops are created with a `#foreach` and require you to supply a **loop variable** and an **iterable object** such as an array, list, map, or collection. A classic programming example with a `#foreach` loop is to loop over the items in a collection and print them out, so in our case we pluck them out and add them to the map:

```
#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
```

This example shows a few things. The first is using variables with the range `[..]` operator to create an iterable object. Then each item is referenced by a variable `$i` that you can operate with. In the previous example, you also see **Comments** that are denoted with a double pound `##`. This also showcases using the loop variable in both the keys or the values, as well as different methods of concatenation using strings.

Notice that `$i` is an integer, so you can call a `.toString()` method. For GraphQL types of INT, this can be handy.

You can also use a range operator directly, for example:

```
#foreach($item in [1..5])
  ...
#end
```

## Arrays

You have been manipulating a map up to this point, but arrays are also common in VTL. With arrays you also have access to some underlying methods such as `.isEmpty()`, `.size()`, `.set()`, `.get()`, and `.add()`, as shown below:

```
#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)))
```

The previous example used array index notation to retrieve an element with `arr2[$idx]`. You can look up by name from a Map/dictionary in a similar way:

```
#set($result = {
  "Author" : "Nadia",
  "Topic" : "GraphQL"
})

$util.qr($myMap.put("Author", $result["Author"]))
```

This is very common when filtering results coming back from data sources in Response Templates when using conditionals.

## Conditional Checks

The earlier section with `#foreach` showcased some examples of using logic to transform data with VTL. You can also apply conditional checks to evaluate data at runtime:

```
#if(!$array.isEmpty())
  $util.qr($myMap.put("ifCheck", "Array not empty"))
#else
  $util.qr($myMap.put("ifCheck", "Your array is empty"))
#end
```

The above `#if()` check of a Boolean expression is nice, but you can also use operators and `#elseif()` for branching:

```
#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
```

These two examples showed negation(!) and equality(==). We can also use `||`, `&&`, `>`, `<`, `>=`, `<=`, 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
```

**Note:** Only `Boolean.FALSE` and `null` are considered false in conditionals. Zero (0) and empty strings ("" ) are not equivalent to false.

## Operators

No programming language would be complete without some operators to perform some mathematical actions. Here are a few examples to get you started:

```
#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))
```

## Loops and Conditionals Together

It is very common when transforming data in VTL, such as before writing or reading from a data source, to loop over objects and then perform checks before performing an action. Combining some of the tools from the previous sections gives you a lot of functionality. One handy tool is knowing that `#foreach` automatically provides you with a `.count` on each item:

```
#foreach ($item in $arr2)
  #set($idx = "item" + $foreach.count)
  $util.qr($myMap.put($idx, $item))
#end
```

For example, maybe you want to just pluck out values from a map if it is under a certain size. Using the count along with conditionals and the `#break` statement allows you to do this:

```
#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
```

The previous `#foreach` is iterated over with `.keySet()`, which you can use on maps. This gives you access to get the `$key` and reference the value with a `.get($key)`. GraphQL arguments from clients

in AWS AppSync are stored as a map. They can also be iterated through with `.entrySet()`, which you can then access both keys and values as a Set, and either populate other variables or perform complex conditional checks, such as validation or transformation of input:

```
#foreach( $entry in $context.arguments.entrySet() )
#if ($entry.key == "XYZ" && $entry.value == "BAD")
    #set($myvar = "...")
#else
    #break
#end
#end
```

Other common examples are autopopulating default information, like the initial object versions when synchronizing data (very important in conflict resolution) or the default owner of an object for authorization checks - Mary created this blog post, so:

```
#set($myMap.owner = "Mary")
#set($myMap.defaultOwners = ["Admins", "Editors"])
```

## Context

Now that you are more familiar with performing logical checks in AWS AppSync resolvers with VTL, take a look at the context object:

```
$util.qr($myMap.put("context", $context))
```

This contains all of the information that you can access in your GraphQL request. For a detailed explanation, see the [context reference \(p. 279\)](#).

## Filtering

So far in this tutorial all information from your Lambda function has been returned to the GraphQL query with a very simple JSON transformation:

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

The VTL logic is just as powerful when you get responses from a data source, especially when doing authorization checks on resources. Let's walk through some examples. First try changing your response template like so:

```
#set($data = {
    "id" : "456",
    "meta" : "Valid Response"
})

$util.toJson($data)
```

No matter what happens with your GraphQL operation, hardcoded values are returned back to the client. Change this slightly so that the `meta` field is populated from the Lambda response, set earlier in the tutorial in the `elseifCheck` value when learning about conditionals:

```
#set($data = {
    "id" : "456"
})
```

```
#foreach($item in $context.result.entrySet())
  #if($item.key == "elseifCheck")
    $util.qr($data.put("meta", $item.value))
  #end
#end

$util.toJson($data)
```

`$context.result` is a map, so you can use `entrySet()` to perform logic on either the keys or the values returned. Because `$context.identity` contains information on the user that performed the GraphQL operation, if you return authorization information from the data source, then you can decide to return all, partial, or no data to a user based on your logic. Change your response template to look like the following:

```
#if($context.result["id"] == 123)
  $util.toJson($context.result)
#else
  $util.unauthorized()
#end
```

If you run your GraphQL query, the data will be returned as normal. However, if you change the `id` argument to something other than 123 (query `test { get(id:456 meta:"badrequest"){ } }`), you will get an authorization failure message.

You can find more examples of authorization scenarios in the [authorization use cases \(p. 233\)](#) section.

## Appendix - Template Sample

If you followed along with the tutorial, you may have built out this template step by step. In case you haven't, we include it below to copy for testing.

### Request Template

```
#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

##Operatorsdoesn'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)
}
```

## Response Template

```
#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
```

# Resolver mapping template context reference

AWS AppSync defines a set of variables and functions for working with resolver mapping templates. This makes logical operations on data easier with GraphQL. This document describes those functions and provides examples for working with templates.

## Accessing the `$context`

The `$context` variable is a map that holds all of the contextual information for your resolver invocation. It has the following structure:

```
{
  "arguments" : { ... },
  "source" : { ... },
  "result" : { ... },
  "identity" : { ... },
  "request" : { ... },
  "info": { ... }
}
```

**Note:** If you're trying to access a dictionary/map entry (such as an entry in `context`) by its key to retrieve the value, the Velocity Template Language (VTL) lets you directly use the notation `<dictionary-element>.<key-name>`. However, this might not work for all cases, such as when the key names have special characters (for example, an underscore `_`). We recommend that you always use `<dictionary-element>.get("<key-name>")` notation.

Each field in the `$context` map is defined as follows:

### **arguments**

A map that contains all GraphQL arguments for this field.

### **identity**

An object that contains information about the caller. For more information about the structure of this field, see [Identity \(p. 280\)](#).

### **source**

A map that contains the resolution of the parent field.

### **stash**

The stash is a map that is made available inside each resolver and function mapping template. The same stash instance lives through a single resolver execution. This means that you can use the stash to pass arbitrary data across request and response mapping templates, and across functions in a pipeline resolver. The stash exposes the same methods as the [Java Map](#) data structure.

### **result**

A container for the results of this resolver. This field is available only to response mapping templates.

For example, if you're resolving the `author` field of the following query:

```
query {
  getPost(id: 1234) {
    postId
    title
    content
  }
}
```

```
        author {
            id
            name
        }
    }
}
```

Then the full `$context` variable that is available when processing a response mapping template might be:

```
{
  "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" : "6666666666666666",
    "user" : "AIDAAAAAAAAAAAAAAAAAAAA"
  }
}
```

#### **prev.result**

The result of whatever previous operation was executed in a pipeline resolver. If the previous operation was the pipeline resolver request mapping template, then `$ctx.prev.result` represents the output of the evaluation of the template, and is made available to the first function in the pipeline. If the previous operation was the first function, then `$ctx.prev.result` represents the output of the first function, and is made available to the second function in the pipeline. If the previous operation was the last function, then `$ctx.prev.result` represents the output of the first function, and is made available to the second function in the pipeline. If the previous operation was the last function, then `$ctx.prev.result` represents the output of the last function, and is made available to the pipeline resolver response mapping template.

#### **info**

An object that contains information about the GraphQL request. For the structure of this field, see [Info \(p. 283\)](#).

## Identity

The `identity` section contains information about the caller. The shape of this section depends on the authorization type of your AWS AppSync API.

For more information about AWS AppSync security options, see [Authorization and Authentication \(p. 221\)](#).

#### **API\_KEY authorization**

The `identity` field isn't populated.



### **AWS\_LAMBDA authorization**

The identity contains the `resolverContext` key, containing the same `resolverContext` content returned by the Lambda function authorizing the request.

### **AWS\_IAM authorization**

The identity has the following shape:

```
{
  "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 authorization**

The identity has the following shape:

```
{
  "sub" : "uuid",
  "issuer" : "string",
  "username" : "string"
  "claims" : { ... },
  "sourceIp" : ["x.x.x.x"],
  "defaultAuthStrategy" : "string"
}
```

Each field is defined as follows:

#### **accountId**

The AWS account ID of the caller.

#### **claims**

The claims that the user has.

#### **cognitoIdentityAuthType**

Either authenticated or unauthenticated based on the identity type.

#### **cognitoIdentityAuthProvider**

A comma-separated list of external identity provider information used in obtaining the credentials used to sign the request.

#### **cognitoIdentityId**

The Amazon Cognito identity ID of the caller.

#### **cognitoIdentityPoolId**

The Amazon Cognito identity pool ID associated with the caller.

#### **defaultAuthStrategy**

The default authorization strategy for this caller (ALLOW or DENY).

**issuer**

The token issuer.

**sourceIp**

The source IP address of the caller that AWS AppSync receives. If the request doesn't include the `x-forwarded-for` header, the source IP value contains only a single IP address from the TCP connection. If the request includes a `x-forwarded-for` header, the source IP is a list of IP addresses from the `x-forwarded-for` header, in addition to the IP address from the TCP connection.

**sub**

The UUID of the authenticated user.

**user**

The IAM user.

**userArn**

The Amazon Resource Name (ARN) of the IAM user.

**username**

The user name of the authenticated user. In the case of `AMAZON_COGNITO_USER_POOLS` authorization, the value of `username` is the value of attribute `cognito:username`. In the case of `AWS_IAM` authorization, the value of `username` is the value of the AWS user principal. If you're using IAM authorization with credentials vended from Amazon Cognito identity pools, we recommend that you use `cognitoIdentityId`.

## Access request headers

AWS AppSync supports passing custom headers from clients and accessing them in your GraphQL resolvers by using `$context.request.headers`. You can then use the header values for actions such as inserting data into a data source or authorization checks. You can use single or multiple request headers using `$curl` with an API key from the command line, as shown in the following examples:

### Single header example

Suppose you set a header of `custom` with a value of `nadia` like the following:

```
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
```

This could then be accessed with `$context.request.headers.custom`. For example, it might be in the following VTL for DynamoDB:

```
"custom": $util.dynamodb.toDynamoDBJson($context.request.headers.custom)
```

### Multiple header example

You can also pass multiple headers in a single request and access these in the resolver mapping template. For example, if the `custom` header is set with two values:

```
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
```

You could then access these as an array, such as `$context.request.headers.custom[1]`.

**Note:** AWS AppSync doesn't expose the cookie header in `$context.request.headers`.

## Access the request custom domain name

AWS AppSync supports configuring a custom domain that you can use to access your GraphQL and real-time endpoints for your APIs. When making a request with a custom domain name, you can get the domain name using `$context.request.domainName`.

When using the default GraphQL endpoint domain name, the value is `null`.

## Info

The `info` section contains information about the GraphQL request. This section has the following shape:

```
{
  "fieldName": "string",
  "parentTypeName": "string",
  "variables": { ... },
  "selectionSetList": ["string"],
  "selectionSetGraphQL": "string"
}
```

Each field is defined as follows:

### **fieldName**

The name of the field that is currently being resolved.

### **parentTypeName**

The name of the parent type for the field that is currently being resolved.

### **variables**

A map which holds all variables that are passed into the GraphQL request.

### **selectionSetList**

A list representation of the fields in the GraphQL selection set. Fields that are aliased are referenced only by the alias name, not the field name. The following example shows this in detail.

### **selectionSetGraphQL**

A string representation of the selection set, formatted as GraphQL schema definition language (SDL). Although fragments aren't merged into the selection set, inline fragments are preserved, as shown in the following example.

**Note:** When using `$utils.toJson()` on `context.info`, the values that `selectionSetGraphQL` and `selectionSetList` return are not serialized by default.

For example, if you are resolving the `getPost` field of the following query:

```
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
  }
}

```

Then the full `$context.info` variable that is available when processing a mapping template might be:

```

{
  "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}"
}

```

**Note:** `selectionSetList` exposes only fields that belong to the current type. If the current type is an interface or union, only selected fields that belong to the interface are exposed. For example, given the following schema:

```

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!
}
```

And the following query:

```
query {
  node(id: "post1") {
    id
    ... on Post {
      title
    }
    ... on Blog {
      title
    }
  }
}
```

When calling `$ctx.info.selectionSetList` at the `Query.node` field resolution, only `id` is exposed:

```
"selectionSetList": [
  "id"
]
```

## Sanitizing inputs

Applications must sanitize untrusted inputs to prevent any external party from using an application outside of its intended use. As the `$context` contains user inputs in properties such as `$context.arguments`, `$context.identity`, `$context.result`, `$context.info.variables`, and `$context.request.headers`, care must be taken to sanitize their values in mapping templates.

Since mapping templates represent JSON, input sanitization takes the form of escaping JSON reserved characters from strings that represent user inputs. It is best practice to use the `$util.toJson()` utility to escape JSON reserved characters from sensitive string values when placing them into a mapping template.

For example, in the following Lambda request mapping template, because we accessed an unsafe customer input string (`$context.arguments.id`), we wrapped it with `$util.toJson()` to prevent unescaped JSON characters from breaking the JSON template.

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "$util.toJson($context.arguments.id)",
  }
}
```

```
    "postId": $util.toJson($context.arguments.id)
  }
}
```

As opposed to the mapping template below, where we directly insert `$context.arguments.id` without sanitization. This does not work for strings containing unescaped quotation marks or other JSON reserved characters, and can leave your template open to failure.

```
## 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.
  }
}
```

## Resolver mapping template utility reference

AWS AppSync defines a set of utilities that you can use within a GraphQL resolver to simplify interactions with data sources. Some of these utilities are for general use with any data source, such as generating IDs or timestamps. Others are specific to a type of data source.

### Topics

- [Utility helpers in `\$util` \(p. 286\)](#)
- [AWS AppSync directives \(p. 290\)](#)
- [Time helpers in `\$util.time` \(p. 290\)](#)
- [List helpers in `\$util.list` \(p. 292\)](#)
- [Map helpers in `\$util.map` \(p. 292\)](#)
- [DynamoDB helpers in `\$util.dynamodb` \(p. 293\)](#)
- [Amazon RDS helpers in `\$util.rds` \(p. 298\)](#)
- [HTTP helpers in `\$util.http` \(p. 300\)](#)
- [XML helpers in `\$util.xml` \(p. 301\)](#)
- [Transformation helpers in `\$util.transform` \(p. 302\)](#)
- [Math helpers in `\$util.math` \(p. 311\)](#)
- [String helpers in `\$util.str` \(p. 312\)](#)
- [Extensions \(p. 312\)](#)

## Utility helpers in `$util`

The `$util` variable contains general utility methods to help you work with data.

Unless otherwise specified, all utilities use the UTF-8 character set.

### `$util.qr()` and `$util.quiet()`

Runs a VTL statement while suppressing the returned value. This is useful for running methods without using temporary placeholders, such as adding items to a map. For example:

```
#set ($myMap = {})  
#set($discard = $myMap.put("id", "first value"))
```

Becomes:

```
#set ($myMap = {})  
$util.qr($myMap.put("id", "first value"))
```

**\$util.escapeJavaScript(String) : String**

Returns the input string as a JavaScript escaped string.

**\$util.urlEncode(String) : String**

Returns the input string as an application/x-www-form-urlencoded encoded string.

**\$util.urlDecode(String) : String**

Decodes an application/x-www-form-urlencoded encoded string back to its non-encoded form.

**\$util.base64Encode( byte[] ) : String**

Encodes the input into a base64-encoded string.

**\$util.base64Decode(String) : byte[]**

Decodes the data from a base64-encoded string.

**\$util.parseJson(String) : Object**

Takes "stringified" JSON and returns an object representation of the result.

**\$util.toJson(Object) : String**

Takes an object and returns a "stringified" JSON representation of that object.

**\$util.autoId() : String**

Returns a 128-bit randomly generated UUID.

**\$util.autoUlid() : String**

Returns a 128-bit randomly generated ULID (Universally Unique Lexicographically Sortable Identifier).

**\$util.autoKsuid() : String**

Returns a 128-bit randomly generated KSUID (K-Sortable Unique Identifier) base62 encoded as a String with a length of 27.

**\$util.unauthorized()**

Throws `Unauthorized` for the field being resolved. Use this in request or response mapping templates to determine whether to allow the caller to resolve the field.

**\$util.error(String)**

Throws a custom error. Use this in request or response mapping templates to detect an error with the request or with the invocation result.

**\$util.error(String, String)**

Throws a custom error. Use this in request or response mapping templates to detect an error with the request or with the invocation result. You can also specify an `errorType`.

**\$util.error(String, String, Object)**

Throws a custom error. Use this in request or response mapping templates to detect an error with the request or with the invocation result. You can also specify an `errorType` and a data field. The data value will be added to the corresponding `error` block inside `errors` in the GraphQL response. **Note:** data will be filtered based on the query selection set.

**\$util.error(String, String, Object, Object)**

Throws a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally, an `errorType` field, a data field, and a `errorInfo` field can be specified. The data value will be added to the corresponding `error` block inside `errors` in the GraphQL response. **Note:** data will be filtered based on the query selection set. The `errorInfo` value will be added to the corresponding `error` block inside `errors` in the GraphQL response. **Note:** `errorInfo` will **NOT** be filtered based on the query selection set.

**\$util.appendError(String)**

Appends a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Unlike `$util.error(String)`, the template evaluation will not be interrupted, so that data can be returned to the caller.

**\$util.appendError(String, String)**

Appends a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally, an `errorType` can be specified. Unlike `$util.error(String, String)`, the template evaluation will not be interrupted, so that data can be returned to the caller.

**\$util.appendError(String, String, Object)**

Appends a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally, an `errorType` and a data field can be specified. Unlike `$util.error(String, String, Object)`, the template evaluation will not be interrupted, so that data can be returned to the caller. The data value will be added to the corresponding `error` block inside `errors` in the GraphQL response. **Note:** data will be filtered based on the query selection set.

**\$util.appendError(String, String, Object, Object)**

Appends a custom error. This can be used in request or response mapping templates if the template detects an error with the request or with the invocation result. Additionally, an `errorType` field, a data field, and a `errorInfo` field can be specified. Unlike `$util.error(String, String, Object, Object)`, the template evaluation will not be interrupted, so that data can be returned to the caller. The data value will be added to the corresponding `error` block inside `errors` in the GraphQL response. **Note:** data will be filtered based on the query selection set. The `errorInfo` value will be added to the corresponding `error` block inside `errors` in the GraphQL response. **Note:** `errorInfo` will **NOT** be filtered based on the query selection set.

**\$util.validate(Boolean, String) : void**

If the condition is false, throw a `CustomTemplateException` with the specified message.

**\$util.validate(Boolean, String, String) : void**

If the condition is false, throw a `CustomTemplateException` with the specified message and error type.

**\$util.validate(Boolean, String, String, Object) : void**

If the condition is false, throw a `CustomTemplateException` with the specified message and error type, as well as data to return in the response.



**\$util.isNull(Object) : Boolean**

Returns true if the supplied object is null.

**\$util.isNullOrEmpty(String) : Boolean**

Returns true if the supplied data is null or an empty string. Otherwise, returns false.

**\$util.isNullOrBlank(String) : Boolean**

Returns true if the supplied data is null or a blank string. Otherwise, returns false.

**\$util.defaultIfNull(Object, Object) : Object**

Returns the first Object if it is not null. Otherwise, returns second object as a "default Object".

**\$util.defaultIfNullOrEmpty(String, String) : String**

Returns the first String if it is not null or empty. Otherwise, returns second String as a "default String".

**\$util.defaultIfNullOrBlank(String, String) : String**

Returns the first String if it is not null or blank. Otherwise, returns second String as a "default String".

**\$util.isString(Object) : Boolean**

Returns true if Object is a String.

**\$util.isNumber(Object) : Boolean**

Returns true if Object is a Number.

**\$util.isBoolean(Object) : Boolean**

Returns true if Object is a Boolean.

**\$util.isList(Object) : Boolean**

Returns true if Object is a List.

**\$util.isMap(Object) : Boolean**

Returns true if Object is a Map.

**\$util.typeOf(Object) : String**

Returns a String describing the type of the Object. Supported type identifications are: "Null", "Number", "String", "Map", "List", "Boolean". If a type cannot be identified, the return type is "Object".

**\$util.matches(String, String) : Boolean**

Returns true if the specified pattern in the first argument matches the supplied data in the second argument. The pattern must be a regular expression such as `$util.matches("a*b", "aaaaab")`. The functionality is based on [Pattern](#), which you can reference for further documentation.

**\$util.authType() : String**

Returns a String describing the multi-auth type being used by a request, returning back either "IAM Authorization", "User Pool Authorization", "Open ID Connect Authorization", or "API Key Authorization".

**\$util.log.info(Object) : Void**

Logs the String representation of the provided Object to the requested log stream when request-level and field-level CloudWatch logging is enabled with log level ALL on an API.

**\$util.log.info(String, Object...) : Void**

Logs the String representation of the provided Objects to the requested log stream when request-level and field-level CloudWatch logging is enabled with log level `ALL` on an API. This utility will replace all variables indicated by "{}" in the first input format String with the String representation of the provided Objects in order.

**\$util.log.error(Object) : Void**

Logs the String representation of the provided Object to the requested log stream when field-level CloudWatch logging is enabled with log level `ERROR` or log level `ALL` on an API.

**\$util.log.error(String, Object...) : Void**

Logs the String representation of the provided Objects to the requested log stream when field-level CloudWatch logging is enabled with log level `ERROR` or log level `ALL` on an API. This utility will replace all variables indicated by "{}" in the first input format String with the String representation of the provided Objects in order.

## AWS AppSync directives

AWS AppSync exposes directives to facilitate developer productivity when writing Velocity.

**#return(Object)** The **#return** directive comes handy if you need to return prematurely from any mapping template. **#return** is analogous to the *return* keyword in programming languages, as it will return from the closest scoped block of logic. What this means is using **#return** inside a resolver mapping template will return from the resolver. Additionally, using **#return** from a function mapping template will return from the function and will continue the execution to either the next function in the pipeline or the resolver response mapping template.

**#return** Same as **#return(Object)** but `null` will be returned instead.

## Time helpers in \$util.time

The `$util.time` variable contains datetime methods to help generate timestamps, convert between datetime formats, and parse datetime strings. The syntax for datetime formats is based on [DateTimeFormatter](#) which you can reference for further documentation. Below we provide some examples, as well as a list of available methods and descriptions.

## Standalone function examples

```
$util.time.nowISO8601()           :  
2018-02-06T19:01:35.749Z  
$util.time.nowEpochSeconds()    : 1517943695  
$util.time.nowEpochMillis()     : 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
```

## Conversion examples

```
#set( $nowEpochMillis = 1517943695758 )  
$util.time.epochMillisToSeconds($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
```

## Parsing examples

```
$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
```

## Usage with AWS AppSync defined scalars

The following formats are compatible with `AWSDate`, `AWSDatetime`, and `AWSTime`.

```
$util.time.nowFormatted("yyyy-MM-dd[XXX]", "-07:00:30") : 2018-07-11-07:00
$util.time.nowFormatted("yyyy-MM-dd'T'HH:mm:ss[XXXXX]", "-07:00:30") :
2018-07-11T15:14:15-07:00:30
```

**`$util.time.nowISO8601()` : String**

Returns a String representation of UTC in [ISO8601 format](#).

**`$util.time.nowEpochSeconds()` : long**

Returns the number of seconds from the epoch of 1970-01-01T00:00:00Z to now.

**`$util.time.nowEpochMilliseconds()` : long**

Returns the number of milliseconds from the epoch of 1970-01-01T00:00:00Z to now.

**`$util.time.nowFormatted(String)` : String**

Returns a string of the current timestamp in UTC using the specified format from a String input type.

**`$util.time.nowFormatted(String, String)` : String**

Returns a string of the current timestamp for a timezone using the specified format and timezone from String input types.

**`$util.time.parseFormattedToEpochMilliseconds(String, String)` : Long**

Parses a timestamp passed as a String, along with a format, and return the timestamp as milliseconds since epoch.

**`$util.time.parseFormattedToEpochMilliseconds(String, String, String)` : Long**

Parses a timestamp passed as a String, along with a format and time zone, and return the timestamp as milliseconds since epoch.

**`$util.time.parseISO8601ToEpochMilliseconds(String)` : Long**

Parses an ISO8601 timestamp, passed as a String, and return the timestamp as milliseconds since epoch.

**`$util.time.epochMillisecondsToSeconds(long)` : long**

Converts an epoch milliseconds timestamp to an epoch seconds timestamp.

**\$util.time.epochMillisecondsToISO8601(long) : String**

Converts a epoch milliseconds timestamp to an ISO8601 timestamp.

**\$util.time.epochMillisecondsToFormatted(long, String) : String**

Converts a epoch milliseconds timestamp, passed as long, to a timestamp formatted according to the supplied format in UTC.

**\$util.time.epochMillisecondsToFormatted(long, String, String) : String**

Converts a epoch milliseconds timestamp, passed as a long, to a timestamp formatted according to the supplied format in the supplied timezone.

## List helpers in \$util.list

\$util.list contains methods to help with common List operations, such as removing or retaining items from a list for filtering use cases.

**\$util.list.copyAndRetainAll(List, List) : List**

Makes a shallow copy of the supplied list in the first argument, retaining only the items specified in the second argument, if they are present. All other items will be removed from the copy.

**\$util.list.copyAndRemoveAll(List, List) : List**

Makes a shallow copy of the supplied list in the first argument, removing any items where the item is specified in the second argument, if they are present. All other items will be retained in the copy.

**\$util.list.sortList(List, Boolean, String) : List**

Sorts a list of objects, which is provided in the first argument. If the second argument is true, the list is sorted in a descending manner; if the second argument is false, the list is sorted in an ascending manner. The third argument is the string name of the property used to sort a list of custom objects. If it's a list of just Strings, Integers, Floats, or Doubles, the third argument can be any random string. If all of the objects are not from the same class, the original list is returned. Only lists containing a maximum of 1000 objects are supported. The following is an example of this utility's usage:

```
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}]
```

## Map helpers in \$util.map

\$util.map contains methods to help with common Map operations, such as removing or retaining items from a Map for filtering use cases.

**\$util.map.copyAndRetainAllKeys(Map, List) : Map**

Makes a shallow copy of the first map, retaining only the keys specified in the list, if they are present. All other keys will be removed from the copy.

**\$util.map.copyAndRemoveAllKeys(Map, List) : Map**

Makes a shallow copy of the first map, removing any entries where the key is specified in the list, if they are present. All other keys will be retained in the copy.

## DynamoDB helpers in `$util.dynamodb`

`$util.dynamodb` contains helper methods that make it easier to write and read data to Amazon DynamoDB, such as automatic type mapping and formatting. These methods are designed to make mapping primitive types and Lists to the proper DynamoDB input format automatically, which is a Map of the format `{ "TYPE" : VALUE }`.

For example, previously, a request mapping template to create a new item in DynamoDB might have looked like this:

```
{
  "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) }
  }
}
```

If we wanted to add fields to the object we would have to update the GraphQL query in the schema, as well as the request mapping template. However, we can now restructure our request mapping template so it automatically picks up new fields added in our schema and adds them to DynamoDB with the correct types:

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

In the previous example, we are using the `$util.dynamodb.toDynamoDBJson(...)` helper to automatically take the generated id and convert it to the DynamoDB representation of a string attribute. We then take all the arguments and convert them to their DynamoDB representations and output them to the `attributeValues` field in the template.

Each helper has two versions: a version that returns an object (for example, `$util.dynamodb.toString(...)`), and a version that returns the object as a JSON string (for example, `$util.dynamodb.toStringJson(...)`). In the previous example, we used the version that returns the data as a JSON string. If you want to manipulate the object before it's used in the template, you can choose to return an object instead, as shown following:

```
{
  "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)
}
```

In the previous example, we are returning the converted arguments as a map instead of a JSON string, and are then adding the `version` and `timestamp` fields before finally outputting them to the `attributeValues` field in the template using `$util.toJson(...)`.

The JSON version of each of the helpers is equivalent to wrapping the non-JSON version in `$util.toJson(...)`. For example, the following statements are exactly the same:

```
$util.toStringJson("Hello, World!")
$util.toJson($util.toString("Hello, World!"))
```

#### **`$util.dynamodb.toDynamoDB(Object) : Map`**

General object conversion tool for DynamoDB that converts input objects to the appropriate DynamoDB representation. It's opinionated about how it represents some types: e.g., it will use lists ("L") rather than sets ("SS", "NS", "BS"). This returns an object that describes the DynamoDB attribute value.

String example:

```
Input:      $util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

Number example:

```
Input:      $util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

Boolean example:

```
Input:      $util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

List example:

```
Input:      $util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
      "L" : [
        { "S" : "foo" },
        { "N" : 123 },
        {
          "M" : {
            "bar" : { "S" : "baz" }
          }
        }
      ]
    }
```

Map example:

```
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**

The same as `$util.dynamodb.toDynamoDB(Object) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

**\$util.dynamodb.toString(String) : String**

Convert an input string to the DynamoDB string format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

**\$util.dynamodb.toStringJson(String) : Map**

The same as `$util.dynamodb.toString(String) : String`, but returns the DynamoDB attribute value as a JSON encoded string.

**\$util.dynamodb.toStringSet(List<String>) : Map**

Converts a lists with Strings to the DynamoDB string set format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

**\$util.dynamodb.toStringSetJson(List<String>) : String**

The same as `$util.dynamodb.toStringSet(List<String>) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

**\$util.dynamodb.toNumber(Number) : Map**

Converts a number to the DynamoDB number format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

**\$util.dynamodb.toNumberJson(Number) : String**

The same as `$util.dynamodb.toNumber(Number) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

**\$util.dynamodb.toNumberSet(List<Number>) : Map**

Converts a list of numbers to the DynamoDB number set format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

**\$util.dynamodb.toNumberSetJson(List<Number>) : String**

The same as `$util.dynamodb.toNumberSet(List<Number>) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

**\$util.dynamodb.toBinary(String) : Map**

Converts binary data encoded as a base64 string to DynamoDB binary format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

**\$util.dynamodb.toBinaryJson(String) : String**

The same as `$util.dynamodb.toBinary(String) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

**\$util.dynamodb.toBinarySet(List<String>) : Map**

Converts a list of binary data encoded as base64 strings to DynamoDB binary set format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

**\$util.dynamodb.toBinarySetJson(List<String>) : String**

The same as `$util.dynamodb.toBinarySet(List<String>) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

**\$util.dynamodb.toBoolean(Boolean) : Map**

Converts a Boolean to the appropriate DynamoDB Boolean format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

**\$util.dynamodb.toBooleanJson(Boolean) : String**

The same as `$util.dynamodb.toBoolean(Boolean) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

**\$util.dynamodb.toNull() : Map**

Returns a null in DynamoDB null format. This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toNull()
Output:     { "NULL" : null }
```

**\$util.dynamodb.toNullJson() : String**

The same as `$util.dynamodb.toNull() : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

**\$util.dynamodb.toList(List) : Map**

Converts a list of object to DynamoDB list format. Each item in the list is also converted to its appropriate DynamoDB format. It's opinionated about how it represents some of the nested objects:



e.g., it will use lists ("L") rather than sets ("SS", "NS", "BS"). This returns an object that describes the DynamoDB attribute value.

```
Input:      $util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
              "L" : [
                  { "S" : "foo" },
                  { "N" : 123 },
                  {
                      "M" : {
                          "bar" : { "S" : "baz" }
                      }
                  }
              ]
            }
```

**\$util.dynamodb.toListJson(List) : String**

The same as \$util.dynamodb.toList(List) : Map, but returns the DynamoDB attribute value as a JSON encoded string.

**\$util.dynamodb.toMap(Map) : Map**

Converts a map to DynamoDB map format. Each value in the map is also converted to its appropriate DynamoDB format. It's opinionated about how it represents some of the nested objects: e.g., it will use lists ("L") rather than sets ("SS", "NS", "BS"). This returns an object that describes the DynamoDB attribute value.

```
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**

The same as \$util.dynamodb.toMap(Map) : Map, but returns the DynamoDB attribute value as a JSON encoded string.

**\$util.dynamodb.toMapValues(Map) : Map**

Creates a copy of the map where each value has been converted to its appropriate DynamoDB format. It's opinionated about how it represents some of the nested objects: e.g., it will use lists ("L") rather than sets ("SS", "NS", "BS").

```
Input:      $util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
[ "boop" ] })
Output:     {
              "foo" : { "S" : "bar" },
              "baz" : { "N" : 1234 },
              "beep" : {
                  "L" : [
                      { "S" : "boop" }
                  ]
              }
            }
```

Note: this is slightly different to `$util.dynamodb.toMap(Map) : Map` as it returns only the contents of the DynamoDB attribute value, but not the whole attribute value itself. For example, the following statements are exactly the same:

```
$util.dynamodb.toMapValues($map)
$util.dynamodb.toMap($map).get("M")
```

**`$util.dynamodb.toMapValuesJson(Map) : String`**

The same as `$util.dynamodb.toMapValues(Map) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

**`$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`**

Converts the key, bucket and region into the DynamoDB S3 Object representation. This returns an object that describes the DynamoDB attribute value.

```
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`**

The same as `$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

**`$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map`**

Converts the key, bucket, region and optional version into the DynamoDB S3 Object representation. This returns an object that describes the DynamoDB attribute value.

```
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`**

The same as `$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map`, but returns the DynamoDB attribute value as a JSON encoded string.

**`$util.dynamodb.fromS3ObjectJson(String) : Map`**

Accepts the string value of a DynamoDB S3 Object and returns a map that contains the key, bucket, region and optional version.

```
Input:      $util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" : "beep" }
```

## Amazon RDS helpers in \$util.rds

**`$util.rds.toJsonString(String serializedSQLResult): String`**

Returns a `String` by transforming the stringified raw Amazon Relational Database Service (Amazon RDS) Data API operation result format to a more concise string. The returned string is a serialized list

of SQL records of the result set. Every record is represented as a collection of key-value pairs. The keys are the corresponding column names.

If the corresponding statement in the input was a SQL query that causes a mutation (for example INSERT, UPDATE, DELETE), then an empty list is returned. For example, the query `select * from Books limit 2` provides the raw result from the Amazon RDS Data operation:

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

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

The `util.rds.toJsonString` is:

```
[
  {
    "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`**

This is the same as `util.rds.toJsonString`, but with the result being a JSON Object.

## HTTP helpers in \$util.http

The `$util.http` utility provides helper methods that you can use to manage HTTP request parameters and to add response headers.

**`$util.http.copyHeaders(Map) : Map`**

Copies the header from the map without the restricted set of HTTP headers. You can use this to forward request headers to your downstream HTTP endpoint.

```
{
  ...
  "params": {
    ...
    "headers": $util.http.copyHeaders($ctx.request.headers),
    ...
  },
  ...
}
```

```
}
```

### **\$util.http.addResponseHeader(String, Object)**

Adds a single custom header with the name (String) and value (Object) of the response. The following limitations apply:

- Header names can't match any of the existing or restricted AWS or AWS AppSync headers.
- Header names can't start with restricted prefixes, such as `x-amzn-` or `x-amz-`.
- The size of custom response headers can't exceed 4 KB. This includes header names and values.
- You should define each response header once per GraphQL operation. However, if you define a custom header with the same name multiple times, the most recent definition appears in the response. All headers count towards the header size limit regardless of naming.

```
...
$util.http.addResponseHeader("itemsCount", 7)
$util.http.addResponseHeader("render", $ctx.args.render)
...
```

### **\$util.http.addResponseHeaders(Map)**

Adds multiple response headers to the response from the specified map of names (String) and values (Object). The same limitations listed for the `addResponseHeader(String, Object)` method also apply to this method.

```
...
#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)
...
```

## XML helpers in \$util.xml

`$util.xml` contains helper methods that can make it easier to translate XML responses to JSON or a Dictionary.

### **\$util.xml.toMap(String) : Map**

Converts an XML string to a Dictionary.

```
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**

Converts an XML string to a JSON string. This is similar to *toMap*, except that the output is a string. This is useful if you want to directly convert and return the XML response from an HTTP object to JSON.

#### **\$util.xml.toJsonString(String, Boolean) : String**

Converts an XML string to a JSON string with an optional Boolean parameter to determine if you want to string-encode the JSON.

## Transformation helpers in \$util.transform

`$util.transform` contains helper methods that make it easier to perform complex operations against data sources, such as Amazon DynamoDB filter operations.

#### **\$util.transform.toDynamoDBFilterExpression(Map) : Map**

Converts an input string to a filter expression for use with DynamoDB.

```
Input:

$util.transform.toDynamoDBFilterExpression({
  "title":{
    "contains":"Hello World"
```

```
    }  
  })  
  
Output:  
  
{  
  "expression" : "contains(#title, :title_contains)"  
  "expressionNames" : {  
    "#title" : "title",  
  },  
  "expressionValues" : {  
    ":title_contains" : { "S" : "Hello World" }  
  },  
}
```

**\$util.transform.toElasticsearchQueryDSL(Map) : Map**

Converts the given input into its equivalent OpenSearch Query DSL expression, returning it as a JSON string.

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

The default operator is assumed to be AND.

**`$util.transform.toSubscriptionFilter(Map) : Map`**

Converts a Map input object to a SubscriptionFilter expression object. The `$util.transform.toSubscriptionFilter` method is used as an input to the `$extensions.setSubscriptionFilter()` extension. For more information, see [Extensions](#).

**`$util.transform.toSubscriptionFilter(Map, List) : Map`**

Converts a Map input object to a SubscriptionFilter expression object. The `$util.transform.toSubscriptionFilter` method is used as an input to the `$extensions.setSubscriptionFilter()` extension. For more information, see [Extensions](#).

The first argument is the Map input object that's converted to the SubscriptionFilter expression object. The second argument is a List of field names that are ignored in the first Map input object while constructing the SubscriptionFilter expression object.

**`$util.transform.toSubscriptionFilter(Map, List, Map) : Map`**

Converts a Map input object to a SubscriptionFilter expression object. The `$util.transform.toSubscriptionFilter` method is used as an input to the `$extensions.setSubscriptionFilter()` extension. For more information, see [Extensions](#).

The first argument is the Map input object that's converted to the SubscriptionFilter expression object, the second argument is a List of field names that will be ignored in the first Map input object, and the third argument is a Map input object of strict rules that's included while constructing the SubscriptionFilter expression object. These strict rules are included in the SubscriptionFilter expression object in such a way that at least one of the rules will be satisfied to pass the subscription filter.

The following table explains the how the arguments of the following utilities are defined:

- `$util.transform.toSubscriptionFilter(Map) : Map`
- `$util.transform.toSubscriptionFilter(Map, List) : Map`
- `$util.transform.toSubscriptionFilter(Map, List, Map) : Map`

**Argument 1: Map**

Argument 1 is a Map object with the following key values:



- field names
- "and"
- "or"

For field names as keys, the conditions on these fields' entries are in the form of "operator" : "value".

The following example shows how entries can be added to the Map:

```
"field_name" : {
    "operator1" : value
}

## We can have multiple conditions for the same field_name:

"field_name" : {
    "operator1" : value
    "operator2" : value
    .
    .
    .
}
```

When a field has two or more conditions on it, all of these conditions are considered to use the OR operation.

The input Map can also have "and" and "or" as keys, implying that all entries within these should be joined using AND or OR logic depending on the key. The key values "and" and "or" expect an array of conditions.

```
"and" : [

    {
        "field_name1" : {
            "operator1" : value
        }
    },

    {
        "field_name2" : {
            "operator1" : value
        }
    },

    :
    .

].
```

Note that you can nest "and" and "or". That is, you can have nested "and"/"or" within another "and"/"or" block. However, this doesn't work for simple fields.

```
"and" : [

    {
        "field_name1" : {
            "operator" : value
        }
    },

    {
```

```
      "or" : [
        {
          "field_name2" : {
            "operator" : value
          }
        },
        {
          "field_name3" : {
            "operator" : value
          }
        }
      ]
    }
  ].
```

The following example shows an input of *argument 1* using `$util.transform.toSubscriptionFilter(Map) : Map`.

### Input(s)

Argument 1: Map:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

### Output

The result is a Map object:

```
{
  "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

Argument 2 contains a `List` of field names that shouldn't be considered in the input `Map` (argument 1) while constructing the `SubscriptionFilter` expression object. The `List` can also be empty.

The following example shows the inputs of argument 1 and argument 2 using `$util.transform.toSubscriptionFilter(Map, List) : Map`.

#### Input(s)

Argument 1: Map:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    }
  ]
}
```

```
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argument 2: List:

```
["percentageUp", "author"]
```

### Output

The result is a Map object:

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

### Argument 3: Map

Argument 3 is a Map object that has field names as key values (cannot have "and" or "or"). For field names as keys, the conditions on these fields are entries in the form of "operator" : "value". Unlike argument 1, argument 3 cannot have multiple conditions in the same key. In addition, argument 3 doesn't have an "and" or "or" clause, so there's no nesting involved either.

Argument 3 represents a list of strict rules, which are added to the SubscriptionFilter expression object so that **at least one** of these conditions is met to pass the filter.

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
```

```
.
```

The following example shows the inputs of *argument 1*, *argument 2*, and *argument 3* using `$util.transform.toSubscriptionFilter(Map, List, Map) : Map`.

### Input(s)

Argument 1: Map:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argument 2: List:

```
["percentageUp", "author"]
```

Argument 3: Map:

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
}
```

### Output

The result is a Map object:

```
{
```

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

## Math helpers in \$util.math

\$util.math contains methods to help with common Math operations.

**\$util.math.roundNum(Double) : Integer**

Takes a double and rounds it to the nearest integer.

**\$util.math.minVal(Double, Double) : Double**

Takes two doubles and returns the minimum value between the two doubles.

**\$util.math.maxVal(Double, Double) : Double**

Takes two doubles and returns the maximum value between the two doubles.

**\$util.math.randomDouble() : Double**

Returns a random double between 0 and 1.

**Important**

This function shouldn't be used for anything that needs high entropy randomness (for example, cryptography).

**\$util.math.randomWithinRange(Integer, Integer) : Integer**

Returns a random integer value within the specified range, with the first argument specifying the lower value of the range and the second argument specifying the upper value of the range.

**Important**

This function shouldn't be used for anything that needs high entropy randomness (for example, cryptography).

## String helpers in \$util.str

\$util.str contains methods to help with common String operations.

**\$util.str.toUpper(String) : String**

Takes a string and converts it to be entirely uppercase.

**\$util.str.toLower(String) : String**

Takes a string and converts it to be entirely lowercase.

**\$util.str.replace(String, String, String) : String**

Replaces a substring within a string with another string. The first argument specifies the string on which to perform the replacement operation. The second argument specifies the substring to replace. The third argument specifies the string to replace the second argument with. The following is an example of this utility's usage:

```
INPUT:      $util.str.replace("hello world", "hello", "mellow")
OUTPUT:     "mellow world"
```

**\$util.str.normalize(String, String) : String**

Normalizes a string using one of the four unicode normalization forms: NFC, NFD, NFKC, or NFKD. The first argument is the string to normalize. The second argument is either "nfc", "nfd", "nfkc", or "nfkd" specifying the normalization type to use for the normalization process.

## Extensions

\$extensions contains a set of methods to make additional actions within your resolvers.

**\$extensions.evictFromApiCache(String, String, Object) : Object**

Evicts an item from the AWS AppSync server-side cache. The first argument is the type name. The second argument is the field name. The third argument is an object containing key-value pair items that specify



the caching key value. You must put the items in the object in the same order as the caching keys in the cached resolver's `cachingKey`.

**Note**

This utility works only for mutations, not queries.

**`$extensions.setSubscriptionFilter(filterJsonObject)`**

Defines enhanced subscription filters. Each subscription notification event is evaluated against provided subscription filters and delivers notifications to clients if all filters evaluate to `true`. The argument is `filterJsonObject` as described in the following.

**Note**

You can use this extension method only in the response mapping templates of a subscription resolver.

**`$extensions.setSubscriptionInvalidationFilter(filterJsonObject)`**

Defines subscription invalidation filters. Subscription filters are evaluated against the invalidation payload, then invalidate a given subscription if the filters evaluate to `true`. The argument is `filterJsonObject` as described in the following.

**Note**

You can use this extension method only in the response mapping templates of a subscription resolver.

**Argument: `filterJsonObject`**

The JSON object defines either subscription or invalidation filters. It's an array of filters in a `filterGroup`. Each filter is a collection of individual filters.

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "userId",
          "operator": "eq",
          "value": 1
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "group",
          "operator": "in",
          "value": ["Admin", "Developer"]
        }
      ]
    }
  ]
}
```

Each filter has three attributes:

- `fieldName` – The GraphQL schema field.
- `operator` – The operator type.
- `value` – The values to compare to the subscription notification `fieldName` value.

The following is an example assignment of these attributes:

```
{
  "fieldName" : "severity",
  "operator" : "le",
  "value" : $context.result.severity
}
```

#### Field: fieldName

The string type `fieldName` refers to a field defined in the GraphQL schema that matches the `fieldName` in the subscription notification payload. When a match is found, the value of the GraphQL schema field is compared to the value of the subscription notification filter. In the following example, the `fieldName` filter matches the `service` field defined in a given GraphQL type. If the notification payload contains a `service` field with a value equivalent to `AWS AppSync`, the filter evaluates to `true`:

```
{
  "fieldName" : "service",
  "operator" : "eq",
  "value" : "AWS AppSync"
}
```

#### Field: value

The value can be a different type based on the operator:

- A single number or Boolean
  - String examples: "test", "service"
  - Number examples: 1, 2, 45.75
  - Boolean examples: true, false
- Pairs of numbers or strings
  - String pair example: ["test1", "test2"], ["start", "end"]
  - Number pair example: [1, 4], [67, 89], [12.45, 95.45]
- Arrays of numbers or strings
  - String array example: ["test1", "test2", "test3", "test4", "test5"]
  - Number array example: [1, 2, 3, 4, 5], [12.11, 46.13, 45.09, 12.54, 13.89]

#### Field: operator

A case-sensitive string with the following possible values:

Operator	Description	Possible value types
eq	Equal	integer, float, string, Boolean
ne	Not equal	integer, float, string, Boolean
le	Less than or equal	integer, float, string
lt	Less than	integer, float, string
ge	Greater than or equal	integer, float, string
gt	Greater than	integer, float, string

contains	Checks for a subsequence or value in the set.	integer, float, string
notContains	Checks for the absence of a subsequence or absence of a value in the set.	integer, float, string
beginsWith	Checks for a prefix.	string
in	Checks for matching elements that are in the list.	Array of integer, float, or string
notIn	Checks for matching elements that aren't in the list.	Array of integer, float, or string
between	Between two values	integer, float, string

The following table describes how each operator is used in the subscription notification.

#### eq (equal)

The `eq` operator evaluates to `true` if the subscription notification field value matches and is strictly equal to the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `service` field with the value equivalent to `AWS AppSync`.

**Possible value types:** integer, float, string, Boolean

```
{
  "fieldName" : "service",
  "operator" : "eq",
  "value" : "AWS AppSync"
}
```

#### ne (not equal)

The `ne` operator evaluates to `true` if the subscription notification field value is different from the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `service` field with a value different from `AWS AppSync`.

**Possible value types:** integer, float, string, Boolean

```
{
  "fieldName" : "service",
  "operator" : "ne",
  "value" : "AWS AppSync"
}
```

#### le (less or equal)

The `le` operator evaluates to `true` if the subscription notification field value is less than or equal to the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `size` field with a value less than or equal to 5.

**Possible value types:** integer, float, string

```
{
  "fieldName" : "size",
```

```
"operator" : "le",  
"value" : 5  
}
```

#### lt (less than)

The `lt` operator evaluates to `true` if the subscription notification field value is lower than the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `size` field with a value lower than 5.

**Possible value types:** integer, float, string

```
{  
  "fieldName" : "size",  
  "operator" : "lt",  
  "value" : 5  
}
```

#### ge (greater or equal)

The `ge` operator evaluates to `true` if the subscription notification field value is greater than or equal to the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `size` field with a value greater than or equal to 5.

**Possible value types:** integer, float, string

```
{  
  "fieldName" : "size",  
  "operator" : "ge",  
  "value" : 5  
}
```

#### gt (greater than)

The `gt` operator evaluates to `true` if the subscription notification field value is greater than the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `size` field with a value greater than 5.

**Possible value types:** integer, float, string

```
{  
  "fieldName" : "size",  
  "operator" : "gt",  
  "value" : 5  
}
```

#### contains

The `contains` operator checks for a substring, subsequence, or value in a set or single item. A filter with the `contains` operator evaluates to `true` if the subscription notification field value contains the filter value. In the following example, the filter evaluates to `true` if the subscription notification has a `seats` field with the array value containing the value 10.

**Possible value types:** integer, float, string

```
{  
  "fieldName" : "seats",  
}
```

```
"operator" : "contains",  
"value" : 10  
}
```

In another example, the filter evaluates to `true` if the subscription notification has an event field with `launch` as substring.

```
{  
  "fieldName" : "event",  
  "operator" : "contains",  
  "value" : "launch"  
}
```

#### notContains

The `notContains` operator checks for the absence of a substring, subsequence, or value in a set or single item. The filter with the `notContains` operator evaluates to `true` if the subscription notification field value doesn't contain the filter value. In the following example, the filter evaluates to `true` if the subscription notification has a `seats` field with the array value not containing the value 10.

**Possible value types:** integer, float, string

```
{  
  "fieldName" : "seats",  
  "operator" : "notContains",  
  "value" : 10  
}
```

In another example, filter evaluates to `true` if the subscription notification has an event field value without `launch` as its subsequence.

```
{  
  "fieldName" : "event",  
  "operator" : "notContains",  
  "value" : "launch"  
}
```

#### beginsWith

The `beginsWith` operator checks for a prefix in a string. The filter containing the `beginsWith` operator evaluates to `true` if the subscription notification field value begins with the filter's value. In the following example, the filter evaluates to `true` if the subscription notification has a `service` field with a value that begins with `AWS`.

**Possible value type:** string

```
{  
  "fieldName" : "service",  
  "operator" : "beginsWith",  
  "value" : "AWS"  
}
```

#### in

The `in` operator checks for matching elements in an array. The filter containing the `in` operator evaluates to `true` if the subscription notification field value exists in an array. In the following

example, the filter evaluates to `true` if the subscription notification has a `severity` field with one of the values present in the array: `[1, 2, 3]`.

**Possible value type:** Array of integer, float, or string

```
{
  "fieldName" : "severity",
  "operator" : "in",
  "value" : [1,2,3]
}
```

#### notIn

The `notIn` operator checks for missing elements in an array. The filter containing the `notIn` operator evaluates to `true` if the subscription notification field value doesn't exist in the array. In the following example, the filter evaluates to `true` if the subscription notification has a `severity` field with one of the values not present in the array: `[1, 2, 3]`.

**Possible value type:** Array of integer, float, or string

```
{
  "fieldName" : "severity",
  "operator" : "notIn",
  "value" : [1,2,3]
}
```

#### between

The `between` operator checks for values between two numbers or strings. The filter containing the `between` operator evaluates to `true` if the subscription notification field value is between the filter's value pair. In the following example, the filter evaluates to `true` if the subscription notification has a `severity` field with values 2,3,4.

**Possible value types:** Pair of integer, float, or string

```
{
  "fieldName" : "severity",
  "operator" : "between",
  "value" : [1,5]
}
```

#### AND logic

You can combine multiple filters using AND logic by defining multiple entries within the `filters` object in the `filterGroup` array. In the following example, filters evaluate to `true` if the subscription notification has a `userId` field with a value equivalent to 1 AND a `group` field value of either `Admin` or `Developer`.

```
{
  "filterGroup": [
    {
      "filters" : [
        {
          "fieldName" : "userId",
          "operator" : "eq",
          "value" : 1
        },
        {
          "fieldName" : "group",
          "operator" : "in",
          "value" : ["Admin", "Developer"]
        }
      ]
    }
  ]
}
```

```

        {
          "fieldName" : "group",
          "operator" : "in",
          "value" : ["Admin", "Developer"]
        }
      ]
    }
  ]
}

```

### OR logic

You can combine multiple filters using OR logic by defining multiple filter objects within the `filterGroup` array. In the following example, filters evaluate to `true` if the subscription notification has a `userId` field with a value equivalent to 1 OR a `group` field value of either `Admin` or `Developer`.

```

{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName" : "userId",
          "operator" : "eq",
          "value" : 1
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName" : "group",
          "operator" : "in",
          "value" : ["Admin", "Developer"]
        }
      ]
    }
  ]
}

```

Note that there are several restrictions for using filters:

- In the `filters` object, there can be a maximum of five unique `fieldName` items per filter. This means that you can combine a maximum of five individual `fieldName` objects using AND logic.
- There can be a maximum of five values for `in` and `notIn` operators.
- Each string can be a maximum of 256 characters.
- Each string comparison is case sensitive.
- Each `filterGroup` can have a maximum of 10 filters. This means that you can combine a maximum of 10 filters using OR logic.
- The `in` operator is a special case of OR logic. In the following example, there are two filters:

```

{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName" : "userId",

```

```

        "operator" : "eq",
        "value" : 1
      },
      {
        "fieldName" : "group",
        "operator" : "in",
        "value" : ["Admin", "Developer"]
      }
    ]
  }
]
}

```

The preceding filter group is evaluated as follows and counts towards the maximum filters limit:

```

{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName" : "userId",
          "operator" : "eq",
          "value" : 1
        },
        {
          "fieldName" : "group",
          "operator" : "eq",
          "value" : "Admin"
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName" : "userId",
          "operator" : "eq",
          "value" : 1
        },
        {
          "fieldName" : "group",
          "operator" : "eq",
          "value" : "Developer"
        }
      ]
    }
  ]
}

```

#### **`$extensions.invalidateSubscriptions(invalidationJsonObject)`**

Used to initiate a subscription invalidation from a mutation. The argument is `invalidationJsonObject` as described in the following.

##### **Note**

This extension can be used only in the response mapping templates of the mutation resolvers.

#### **Argument: `invalidationJsonObject`**

The `invalidationJsonObject` defines the following:

- `subscriptionField` – The GraphQL schema subscription to invalidate. A single subscription, defined as a string in the `subscriptionField`, is considered for invalidation.



- `payload` – A key-value pair list that's used as the input for invalidating subscriptions if the invalidation filter evaluates to `true` against their values.

The following example invalidates subscribed and connected clients using the `onUserDelete` subscription when the invalidation filter defined in the subscription resolver evaluates to `true` against the `payload` value.

```
$extensions.invalidateSubscriptions({
  "subscriptionField": "onUserDelete",
  "payload": {
    "group": "Developer"
    "type" : "Full-Time"
  }
})
```

## Resolver Mapping Template Reference for DynamoDB

The AWS AppSync DynamoDB resolver enables you to use [GraphQL](#) to store and retrieve data in existing Amazon DynamoDB tables in your account. This resolver works by enabling you to map an incoming GraphQL request into a DynamoDB call, and then map the DynamoDB response back to GraphQL. This section describes the mapping templates for supported DynamoDB operations.

### Topics

- [GetItem \(p. 321\)](#)
- [PutItem \(p. 323\)](#)
- [UpdateItem \(p. 325\)](#)
- [DeleteItem \(p. 328\)](#)
- [Query \(p. 330\)](#)
- [Scan \(p. 333\)](#)
- [Sync \(p. 336\)](#)
- [BatchGetItem \(p. 338\)](#)
- [BatchDeleteItem \(p. 340\)](#)
- [BatchPutItem \(p. 342\)](#)
- [TransactGetItems \(p. 345\)](#)
- [TransactWriteItems \(p. 347\)](#)
- [Type System \(Request Mapping\) \(p. 352\)](#)
- [Type System \(Response Mapping\) \(p. 356\)](#)
- [Filters \(p. 359\)](#)
- [Condition Expressions \(p. 360\)](#)
- [Transaction Condition Expressions \(p. 368\)](#)

## GetItem

The `GetItem` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `GetItem` request to DynamoDB, and enables you to specify:

- The key of the item in DynamoDB
- Whether to use a consistent read or not

The `GetItem` mapping document has the following structure:

```
{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "consistentRead" : true
}
```

The fields are defined as follows:

**version**

The template definition version. 2017-02-28 and 2018-05-29 are currently supported. This value is required.

**operation**

The DynamoDB operation to perform. To perform the `GetItem` DynamoDB operation, this must be set to `GetItem`. This value is required.

**key**

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see [Type System \(Request Mapping\)](#) (p. 352). This value is required.

**consistentRead**

Whether or not to perform a strongly consistent read with DynamoDB. This is optional, and defaults to `false`.

The item returned from DynamoDB is automatically converted into GraphQL and JSON primitive types, and is available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see [Type System \(Response Mapping\)](#) (p. 356).

For more information about response mapping templates, see [Resolver Mapping Template Overview](#) (p. 264).

## Example

Following is a mapping template for a GraphQL query `getThing(foo: String!, bar: String!)`:

```
{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
    "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
  },
  "consistentRead" : true
}
```

```
}
```

For more information about the DynamoDB `GetItem` API, see the [DynamoDB API documentation](#).

## PutItem

The `PutItem` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `PutItem` request to DynamoDB, and enables you to specify the following:

- The key of the item in DynamoDB
- The full contents of the item (composed of `key` and `attributeValues`)
- Conditions for the operation to succeed

The `PutItem` mapping document has the following structure:

```
{
  "version" : "2018-05-29",
  "operation" : "PutItem",
  "key": {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "attributeValues" : {
    "baz" : ... typed value
  },
  "condition" : {
    ...
  },
  "_version" : 1
}
```

The fields are defined as follows:

### **version**

The template definition version. 2017-02-28 and 2018-05-29 are currently supported. This value is required.

### **operation**

The DynamoDB operation to perform. To perform the `PutItem` DynamoDB operation, this must be set to `PutItem`. This value is required.

### **key**

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see [Type System \(Request Mapping\)](#) (p. 352). This value is required.

### **attributeValues**

The rest of the attributes of the item to be put into DynamoDB. For more information about how to specify a “typed value”, see [Type System \(Request Mapping\)](#) (p. 352). This field is optional.

### **condition**

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `PutItem` request overwrites any existing entry for that item. For more information about conditions, see [Condition Expressions](#) (p. 360). This value is optional.

## **`_version`**

A numeric value that represents the latest known version of an item. This value is optional. This field is used for *Conflict Detection* and is only supported on versioned data sources.

The item written to DynamoDB is automatically converted into GraphQL and JSON primitive types and is available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see [Type System \(Response Mapping\)](#) (p. 356).

For more information about response mapping templates, see [Resolver Mapping Template Overview](#) (p. 264).

## Example 1

Following is a mapping template for a GraphQL mutation `updateThing(foo: String!, bar: String!, name: String!, version: Int!)`.

If no item with the specified key exists, it's created. If an item already exists with the specified key, it's overwritten.

```
{
  "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)
  }
}
```

## Example 2

Following is a mapping template for a GraphQL mutation `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)`.

This example checks to be sure the item currently in DynamoDB has the `version` field set to `expectedVersion`.

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
    "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
  },
  "attributeValues" : {
    "name" : $util.dynamodb.toDynamoDBJson($ctx.args.name),
    #set( $newVersion = $context.arguments.expectedVersion + 1 )
    "version" : $util.dynamodb.toDynamoDBJson($newVersion)
  },
  "condition" : {
    "expression" : "version = :expectedVersion",
    "expressionValues" : {
      ":expectedVersion" : $util.dynamodb.toDynamoDBJson($expectedVersion)
    }
  }
}
```

```
}
```

For more information about the DynamoDB `PutItem` API, see the [DynamoDB API documentation](#).

## UpdateItem

The `UpdateItem` request mapping document enables you to tell the AWS AppSync DynamoDB resolver to make a `UpdateItem` request to DynamoDB, and allows you to specify the following:

- The key of the item in DynamoDB
- An update expression describing how to update the item in DynamoDB
- Conditions for the operation to succeed

The `UpdateItem` mapping document has the following structure:

```
{
  "version" : "2018-05-29",
  "operation" : "UpdateItem",
  "key": {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "update" : {
    "expression" : "someExpression"
    "expressionNames" : {
      "#foo" : "foo"
    },
    "expressionValues" : {
      ":bar" : ... typed value
    }
  },
  "condition" : {
    ...
  },
  "_version" : 1
}
```

The fields are defined as follows:

### version

The template definition version. 2017-02-28 and 2018-05-29 are currently supported. This value is required.

### operation

The DynamoDB operation to perform. To perform the `UpdateItem` DynamoDB operation, this must be set to `UpdateItem`. This value is required.

### key

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about specifying a “typed value”, see [Type System \(Request Mapping\)](#) (p. 352). This value is required.

### update

The update section lets you specify an update expression that describes how to update the item in DynamoDB. For more information about how to write update expressions, see the [DynamoDB UpdateExpressions documentation](#). This section is required.

The update section has three components:

**expression**

The update expression. This value is required.

**expressionNames**

The substitutions for expression attribute *name* placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the *expression*, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the *expression*.

**expressionValues**

The substitutions for expression attribute *value* placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the *expression*, and the value must be a typed value. For more information about how to specify a “typed value”, see [Type System \(Request Mapping\) \(p. 352\)](#). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the *expression*.

**condition**

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `UpdateItem` request updates the existing entry regardless of its current state. For more information about conditions, see [Condition Expressions \(p. 360\)](#). This value is optional.

**\_version**

A numeric value that represents the latest known version of an item. This value is optional. This field is used for *Conflict Detection* and is only supported on versioned data sources.

The item updated in DynamoDB is automatically converted into GraphQL and JSON primitive types and is available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see [Type System \(Response Mapping\) \(p. 356\)](#).

For more information about response mapping templates, see [Resolver Mapping Template Overview \(p. 264\)](#).

## Example 1

Following is a mapping template for the GraphQL mutation `upvote(id: ID!)`.

In this example, an item in DynamoDB has its `upvotes` and `version` fields incremented by 1.

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },
  "update" : {
    "expression" : "ADD #voteField :plusOne, version :plusOne",
    "expressionNames" : {
      "#voteField" : "upvotes"
    },
    "expressionValues" : {
      ":plusOne" : { "N" : 1 }
    }
  }
}
```

```

    }
  }
}

```

## Example 2

Following is a mapping template for a GraphQL mutation `updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)`.

This is a complex example that inspects the arguments and dynamically generates the update expression that only includes the arguments that have been provided by the client. For example, if `title` and `author` are omitted, they are not updated. If an argument is specified but its value is `null`, then that field is deleted from the object in DynamoDB. Finally, the operation has a condition, which verifies whether the item currently in DynamoDB has the `version` field set to `expectedVersion`:

```

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

  "operation" : "UpdateItem",

  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },

  ## Set up some space to keep track of things we're updating **
  #set( $expNames = {} )
  #set( $expValues = {} )
  #set( $expSet = {} )
  #set( $expAdd = {} )
  #set( $expRemove = [] )

  ## Increment "version" by 1 **
  ${expAdd.put("version", ":newVersion")}
  ${expValues.put(":newVersion", { "N" : 1 })}

  ## Iterate through each argument, skipping "id" and "expectedVersion" **
  #foreach( $entry in $context.arguments.entrySet() )
    #if( $entry.key != "id" && $entry.key != "expectedVersion" )
      #if( (!$entry.value) && ("${entry.value}" == "") )
        ## If the argument is set to "null", then remove that attribute from the
        item in DynamoDB **

        #set( $discard = ${expRemove.add("#${entry.key}")} )
        ${expNames.put("#${entry.key}", "${entry.key}")}
      #else
        ## Otherwise set (or update) the attribute on the item in DynamoDB **

        ${expSet.put("#${entry.key}", ":${entry.key}")}
        ${expNames.put("#${entry.key}", "${entry.key}")}

        #if( $entry.key == "ups" || $entry.key == "downs" )
          ${expValues.put(":${entry.key}", { "N" : $entry.value })}
        #else
          ${expValues.put(":${entry.key}", { "S" : "${entry.value}" })}
        #end
      #end
    #end
  #end

  ## Start building the update expression, starting with attributes we're going to SET **
  #set( $expression = "" )
  #if( !$expSet.isEmpty() )
    #set( $expression = "SET" )

```

```

    #foreach( $entry in $expSet.entrySet() )
        #set( $expression = "${expression} ${entry.key} = ${entry.value}" )
        #if ( $foreach.hasNext )
            #set( $expression = "${expression}," )
        #end
    #end
#end

## Continue building the update expression, adding attributes we're going to ADD **
#if( !${expAdd.isEmpty()} )
    #set( $expression = "${expression} ADD" )
    #foreach( $entry in $expAdd.entrySet() )
        #set( $expression = "${expression} ${entry.key} ${entry.value}" )
        #if ( $foreach.hasNext )
            #set( $expression = "${expression}," )
        #end
    #end
#end

## Continue building the update expression, adding attributes we're going to REMOVE **
#if( !${expRemove.isEmpty()} )
    #set( $expression = "${expression} REMOVE" )

    #foreach( $entry in $expRemove )
        #set( $expression = "${expression} ${entry}" )
        #if ( $foreach.hasNext )
            #set( $expression = "${expression}," )
        #end
    #end
#end

## Finally, write the update expression into the document, along with any
expressionNames and expressionValues **
"update" : {
    "expression" : "${expression}"
    #if( !${expNames.isEmpty()} )
        , "expressionNames" : $utils.toJson($expNames)
    #end
    #if( !${expValues.isEmpty()} )
        , "expressionValues" : $utils.toJson($expValues)
    #end
},

"condition" : {
    "expression" : "version = :expectedVersion",
    "expressionValues" : {
        ":expectedVersion" : $util.dynamodb.toDynamoDBJson($ctx.args.expectedVersion)
    }
}
}

```

For more information about the DynamoDB UpdateItem API, see the [DynamoDB API documentation](#).

## DeleteItem

The DeleteItem request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a DeleteItem request to DynamoDB, and enables you to specify the following:

- The key of the item in DynamoDB
- Conditions for the operation to succeed

The DeleteItem mapping document has the following structure:



```
{
  "version" : "2018-05-29",
  "operation" : "DeleteItem",
  "key": {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "condition" : {
    ...
  },
  "_version" : 1
}
```

The fields are defined as follows:

**version**

The template definition version. 2017-02-28 and 2018-05-29 are currently supported. This value is required.

**operation**

The DynamoDB operation to perform. To perform the `DeleteItem` DynamoDB operation, this must be set to `DeleteItem`. This value is required.

**key**

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about specifying a “typed value”, see [Type System \(Request Mapping\) \(p. 352\)](#). This value is required.

**condition**

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `DeleteItem` request deletes an item regardless of its current state. For more information about conditions, see [Condition Expressions \(p. 360\)](#). This value is optional.

**\_version**

A numeric value that represents the latest known version of an item. This value is optional. This field is used for *Conflict Detection* and is only supported on versioned data sources.

The item deleted from DynamoDB is automatically converted into GraphQL and JSON primitive types and is available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see [Type System \(Response Mapping\) \(p. 356\)](#).

For more information about response mapping templates, see [Resolver Mapping Template Overview \(p. 264\)](#).

## Example 1

Following is a mapping template for a GraphQL mutation `deleteItem(id: ID!)`. If an item exists with this ID, it's deleted.

```
{
  "version" : "2017-02-28",
  "operation" : "DeleteItem",
  "key" : {
```

```

    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  }
}

```

## Example 2

Following is a mapping template for a GraphQL mutation `deleteItem(id: ID!, expectedVersion: Int!)`. If an item exists with this ID, it's deleted, but only if its `version` field set to `expectedVersion`:

```

{
  "version" : "2017-02-28",
  "operation" : "DeleteItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },
  "condition" : {
    "expression" : "attribute_not_exists(id) OR version = :expectedVersion",
    "expressionValues" : {
      ":expectedVersion" : $util.dynamodb.toDynamoDBJson($expectedVersion)
    }
  }
}

```

For more information about the DynamoDB `DeleteItem` API, see the [DynamoDB API documentation](#).

## Query

The `Query` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `Query` request to DynamoDB, and enables you to specify the following:

- Key expression
- Which index to use
- Any additional filter
- How many items to return
- Whether to use consistent reads
- query direction (forward or backward)
- Pagination token

The `Query` mapping document has the following structure:

```

{
  "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",
"filter" : {
    ...
}
}
```

The fields are defined as follows:

**version**

The template definition version. 2017-02-28 and 2018-05-29 are currently supported. This value is required.

**operation**

The DynamoDB operation to perform. To perform the `Query` DynamoDB operation, this must be set to `Query`. This value is required.

**query**

The query section lets you specify a key condition expression that describes which items to retrieve from DynamoDB. For more information about how to write key condition expressions, see the [DynamoDB KeyConditions documentation](#). This section must be specified.

**expression**

The query expression. This field must be specified.

**expressionNames**

The substitutions for expression attribute *name* placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the *expression*, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the *expression*.

**expressionValues**

The substitutions for expression attribute *value* placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the *expression*, and the value must be a typed value. For more information about how to specify a "typed value", see [Type System \(Request Mapping\) \(p. 352\)](#). This value is required. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the *expression*.

**filter**

An additional filter that can be used to filter the results from DynamoDB before they are returned. For more information about filters, see [Filters \(p. 359\)](#). This field is optional.

**index**

The name of the index to query. The DynamoDB query operation allows you to scan on Local Secondary Indexes and Global Secondary Indexes in addition to the primary key index for a hash key. If specified, this tells DynamoDB to query the specified index. If omitted, the primary key index is queried.

**nextToken**

The pagination token to continue a previous query. This would have been obtained from a previous query. This field is optional.

**limit**

The maximum number of items to evaluate (not necessarily the number of matching items). This field is optional.

**scanIndexForward**

A boolean indicating whether to query forwards or backwards. This field is optional, and defaults to `true`.

**consistentRead**

A boolean indicating whether to use consistent reads when querying DynamoDB. This field is optional, and defaults to `false`.

**select**

By default, the AWS AppSync DynamoDB resolver only returns attributes that are projected into the index. If more attributes are required, you can set this field. This field is optional. The supported values are:

**ALL\_ATTRIBUTES**

Returns all of the item attributes from the specified table or index. If you query a local secondary index, DynamoDB fetches the entire item from the parent table for each matching item in the index. If the index is configured to project all item attributes, all of the data can be obtained from the local secondary index and no fetching is required.

**ALL\_PROJECTED\_ATTRIBUTES**

Allowed only when querying an index. Retrieves all attributes that have been projected into the index. If the index is configured to project all attributes, this return value is equivalent to specifying `ALL_ATTRIBUTES`.

The results from DynamoDB are automatically converted into GraphQL and JSON primitive types and are available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see [Type System \(Response Mapping\) \(p. 356\)](#).

For more information about response mapping templates, see [Resolver Mapping Template Overview \(p. 264\)](#).

The results have the following structure:

```
{
  items = [ ... ],
  nextToken = "a pagination token",
  scannedCount = 10
}
```

The fields are defined as follows:

**items**

A list containing the items returned by the DynamoDB query.

**nextToken**

If there might be more results, `nextToken` contains a pagination token that you can use in another request. Note that AWS AppSync encrypts and obfuscates the pagination token returned from

DynamoDB. This prevents your table data from being inadvertently leaked to the caller. Also note that these pagination tokens cannot be used across different resolvers.

#### **scannedCount**

The number of items that matched the query condition expression, before a filter expression (if present) was applied.

## Example

Following is a mapping template for a GraphQL query `getPosts(owner: ID!)`.

In this example, a global secondary index on a table is queried to return all posts owned by the specified ID.

```
{
  "version" : "2017-02-28",
  "operation" : "Query",
  "query" : {
    "expression" : "ownerId = :ownerId",
    "expressionValues" : {
      ":ownerId" : $util.dynamodb.toDynamoDBJson($context.arguments.owner)
    }
  },
  "index" : "owner-index"
}
```

For more information about the DynamoDB Query API, see the [DynamoDB API documentation](#).

## Scan

The Scan request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a Scan request to DynamoDB, and enables you to specify the following:

- A filter to exclude results
- Which index to use
- How many items to return
- Whether to use consistent reads
- Pagination token
- Parallel scans

The Scan mapping document has the following structure:

```
{
  "version" : "2017-02-28",
  "operation" : "Scan",
  "index" : "fooIndex",
  "limit" : 10,
  "consistentRead" : false,
  "nextToken" : "aPaginationToken",
  "totalSegments" : 10,
  "segment" : 1,
  "filter" : {
    ...
  }
}
```

```
}
```

The fields are defined as follows:

**version**

The template definition version. 2017-02-28 and 2018-05-29 are currently supported. This value is required.

**operation**

The DynamoDB operation to perform. To perform the `Scan` DynamoDB operation, this must be set to `Scan`. This value is required.

**filter**

A filter that can be used to filter the results from DynamoDB before they are returned. For more information about filters, see [Filters \(p. 359\)](#). This field is optional.

**index**

The name of the index to query. The DynamoDB query operation allows you to scan on Local Secondary Indexes and Global Secondary Indexes in addition to the primary key index for a hash key. If specified, this tells DynamoDB to query the specified index. If omitted, the primary key index is queried.

**limit**

The maximum number of items to evaluate at a single time. This field is optional.

**consistentRead**

A Boolean that indicates whether to use consistent reads when querying DynamoDB. This field is optional, and defaults to `false`.

**nextToken**

The pagination token to continue a previous query. This would have been obtained from a previous query. This field is optional.

**select**

By default, the AWS AppSync DynamoDB resolver only returns whatever attributes are projected into the index. If more attributes are required, then this field can be set. This field is optional. The supported values are:

**ALL\_ATTRIBUTES**

Returns all of the item attributes from the specified table or index. If you query a local secondary index, DynamoDB fetches the entire item from the parent table for each matching item in the index. If the index is configured to project all item attributes, all of the data can be obtained from the local secondary index and no fetching is required.

**ALL\_PROJECTED\_ATTRIBUTES**

Allowed only when querying an index. Retrieves all attributes that have been projected into the index. If the index is configured to project all attributes, this return value is equivalent to specifying `ALL_ATTRIBUTES`.

**totalSegments**

The number of segments to partition the table by when performing a parallel scan. This field is optional, but must be specified if `segment` is specified.

**segment**

The table segment in this operation when performing a parallel scan. This field is optional, but must be specified if `totalSegments` is specified.

The results returned by the DynamoDB scan are automatically converted into GraphQL and JSON primitive types and is available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see [Type System \(Response Mapping\) \(p. 356\)](#).

For more information about response mapping templates, see [Resolver Mapping Template Overview \(p. 264\)](#).

The results have the following structure:

```
{
  items = [ ... ],
  nextToken = "a pagination token",
  scannedCount = 10
}
```

The fields are defined as follows:

**items**

A list containing the items returned by the DynamoDB scan.

**nextToken**

If there might be more results, `nextToken` contains a pagination token that you can use in another request. AWS AppSync encrypts and obfuscates the pagination token returned from DynamoDB. This prevents your table data from being inadvertently leaked to the caller. Also, these pagination tokens can't be used across different resolvers.

**scannedCount**

The number of items that were retrieved by DynamoDB before a filter expression (if present) was applied.

## Example 1

Following is a mapping template for the GraphQL query: `allPosts`.

In this example, all entries in the table are returned.

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

## Example 2

Following is a mapping template for the GraphQL query: `postsMatching(title: String!)`.

In this example, all entries in the table are returned where the title starts with the `title` argument.

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

```
"operation" : "Scan",
"filter" : {
  "expression" : "begins_with(title, :title)",
  "expressionValues" : {
    ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
  },
}
}
```

For more information about the DynamoDB Scan API, see the [DynamoDB API documentation](#).

## Sync

The Sync request mapping document lets you retrieve all the results from a DynamoDB table and then receive only the data altered since your last query (the delta updates). Sync requests can only be made to versioned DynamoDB data sources. You can specify the following:

- A filter to exclude results
- How many items to return
- Pagination Token
- When your last Sync operation was started

The Sync mapping document has the following structure:

```
{
  "version" : "2018-05-29",
  "operation" : "Sync",
  "limit" : 10,
  "nextToken" : "aPaginationToken",
  "lastSync" : 1550000000000,
  "filter" : {
    ...
  }
}
```

The fields are defined as follows:

### **version**

The template definition version. Only 2018-05-29 is currently supported. This value is required.

### **operation**

The DynamoDB operation to perform. To perform the Sync operation, this must be set to Sync. This value is required.

### **filter**

A filter that can be used to filter the results from DynamoDB before they are returned. For more information about filters, see [Filters \(p. 359\)](#). This field is optional.

### **limit**

The maximum number of items to evaluate at a single time. This field is optional. If omitted, the default limit will be set to 100 items. The maximum value for this field is 1000 items.

### **nextToken**

The pagination token to continue a previous query. This would have been obtained from a previous query. This field is optional.



**lastSync**

The moment, in epoch milliseconds, when the last successful `Sync` operation started. If specified, only items that have changed after `lastSync` are returned. This field is optional, and should only be populated after retrieving all pages from an initial `Sync` operation. If omitted, results from the *Base* table will be returned, otherwise, results from the *Delta* table will be returned.

The results returned by the DynamoDB sync are automatically converted into GraphQL and JSON primitive types and are available in the mapping context (`$context.result`).

For more information about DynamoDB type conversion, see [Type System \(Response Mapping\)](#) (p. 356).

For more information about response mapping templates, see [Resolver Mapping Template Overview](#) (p. 264).

The results have the following structure:

```
{
  items = [ ... ],
  nextToken = "a pagination token",
  scannedCount = 10,
  startedAt = 1550000000000
}
```

The fields are defined as follows:

**items**

A list containing the items returned by the sync.

**nextToken**

If there might be more results, `nextToken` contains a pagination token that you can use in another request. AWS AppSync encrypts and obfuscates the pagination token returned from DynamoDB. This prevents your table data from being inadvertently leaked to the caller. Also, these pagination tokens can't be used across different resolvers.

**scannedCount**

The number of items that were retrieved by DynamoDB before a filter expression (if present) was applied.

**startedAt**

The moment, in epoch milliseconds, when the sync operation started that you can store locally and use in another request as your `lastSync` argument. If a pagination token was included in the request, this value will be the same as the one returned by the request for the first page of results.

## Example 1

Following is a mapping template for the GraphQL query: `syncPosts(nextToken: String, lastSync: AWSTimestamp)`.

In this example, if `lastSync` is omitted, all entries in the base table are returned. If `lastSync` is supplied, only the entries in the delta sync table that have changed since `lastSync` are returned.

```
{
  "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

The `BatchGetItem` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `BatchGetItem` request to DynamoDB to retrieve multiple items, potentially across multiple tables. For this request template, you must specify the following:

- The table names where to retrieve the items from
- The keys of the items to retrieve from each table

The `DynamoDBBatchGetItem` limits apply and **no condition expression** can be provided.

The `BatchGetItem` mapping document has the following structure:

```

{
  "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
    },
    "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
    }
  }
}

```

The fields are defined as follows:

### version

The template definition version. Only 2018-05-29 is supported. This value is required.

### operation

The DynamoDB operation to perform. To perform the `BatchGetItemDynamoDB` operation, this must be set to `BatchGetItem`. This value is required.

### tables

The DynamoDB tables to retrieve the items from. The value is a map where table names are specified as the keys of the map. At least one table must be provided. This `tables` value is required.

### keys

List of DynamoDB keys representing the primary key of the items to retrieve. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see [Type System \(Request Mapping\) \(p. 352\)](#).

### consistentRead

Whether to use a consistent read when executing a *GetItem* operation. This value is optional and defaults to *false*.

Things to remember:

- If an item has not been retrieved from the table, a *null* element appears in the data block for that table.
- Invocation results are sorted per table, based on the order they were provided inside the request mapping template.
- Each *Get* command inside a *BatchGetItem* is atomic, however, a batch can be partially processed. If a batch is partially processed due to an error, the unprocessed keys are returned as part of the invocation result inside the *unprocessedKeys* block.
- *BatchGetItem* is limited to 100 keys.

For the following example request mapping template:

```
{
  "version": "2018-05-29",
  "operation": "BatchGetItem",
  "tables": {
    "authors": [
      {
        "author_id": {
          "S": "a1"
        }
      },
    ],
    "posts": [
      {
        "author_id": {
          "S": "a1"
        },
        "post_id": {
          "S": "p2"
        }
      }
    ],
  },
}
```

The invocation result available in `$ctx.result` is as follows:

```
{
  "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": []
  }
}
```

The `$ctx.error` contains details about the error. The keys **data**, **unprocessedKeys**, and each table key that was provided in the request mapping template are guaranteed to be present in the invocation result. Items that have been deleted appear in the **data** block. Items that haven't been processed are marked as *null* inside the data block and are placed inside the **unprocessedKeys** block.

For a more complete example, follow the DynamoDB Batch tutorial with AppSync here [Tutorial: DynamoDB Batch Resolvers \(p. 112\)](#).

## BatchDeleteItem

The `BatchDeleteItem` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `BatchWriteItem` request to DynamoDB to delete multiple items, potentially across multiple tables. For this request template, you must specify the following:

- The table names where to delete the items from
- The keys of the items to delete from each table

The `DynamoDBBatchWriteItem` limits apply and **no condition expression** can be provided.

The `BatchDeleteItem` mapping document has the following structure:

```
{
  "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
      },
    ],
  }
}

```

The fields are defined as follows:

**version**

The template definition version. Only 2018-05-29 is supported. This value is required.

**operation**

The DynamoDB operation to perform. To perform the `BatchDeleteItem` DynamoDB operation, this must be set to `BatchDeleteItem`. This value is required.

**tables**

The DynamoDB tables to delete the items from. Each table is a list of DynamoDB keys representing the primary key of the items to delete. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see [Type System \(Request Mapping\) \(p. 352\)](#). At least one table must be provided. This `tables` value is required.

Things to remember:

- Contrary to the `DeleteItem` operation, the fully deleted item isn’t returned in the response. Only the passed key is returned.
- If an item has not been deleted from the table, a *null* element appears in the data block for that table.
- Invocation results are sorted per table, based on the order they were provided inside the request mapping template.
- Each delete command inside a `BatchDeleteItem` is atomic. However a batch can be partially processed. If a batch is partially processed due to an error, the unprocessed keys are returned as part of the invocation result inside the *unprocessedKeys* block.
- `BatchDeleteItem` is limited to 25 keys.

For the following example request mapping template:

```

{
  "version": "2018-05-29",
  "operation": "BatchDeleteItem",
  "tables": {
    "authors": [
      {
        "author_id": {
          "S": "a1"
        }
      },
    ],
  }
}

```

```

    "posts": [
      {
        "author_id": {
          "S": "a1"
        },
        "post_id": {
          "S": "p2"
        }
      }
    ],
  }
}

```

The invocation result available in `$ctx.result` is as follows:

```

{
  "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": []
  }
}

```

The `$ctx.error` contains details about the error. The keys **data**, **unprocessedKeys**, and each table key that was provided in the request mapping template are guaranteed to be present in the invocation result. Items that have been deleted are present in the **data** block. Items that haven't been processed are marked as null inside the data block and are placed inside the **unprocessedKeys** block.

For a more complete example, follow the DynamoDB Batch tutorial with AppSync here [Tutorial: DynamoDB Batch Resolvers \(p. 112\)](#).

## BatchPutItem

The `BatchPutItem` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `BatchWriteItem` request to DynamoDB to put multiple items, potentially across multiple tables. For this request template, you must specify the following:

- The table names where to put the items in
- The full items to put in each table

The `DynamoDBBatchWriteItem` limits apply and **no condition expression** can be provided.

The `BatchPutItem` mapping document has the following structure:

```

{

```

```

"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
    }],
  }
}

```

The fields are defined as follows:

**version**

The template definition version. Only 2018-05-29 is supported. This value is required.

**operation**

The DynamoDB operation to perform. To perform the `BatchPutItem` DynamoDB operation, this must be set to `BatchPutItem`. This value is required.

**tables**

The DynamoDB tables to put the items in. Each table entry represents a list of DynamoDB items to insert for this specific table. At least one table must be provided. This value is required.

Things to remember:

- The fully inserted items are returned in the response, if successful.
- If an item hasn't been inserted in the table, a *null* element is displayed in the data block for that table.
- The inserted items are sorted per table, based on the order they were provided inside the request mapping template.
- Each put command inside a `BatchPutItem` is atomic, however, a batch can be partially processed. If a batch is partially processed due to an error, the unprocessed keys are returned as part of the invocation result inside the *unprocessedKeys* block.
- `BatchPutItem` is limited to 25 items.

For the following example request mapping template:

```

{
  "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"
      }
    },
  ],
}
}

```

The invocation result available in `$ctx.result` is as follows:

```

{
  "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": []
  }
}

```

The `$ctx.error` contains details about the error. The keys **data**, **unprocessedItems**, and each table key that was provided in the request mapping template are guaranteed to be present in the invocation result. Items that have been inserted are in the **data** block. Items that haven't been processed are marked as *null* inside the data block and are placed inside the **unprocessedItems** block.

For a more complete example, follow the DynamoDB Batch tutorial with AppSync here [Tutorial: DynamoDB Batch Resolvers \(p. 112\)](#).



## TransactGetItems

The `TransactGetItems` request mapping document lets you to tell the AWS AppSync DynamoDB resolver to make a `TransactGetItems` request to DynamoDB to retrieve multiple items, potentially across multiple tables. For this request template, you must specify the following:

- The table name of each request item where to retrieve the item from
- The key of each request item to retrieve from each table

The DynamoDB `TransactGetItems` limits apply and **no condition expression** can be provided.

The `TransactGetItems` mapping document has the following structure:

```
{
  "version": "2018-05-29",
  "operation": "TransactGetItems",
  "transactItems": [
    ## First request item
    {
      "table": "table1",
      "key": {
        "foo": ... typed value,
        "bar": ... typed value
      }
    },
    ## Second request item
    {
      "table": "table2",
      "key": {
        "foo": ... typed value,
        "bar": ... typed value
      }
    }
  ]
}
```

The fields are defined as follows:

### **version**

The template definition version. Only 2018-05-29 is supported. This value is required.

### **operation**

The DynamoDB operation to perform. To perform the `TransactGetItems` DynamoDB operation, this must be set to `TransactGetItems`. This value is required.

### **transactItems**

The request items to include. The value is an array of request items. At least one request item must be provided. This `transactItems` value is required.

#### **table**

The DynamoDB table to retrieve the item from. The value is a string of the table name. This table value is required.

#### **key**

The DynamoDB key representing the primary key of the item to retrieve. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure.

For more information about how to specify a “typed value”, see [Type System \(Request Mapping\) \(p. 352\)](#).

Things to remember:

- If a transaction succeeds, the order of retrieved items in the `items` block will be the same as the order of request items.
- Transactions are performed in an all-or-nothing way. If any request item causes an error, the whole transaction will not be performed and error details will be returned.
- A request item being unable to be retrieved is not an error. Instead, a `null` element appears in the `items` block in the corresponding position.
- If the error of a transaction is *TransactionCanceledException*, the `cancellationReasons` block will be populated. The order of cancellation reasons in `cancellationReasons` block will be the same as the order of request items.
- `TransactGetItems` is limited to 25 request items.

For the following example request mapping template:

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

If the transaction succeeds and only the first requested item is retrieved, the invocation result available in `$ctx.result` is as follows:

```
{
  "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
}
```

```
}

```

If the transaction fails due to *TransactionCanceledException* caused by the first request item, the invocation result available in `$ctx.result` is as follows:

```
{
  "items": null,
  "cancellationReasons": [
    {
      "type": "Sample error type",
      "message": "Sample error message"
    },
    {
      "type": "None",
      "message": "None"
    }
  ]
}
```

The `$ctx.error` contains details about the error. The keys **items** and **cancellationReasons** are guaranteed to be present in `$ctx.result`.

For a more complete example, follow the DynamoDB Transaction tutorial with AppSync here [Tutorial: DynamoDB Transaction Resolvers \(p. 125\)](#).

## TransactWriteItems

The `TransactWriteItems` request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a `TransactWriteItems` request to DynamoDB to write multiple items, potentially to multiple tables. For this request template, you must specify the following:

- The destination table name of each request item
- The operation of each request item to perform. There are four types of operations that are supported: *PutItem*, *UpdateItem*, *DeleteItem*, and *ConditionCheck*
- The key of each request item to write

The DynamoDB `TransactWriteItems` limits apply.

The `TransactWriteItems` mapping document has the following structure:

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

The fields are defined as follows:

**version**

The template definition version. Only 2018-05-29 is supported. This value is required.

**operation**

The DynamoDB operation to perform. To perform the `TransactWriteItems` DynamoDB operation, this must be set to `TransactWriteItems`. This value is required.

**transactItems**

The request items to include. The value is an array of request items. At least one request item must be provided. This `transactItems` value is required.

For `PutItem`, the fields are defined as follows:

**table**

The destination DynamoDB table. The value is a string of the table name. This `table` value is required.

**operation**

The DynamoDB operation to perform. To perform the `PutItem` DynamoDB operation, this must be set to `PutItem`. This value is required.

**key**

The DynamoDB key representing the primary key of the item to put. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see [Type System \(Request Mapping\) \(p. 352\)](#). This value is required.

**attributeValues**

The rest of the attributes of the item to be put into DynamoDB. For more information about how to specify a “typed value”, see [Type System \(Request Mapping\) \(p. 352\)](#). This field is optional.

**condition**

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `PutItem` request overwrites any existing entry for that item. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see [Transaction Condition Expressions \(p. 368\)](#). This value is optional.

For `UpdateItem`, the fields are defined as follows:

**table**

The DynamoDB table to update. The value is a string of the table name. This `table` value is required.

**operation**

The DynamoDB operation to perform. To perform the `UpdateItem` DynamoDB operation, this must be set to `UpdateItem`. This value is required.

**key**

The DynamoDB key representing the primary key of the item to update. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see [Type System \(Request Mapping\)](#) (p. 352). This value is required.

**update**

The update section lets you specify an update expression that describes how to update the item in DynamoDB. For more information about how to write update expressions, see the [DynamoDB UpdateExpressions documentation](#) . This section is required.

**condition**

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `UpdateItem` request updates the existing entry regardless of its current state. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see [Transaction Condition Expressions](#) (p. 368). This value is optional.

For `DeleteItem`, the fields are defined as follows:

**table**

The DynamoDB table in which to delete the item. The value is a string of the table name. This `table` value is required.

**operation**

The DynamoDB operation to perform. To perform the `DeleteItem` DynamoDB operation, this must be set to `DeleteItem`. This value is required.

**key**

The DynamoDB key representing the primary key of the item to delete. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see [Type System \(Request Mapping\)](#) (p. 352). This value is required.

**condition**

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `DeleteItem` request deletes an item regardless of its current state. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see [Transaction Condition Expressions](#) (p. 368). This value is optional.

For `ConditionCheck`, the fields are defined as follows:

**table**

The DynamoDB table in which to check the condition. The value is a string of the table name. This `table` value is required.

**operation**

The DynamoDB operation to perform. To perform the `ConditionCheck` DynamoDB operation, this must be set to `ConditionCheck`. This value is required.

**key**

The DynamoDB key representing the primary key of the item to condition check. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see [Type System \(Request Mapping\)](#) (p. 352). This value is required.

### condition

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see [Transaction Condition Expressions \(p. 368\)](#). This value is required.

Things to remember:

- Only keys of request items are returned in the response, if successful. The order of keys will be the same as the order of request items.
- Transactions are performed in an all-or-nothing way. If any request item causes an error, the whole transaction will not be performed and error details will be returned.
- No two request items can target the same item. Otherwise they will cause *TransactionCanceledException* error.
- If the error of a transaction is *TransactionCanceledException*, the *cancellationReasons* block will be populated. If a request item's condition check fails **and** you did not specify *returnValuesOnConditionCheckFailure* to be false, the item existing in the table will be retrieved and stored in *item* at the corresponding position of *cancellationReasons* block.
- *TransactWriteItems* is limited to 25 request items.

For the following example request mapping template:

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

If the transaction succeeds, the invocation result available in `$ctx.result` is as follows:

```
{
  "keys": [
    // Key of the PutItem request
    {
      "post_id": "p1",
    },
    // Key of the UpdateItem request
    {
      "author_id": "a1"
    }
  ],
  "cancellationReasons": null
}
```

If the transaction fails due to condition check failure of the PutItem request, the invocation result available in `$ctx.result` is as follows:

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

The `$ctx.error` contains details about the error. The keys **keys** and **cancellationReasons** are guaranteed to be present in `$ctx.result`.

For a more complete example, follow the DynamoDB Transaction tutorial with AppSync here [Tutorial: DynamoDB Transaction Resolvers \(p. 125\)](#).

## Type System (Request Mapping)

When using the AWS AppSync DynamoDB resolver to call your DynamoDB tables, AWS AppSync needs to know the type of each value to use in that call. This is because DynamoDB supports more type primitives than GraphQL or JSON (such as sets and binary data). AWS AppSync needs some hints when translating



between GraphQL and DynamoDB, otherwise it would have to make some assumptions on how data is structured in your table.

For more information about DynamoDB data types, see the [DynamoDB Data Type Descriptors](#) and [Data Types](#) documentation.

A DynamoDB value is represented by a JSON object containing a single key-value pair. The key specifies the DynamoDB type, and the value specifies the value itself. In the following example, the key `S` denotes that the value is a string, and the value `identifier` is the string value itself.

```
{ "S" : "identifier" }
```

Note that the JSON object cannot have more than one key-value pair. If more than one key-value pair is specified, the request mapping document isn't parsed.

A DynamoDB value is used anywhere in a request mapping document where you need to specify a value. Some places where you need to do this include: `key` and `attributeValue` sections, and the `expressionValues` section of expression sections. In the following example, the DynamoDB String value `identifier` is being assigned to the `id` field in a `key` section (perhaps in a `GetItem` request mapping document).

```
"key" : {  
  "id" : { "S" : "identifier" }  
}
```

### Supported Types

AWS AppSync supports the following DynamoDB scalar, document, and set types:

#### String type `s`

A single string value. A DynamoDB String value is denoted by:

```
{ "S" : "some string" }
```

An example usage is:

```
"key" : {  
  "id" : { "S" : "some string" }  
}
```

#### String set type `ss`

A set of string values. A DynamoDB String Set value is denoted by:

```
{ "SS" : [ "first value", "second value", ... ] }
```

An example usage is:

```
"attributeValues" : {  
  "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }  
}
```

#### Number type `n`

A single numeric value. A DynamoDB Number value is denoted by:

```
{ "N" : 1234 }
```

An example usage is:

```
"expressionValues" : {  
  ":expectedVersion" : { "N" : 1 }  
}
```

### Number set type NS

A set of number values. A DynamoDB Number Set value is denoted by:

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

An example usage is:

```
"attributeValues" : {  
  "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }  
}
```

### Binary type B

A binary value. A DynamoDB Binary value is denoted by:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

Note that the value is actually a string, where the string is the base64-encoded representation of the binary data. AWS AppSync decodes this string back into its binary value before sending it to DynamoDB. AWS AppSync uses the base64 decoding scheme as defined by RFC 2045: any character that isn't in the base64 alphabet is ignored.

An example usage is:

```
"attributeValues" : {  
  "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }  
}
```

### Binary set type BS

A set of binary values. A DynamoDB Binary Set value is denoted by:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

Note that the value is actually a string, where the string is the base64-encoded representation of the binary data. AWS AppSync decodes this string back into its binary value before sending it to DynamoDB. AWS AppSync uses the base64 decoding scheme as defined by RFC 2045: any character that is not in the base64 alphabet is ignored.

An example usage is:

```
"attributeValues" : {  
  "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }  
}
```

### Boolean type **BOOL**

A Boolean value. A DynamoDB Boolean value is denoted by:

```
{ "BOOL" : true }
```

Note that only `true` and `false` are valid values.

An example usage is:

```
"attributeValues" : {  
  "orderComplete" : { "BOOL" : false }  
}
```

### List type **L**

A list of any other supported DynamoDB value. A DynamoDB List value is denoted by:

```
{ "L" : [ ... ] }
```

Note that the value is a compound value, where the list can contain zero or more of any supported DynamoDB value (including other lists). The list can also contain a mix of different types.

An example usage is:

```
{ "L" : [  
  { "S" : "A string value" },  
  { "N" : 1 },  
  { "SS" : [ "Another string value", "Even more string values!" ] }  
]
```

### Map type **M**

Representing an unordered collection of key-value pairs of other supported DynamoDB values. A DynamoDB Map value is denoted by:

```
{ "M" : { ... } }
```

Note that a map can contain zero or more key-value pairs. The key must be a string, and the value can be any supported DynamoDB value (including other maps). The map can also contain a mix of different types.

An example usage is:

```
{ "M" : {  
  "someString" : { "S" : "A string value" },  
  "someNumber" : { "N" : 1 },  
  "stringSet" : { "SS" : [ "Another string value", "Even more string values!" ] }  
}
```

### Null type **NULL**

A null value. A DynamoDB Null value is denoted by:

```
{ "NULL" : null }
```

An example usage is:

```
"attributeValues" : {  
  "phoneNumbers" : { "NULL" : null }  
}
```

For more information about each type, see the [DynamoDB documentation](#) .

## Type System (Response Mapping)

When receiving a response from DynamoDB, AWS AppSync automatically converts it into GraphQL and JSON primitive types. Each attribute in DynamoDB is decoded and returned in the response mapping context.

For example, if DynamoDB returns the following:

```
{  
  "id" : { "S" : "1234" },  
  "name" : { "S" : "Nadia" },  
  "age" : { "N" : 25 }  
}
```

Then the AWS AppSync DynamoDB resolver converts it into GraphQL and JSON types as:

```
{  
  "id" : "1234",  
  "name" : "Nadia",  
  "age" : 25  
}
```

This section explains how AWS AppSync converts the following DynamoDB scalar, document, and set types:

### String type s

A single string value. A DynamoDB String value is returned as a string.

For example, if DynamoDB returned the following DynamoDB String value:

```
{ "S" : "some string" }
```

AWS AppSync converts it to a string:

```
"some string"
```

### String set type ss

A set of string values. A DynamoDB String Set value is returned as a list of strings.

For example, if DynamoDB returned the following DynamoDB String Set value:

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync converts it to a list of strings:

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

### Number type **N**

A single numeric value. A DynamoDB Number value is returned as a number.

For example, if DynamoDB returned the following DynamoDB Number value:

```
{ "N" : 1234 }
```

AWS AppSync converts it to a number:

```
1234
```

### Number set type **NS**

A set of number values. A DynamoDB Number Set value is returned as a list of numbers.

For example, if DynamoDB returned the following DynamoDB Number Set value:

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync converts it to a list of numbers:

```
[ 67.8, 12.2, 70 ]
```

### Binary type **B**

A binary value. A DynamoDB Binary value is returned as a string containing the base64 representation of that value.

For example, if DynamoDB returned the following DynamoDB Binary value:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync converts it to a string containing the base64 representation of the value:

```
"SGVsbG8sIFdvcmxkIQo="
```

Note that the binary data is encoded in the base64 encoding scheme as specified in [RFC 4648](#) and [RFC 2045](#).

### Binary set type **BS**

A set of binary values. A DynamoDB Binary Set value is returned as a list of strings containing the base64 representation of the values.

For example, if DynamoDB returned the following DynamoDB Binary Set value:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync converts it to a list of strings containing the base64 representation of the values:

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

Note that the binary data is encoded in the base64 encoding scheme as specified in [RFC 4648](#) and [RFC 2045](#).

### Boolean type **BOOL**

A Boolean value. A DynamoDB Boolean value is returned as a Boolean.

For example, if DynamoDB returned the following DynamoDB Boolean value:

```
{ "BOOL" : true }
```

AWS AppSync converts it to a Boolean:

```
true
```

### List type **L**

A list of any other supported DynamoDB value. A DynamoDB List value is returned as a list of values, where each inner value is also converted.

For example, if DynamoDB returned the following DynamoDB List value:

```
{ "L" : [
  { "S" : "A string value" },
  { "N" : 1 },
  { "SS" : [ "Another string value", "Even more string values!" ] }
]
```

AWS AppSync converts it to a list of converted values:

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

### Map type **M**

A key/value collection of any other supported DynamoDB value. A DynamoDB Map value is returned as a JSON object, where each key/value is also converted.

For example, if DynamoDB returned the following DynamoDB Map value:

```
{ "M" : {
  "someString" : { "S" : "A string value" },
  "someNumber" : { "N" : 1 },
  "stringSet" : { "SS" : [ "Another string value", "Even more string values!" ] }
}
```

AWS AppSync converts it to a JSON object:

```
{
  "someString" : "A string value",
  "someNumber" : 1,
  "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

### Null type **NULL**

A null value.

For example, if DynamoDB returned the following DynamoDB Null value:

```
{ "NULL" : null }
```

AWS AppSync converts it to a null:

```
null
```

## Filters

When querying objects in DynamoDB using the `query` and `scan` operations, you can optionally specify a `filter` that evaluates the results and returns only the desired values.

The filter mapping section of a `query` or `scan` mapping document has the following structure:

```
"filter" : {
  "expression" : "filter expression"
  "expressionNames" : {
    "#name" : "name",
  },
  "expressionValues" : {
    ":value" : ... typed value
  },
}
```

The fields are defined as follows:

### **expression**

The query expression. For more information about how to write filter expressions, see the [DynamoDB QueryFilter](#) and [DynamoDB ScanFilter](#) documentation. This field must be specified.

### **expressionNames**

The substitutions for expression attribute *name* placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the *expression*. The value must be a string that corresponds to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the *expression*.

### **expressionValues**

The substitutions for expression attribute *value* placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the *expression*, and the value must be a typed value. For more information about how to specify a "typed value", see [Type System \(Request Mapping\)](#) (p. 352). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the *expression*.

## Example

Following is a filter section for a mapping template, where entries retrieved from DynamoDB are only returned if the title starts with the `title` argument.

```
"filter" : {
  "expression" : "begins_with(#title, :title)",
  "expressionNames" : {
    "#title" : "title"
  },
  "expressionValues" : {
    ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
  }
}
```

## Condition Expressions

When you mutate objects in DynamoDB by using the `PutItem`, `UpdateItem`, and `DeleteItem` DynamoDB operations, you can optionally specify a condition expression that controls whether the request should succeed or not, based on the state of the object already in DynamoDB before the operation is performed.

The AWS AppSync DynamoDB resolver allows a condition expression to be specified in `PutItem`, `UpdateItem`, and `DeleteItem` request mapping documents, and also a strategy to follow if the condition fails and the object was not updated.

### Example 1

The following `PutItem` mapping document doesn't have a condition expression. As a result, it puts an item in DynamoDB even if an item with the same key already exists, thereby overwriting the existing item.

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "1" }
  }
}
```

### Example 2

The following `PutItem` mapping document does have a condition expression that allows the operation succeed only if an item with the same key does *not* exist in DynamoDB.

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "1" }
  },
  "condition" : {
    "expression" : "attribute_not_exists(id)"
  }
}
```



By default, if the condition check fails, the AWS AppSync DynamoDB resolver returns an error for the mutation and the current value of the object in DynamoDB in a `data` field in the `error` section of the GraphQL response. However, the AWS AppSync DynamoDB resolver offers some additional features to help developers handle some common edge cases:

- If AWS AppSync DynamoDB resolver can determine that the current value in DynamoDB matches the desired result, it treats the operation as if it succeeded anyway.
- Instead of returning an error, you can configure the resolver to invoke a custom Lambda function to decide how the AWS AppSync DynamoDB resolver should handle the failure.

These are described in greater detail in the [Handling a Condition Check Failure \(p. 362\)](#) section.

For more information about DynamoDB conditions expressions, see the [DynamoDB ConditionExpressions documentation](#).

## Specifying a Condition

The `PutItem`, `UpdateItem`, and `DeleteItem` request mapping documents all allow an optional `condition` section to be specified. If omitted, no condition check is made. If specified, the condition must be true for the operation to succeed.

A `condition` section has the following structure:

```
"condition" : {
  "expression" : "someExpression"
  "expressionNames" : {
    "#foo" : "foo"
  },
  "expressionValues" : {
    ":bar" : ... typed value
  },
  "equalsIgnore" : [ "version" ],
  "consistentRead" : true,
  "conditionalCheckFailedHandler" : {
    "strategy" : "Custom",
    "lambdaArn" : "arn:..."
  }
}
```

The following fields specify the condition:

### **expression**

The update expression itself. For more information about how to write condition expressions, see the [DynamoDB ConditionExpressions documentation](#). This field must be specified.

### **expressionNames**

The substitutions for expression attribute name placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the *expression*, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the *expression*.

### **expressionValues**

The substitutions for expression attribute value placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the *expression*, and the value must be a typed value. For more information about how to specify a “typed value”, see [Type System \(request mapping\)](#).

This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the expression.

The remaining fields tell the AWS AppSync DynamoDB resolver how to handle a condition check failure:

**`equalsIgnore`**

When a condition check fails when using the `PutItem` operation, the AWS AppSync DynamoDB resolver compares the item currently in DynamoDB against the item it tried to write. If they are the same, it treats the operation as if it succeeded anyway. You can use the `equalsIgnore` field to specify a list of attributes that AWS AppSync should ignore when performing that comparison. For example, if the only difference was a `version` attribute, it treats the operation as if it succeeded. This field is optional.

**`consistentRead`**

When a condition check fails, AWS AppSync gets the current value of the item from DynamoDB using a strongly consistent read. You can use this field to tell the AWS AppSync DynamoDB resolver to use an eventually consistent read instead. This field is optional, and defaults to `true`.

**`conditionalCheckFailedHandler`**

This section allows you to specify how the AWS AppSync DynamoDB resolver treats a condition check failure after it has compared the current value in DynamoDB against the expected result. This section is optional. If omitted, it defaults to a strategy of `Reject`.

**`strategy`**

The strategy the AWS AppSync DynamoDB resolver takes after it has compared the current value in DynamoDB against the expected result. This field is required and has the following possible values:

**`Reject`**

The mutation fails, and an error for the mutation and the current value of the object in DynamoDB in a `data` field in the `error` section of the GraphQL response.

**`Custom`**

The AWS AppSync DynamoDB resolver invokes a custom Lambda function to decide how to handle the condition check failure. When the `strategy` is set to `Custom`, the `lambdaArn` field must contain the ARN of the Lambda function to invoke.

**`lambdaArn`**

The ARN of the Lambda function to invoke that determines how the AWS AppSync DynamoDB resolver should handle the condition check failure. This field must only be specified when `strategy` is set to `Custom`. For more information about how to use this feature, see [Handling a Condition Check Failure \(p. 362\)](#).

## Handling a Condition Check Failure

By default, when a condition check fails, the AWS AppSync DynamoDB resolver returns an error for the mutation and the current value of the object in DynamoDB in a `data` field in the `error` section of the GraphQL response. However, the AWS AppSync DynamoDB resolver offers some additional features to help developers handle some common edge cases:

- If AWS AppSync DynamoDB resolver can determine that the current value in DynamoDB matches the desired result, it treats the operation as if it succeeded anyway.
- Instead of returning an error, you can configure the resolver to invoke a custom Lambda function to decide how the AWS AppSync DynamoDB resolver should handle the failure.

The flowchart for this process is:

## Checking for the Desired Result

When the condition check fails, the AWS AppSync DynamoDB resolver performs a `GetItem` DynamoDB request to get the current value of the item from DynamoDB. By default, it uses a strongly consistent read, however this can be configured using the `consistentRead` field in the `condition` block and compare it against the expected result:

- For the `PutItem` operation, the AWS AppSync DynamoDB resolver compares the current value against the one it attempted to write, excluding any attributes listed in `equalsIgnore` from the comparison. If the items are the same, it treats the operation as successful and returns the item that was retrieved from DynamoDB. Otherwise, it follows the configured strategy.

For example, if the `PutItem` request mapping document looked like the following:

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

And the item currently in DynamoDB looked like the following:

```
{
  "id" : { "S" : "1" },
  "name" : { "S" : "Steve" },
  "version" : { "N" : 8 }
}
```

The AWS AppSync DynamoDB resolver would compare the item it tried to write against the current value, see that the only difference was the `version` field, but because it's configured to ignore the `version` field, it treats the operation as successful and returns the item that was retrieved from DynamoDB.

- For the `DeleteItem` operation, the AWS AppSync DynamoDB resolver checks to verify that an item was returned from DynamoDB. If no item was returned, it treats the operation as successful. Otherwise, it follows the configured strategy.
- For the `UpdateItem` operation, the AWS AppSync DynamoDB resolver does not have enough information to determine if the item currently in DynamoDB matches the expected result, and therefore follows the configured strategy.

If the current state of the object in DynamoDB is different from the expected result, the AWS AppSync DynamoDB resolver follows the configured strategy, to either reject the mutation or invoke a Lambda function to determine what to do next.

## Following the “Reject” Strategy

When following the `Reject` strategy, the AWS AppSync DynamoDB resolver returns an error for the mutation, and the current value of the object in DynamoDB is also returned in a `data` field in the `error` section of the GraphQL response. The item returned from DynamoDB is put through the response mapping template to translate it into a format the client expects, and it is filtered by the selection set.

For example, given the following mutation request:

```
mutation {
  updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
    Name
    theVersion
  }
}
```

If the item returned from DynamoDB looks like the following:

```
{
  "id" : { "S" : "1" },
  "name" : { "S" : "Steve" },
  "version" : { "N" : 8 }
}
```

And the response mapping template looks like the following:

```
{
  "id" : $util.toJson($context.result.id),
  "Name" : $util.toJson($context.result.name),
  "theVersion" : $util.toJson($context.result.version)
}
```

The GraphQL response looks like the following:

```
{
  "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
      },
      ...
    }
  ]
}
```

Also, if any fields in the returned object are filled by other resolvers and the mutation had succeeded, they won't be resolved when the object is returned in the `error` section.

## Following the “Custom” Strategy

When following the `Custom` strategy, the AWS AppSync DynamoDB resolver invokes a Lambda function to decide what to do next. The Lambda function chooses one of the following options:

- **reject** the mutation. This tells the AWS AppSync DynamoDB resolver to behave as if the configured strategy was `Reject`, returning an error for the mutation and the current value of the object in DynamoDB as described in the previous section.
- **discard** the mutation. This tells the AWS AppSync DynamoDB resolver to silently ignore the condition check failure and returns the value in DynamoDB.
- **retry** the mutation. This tells the AWS AppSync DynamoDB resolver to retry the mutation with a new request mapping document.

### The Lambda invocation request

The AWS AppSync DynamoDB resolver invokes the Lambda function specified in the `lambdaArn`. It uses the same `service-role-arn` configured on the data source. The payload of the invocation has the following structure:

```
{
  "arguments": { ... },
  "requestMapping": {... },
  "currentValue": { ... },
  "resolver": { ... },
  "identity": { ... }
}
```

The fields are defined as follows:

#### **arguments**

The arguments from the GraphQL mutation. This is the same as the arguments available to the request mapping document in `$context.arguments`.

#### **requestMapping**

The request mapping document for this operation.

#### **currentValue**

The current value of the object in DynamoDB.

#### **resolver**

Information about the AWS AppSync resolver.

#### **identity**

Information about the caller. This is the same as the identity information available to the request mapping document in `$context.identity`.

A full example of the payload:

```
{
  "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": "AIDAAAAAAAAAAAAAAAAAAAA",
    "userArn": "arn:aws:iam::123456789012:user/appsync"
}
}

```

### The Lambda Invocation Response

The Lambda function can inspect the invocation payload and apply any business logic to decide how the AWS AppSync DynamoDB resolver should handle the failure. There are three options for handling the condition check failure:

- **reject** the mutation. The response payload for this option must have this structure:

```

{
    "action": "reject"
}

```

This tells the AWS AppSync DynamoDB resolver to behave as if the configured strategy was `Reject`, returning an error for the mutation and the current value of the object in DynamoDB, as described in the section above.

- **discard** the mutation. The response payload for this option must have this structure:

```

{
    "action": "discard"
}

```

This tells the AWS AppSync DynamoDB resolver to silently ignore the condition check failure and returns the value in DynamoDB.

- **retry** the mutation. The response payload for this option must have this structure:

```

{
    "action": "retry",
    "retryMapping": { ... }
}

```

This tells the AWS AppSync DynamoDB resolver to retry the mutation with a new request mapping document. The structure of the `retryMapping` section depends on the DynamoDB operation, and is a subset of the full request mapping document for that operation.

For `PutItem`, the `retryMapping` section has the following structure. For a description of the `attributeValues` field, see [PutItem \(p. 323\)](#).

```
{
  "attributeValues": { ... },
  "condition": {
    "equalsIgnore" = [ ... ],
    "consistentRead" = true
  }
}
```

For `UpdateItem`, the `retryMapping` section has the following structure. For a description of the `update` section, see [UpdateItem \(p. 325\)](#).

```
{
  "update" : {
    "expression" : "someExpression"
    "expressionNames" : {
      "#foo" : "foo"
    },
    "expressionValues" : {
      ":bar" : ... typed value
    }
  },
  "condition": {
    "consistentRead" = true
  }
}
```

For `DeleteItem`, the `retryMapping` section has the following structure.

```
{
  "condition": {
    "consistentRead" = true
  }
}
```

There is no way to specify a different operation or key to work on. The AWS AppSync DynamoDB resolver only allows retries of the same operation on the same object. Also, the `condition` section doesn't allow a `conditionalCheckFailedHandler` to be specified. If the retry fails, the AWS AppSync DynamoDB resolver follows the `Reject` strategy.

Here is an example Lambda function to deal with a failed `PutItem` request. The business logic looks at who made the call. If it was made by `jeffTheAdmin`, it retries the request, updating the `version` and `expectedVersion` from the item currently in DynamoDB. Otherwise, it rejects the mutation.

```
exports.handler = (event, context, callback) => {
  console.log("Event: " + JSON.stringify(event));

  // Business logic goes here.

  var response;
  if ( event.identity.user == "jeffTheAdmin" ) {
    response = {
```

```
        "action" : "retry",
        "retryMapping" : {
            "attributeValues" : event.requestMapping.attributeValues,
            "condition" : {
                "expression" : event.requestMapping.condition.expression,
                "expressionValues" : event.requestMapping.condition.expressionValues
            }
        }
    }
    response.retryMapping.attributeValues.version = { "N" :
event.currentValue.version.N + 1 }
    response.retryMapping.condition.expressionValues[':expectedVersion'] =
event.currentValue.version

    } else {
        response = { "action" : "reject" }
    }

    console.log("Response: " + JSON.stringify(response))
    callback(null, response)
};
```

## Transaction Condition Expressions

Transaction condition expressions are available in request mapping templates of all four types of operations in `TransactWriteItems`, namely, `PutItem`, `DeleteItem`, `UpdateItem`, and `ConditionCheck`.

For `PutItem`, `DeleteItem`, and `UpdateItem`, transaction condition expression is optional. For `ConditionCheck`, transaction condition expression is required.

### Example 1

The following transactional `DeleteItem` mapping document does not have a condition expression. As a result, it deletes the item in DynamoDB.

```
{
  "version": "2018-05-29",
  "operation": "TransactWriteItems",
  "transactItems": [
    {
      "table": "posts",
      "operation": "DeleteItem",
      "key": {
        "id": { "S" : "1" }
      }
    }
  ]
}
```

### Example 2

The following transactional `DeleteItem` mapping document does have a transaction condition expression that allows the operation succeed only if the author of that post equals certain name.

```
{
  "version": "2018-05-29",
  "operation": "TransactWriteItems",
  "transactItems": [
    {
```



```
    "table": "posts",
    "operation": "DeleteItem",
    "key": {
      "id": { "S" : "1" }
    }
    "condition": {
      "expression": "author = :author",
      "expressionValues": {
        ":author": { "S" : "Chunyan" }
      }
    }
  }
}
```

If the condition check fails, it will cause `TransactionCanceledException` and the error detail will be returned in `$ctx.result.cancellationReasons`. Note that by default, the old item in DynamoDB that made condition check fail will be returned in `$ctx.result.cancellationReasons`

## Specifying a Condition

The `PutItem`, `UpdateItem`, and `DeleteItem` request mapping documents all allow an optional condition section to be specified. If omitted, no condition check is made. If specified, the condition must be true for the operation to succeed. The `ConditionCheck` must have a condition section to be specified. The condition must be true for the whole transaction to succeed.

A condition section has the following structure:

```
"condition": {
  "expression": "someExpression",
  "expressionNames": {
    "#foo": "foo"
  },
  "expressionValues": {
    ":bar": ... typed value
  },
  "returnValuesOnConditionCheckFailure": false
}
```

The following fields specify the condition:

### **expression**

The update expression itself. For more information about how to write condition expressions, see the [DynamoDB ConditionExpressions documentation](#) . This field must be specified.

### **expressionNames**

The substitutions for expression attribute name placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the *expression*, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the *expression*.

### **expressionValues**

The substitutions for expression attribute value placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the expression, and the value must be a typed value. For more information about how to specify a “typed value”, see Type System (request mapping). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the expression.

### **returnValuesOnConditionCheckFailure**

Specify whether to retrieve the item in DynamoDB back when a condition check fails. The retrieved item will be in `$ctx.result.cancellationReasons[$index].item`, where `$index` is the index of the request item that failed the condition check. Defaults to be true.

## Resolver mapping template reference for RDS

The AWS AppSync RDS resolver mapping templates allow developers to send SQL queries to a Data API for Amazon Aurora Serverless and get back the result of these queries.

### Request mapping template

The RDS request mapping template is fairly simple:

```
{
  "version": "2018-05-29",
  "statements": [],
  "variableMap": {}
}
```

Here is the JSON schema representation of the RDS request mapping template, once resolved.

```
{
  "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"
    }
  }
}

```

The following is an example of the request mapping template with a static query:

```

{
  "version": "2018-05-29",
  "statements": [
    "select title, isbn13 from BOOKS where author = 'Mark Twain'"
  ]
}

```

## Version

Common to all request mapping templates, the version field defines the version that the template uses. The version field is required. The value "2018-05-29" is the only version supported for the Amazon RDS mapping templates.

```
"version": "2018-05-29"
```

## Statements

The statements array is a placeholder for the developer provided queries. Currently, up to two queries per request mapping template are supported. The following is possible:

```

{
  "version": "2018-05-29",
  "statements": [
    $util.toJson("insert into BOOKS VALUES ('$ctx.args.newBook.author',
    '$ctx.args.newBook.title', '$ctx.args.newBook.ISBN13')"),
    $util.toJson("select * from BOOKS WHERE isbn13 = '$ctx.args.newBook.isbn13'")
  ]
}

```

AWS AppSync supports up to two statements per request mapping template.

## VariableMap

The variableMap is an optional field which contains aliases that can be used to make the SQL statements shorter and more readable. For example the following is possible:

```

{
  "version": "2018-05-29",
  "statements": [
    "insert into BOOKS VALUES (:AUTHOR, :TITLE, :ISBN13)",
    "select * from BOOKS WHERE isbn13 = :ISBN13"
  ]
}

```

```
    ],  
    "variableMap": {  
      "AUTHOR": $util.toJson($ctx.args.newBook.author),  
      "TITLE": $util.toJson($ctx.args.newBook.title),  
      "ISBN13": $util.toJson($ctx.args.newBook.isbn13)  
    }  
  }  
}
```

AWS AppSync will use the variable map value to construct the queries that are sent to the Amazon Aurora Serverless Data API. For example the two queries that AWS AppSync will send to Amazon RDS will be (assuming "\$ctx.args.newBook.author"='Mark Twain', "\$ctx.args.newBook.title"='Adventures of Huckleberry Finn' and "\$ctx.args.newBook.isbn13"='978-1948132817'):

```
INSERT INTO BOOKS VALUES ('Mark Twain', 'Adventures of Huckleberry Finn',  
  '978-1948132817');
```

and

```
SELECT from BOOKS where isbn13='978-1948132817';
```

## Resolver Mapping Template Reference for OpenSearch

The AWS AppSync resolver for Amazon OpenSearch Service enables you to use GraphQL to store and retrieve data in existing OpenSearch Service domains in your account. This resolver works by allowing you to map an incoming GraphQL request into an OpenSearch Service request, and then map the OpenSearch Service response back to GraphQL. This section describes the mapping templates for the supported OpenSearch Service operations.

### Request Mapping Template

Most OpenSearch Service request mapping templates have a common structure where just a few pieces change. The following example runs a search against an OpenSearch Service domain, where documents are of type `post` and are indexed under `id`. The search parameters are defined in the `body` section, with many of the common query clauses being defined in the `query` field. This example will search for documents containing "Nadia", or "Bailey", or both, in the `author` field of a document:

```
{  
  "version": "2017-02-28",  
  "operation": "GET",  
  "path": "/id/post/_search",  
  "params": {  
    "headers": {},  
    "queryString": {},  
    "body": {  
      "from": 0,  
      "size": 50,  
      "query": {  
        "bool": {  
          "should": [  
            { "match": { "author": "Nadia" } },  
            { "match": { "author": "Bailey" } }  
          ]  
        }  
      }  
    }  
  }  
}
```

```
}
  }
}
}
```

## Response Mapping Template

As with other data sources, OpenSearch Service sends a response to AWS AppSync that needs to be converted to GraphQL. .

Most GraphQL queries are looking for the `_source` field from an OpenSearch Service response. Because you can do searches to return either an individual document or a list of documents, there are two common response mapping templates used in OpenSearch Service:

### List of Results

```
[
  #foreach($entry in $context.result.hits.hits)
    #if( $velocityCount > 1 ) , #end
    $utils.toJson($entry.get("_source"))
  #end
]
```

### Individual Item

```
$utils.toJson($context.result.get("_source"))
```

## operation field

(REQUEST Mapping Template only)

HTTP method or verb (GET, POST, PUT, HEAD or DELETE) that AWS AppSync sends to the OpenSearch Service domain. Both the key and the value must be a string.

```
"operation" : "PUT"
```

## path field

(REQUEST Mapping Template only)

The search path for an OpenSearch Service request from AWS AppSync. This forms a URL for the operation's HTTP verb. Both the key and the value must be strings.

```
"path" : "/indexname/type"
"path" : "/indexname/type/_search"
```

When the mapping template is evaluated, this path is sent as part of the HTTP request, including the OpenSearch Service domain. For example, the previous example might translate to:

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

## params field

(REQUEST Mapping Template only)

Used to specify what action your search performs, most commonly by setting the **query** value inside of the **body**. However, there are several other capabilities that can be configured, such as the formatting of responses.

- **headers**

The header information, as key-value pairs. Both the key and the value must be strings. For example:

```
"headers" : {  
  "Content-Type" : "application/json"  
}
```

**Note:** AWS AppSync currently supports only JSON as a Content-Type.

- **queryString**

Key-value pairs that specify common options, such as code formatting for JSON responses. Both the key and the value must be a string. For example, if you want to get pretty-formatted JSON, you would use:

```
"queryString" : {  
  "pretty" : "true"  
}
```

- **body**

This is the main part of your request, allowing AWS AppSync to craft a well-formed search request to your OpenSearch Service domain. The key must be a string comprised of an object. A couple of demonstrations are shown below.

### Example 1

Return all documents with a city matching "seattle":

```
"body":{  
  "from":0,  
  "size":50,  
  "query" : {  
    "match" : {  
      "city" : "seattle"  
    }  
  }  
}
```

### Example 2

Return all documents matching "washington" as the city or the state:

```
"body":{  
  "from":0,  
  "size":50,  
  "query" : {  
    "multi_match" : {  
      "query" : "washington",  
      "fields" : ["city", "state"]  
    }  
  }  
}
```

```
}
}
}
```

## Passing Variables

(REQUEST Mapping Template only)

You can also pass variables as part of evaluation in the VTL statement. For example, suppose you had a GraphQL query such as the following:

```
query {
  searchForState(state: "washington"){
    ...
  }
}
```

The mapping template could take the state as an argument:

```
"body":{
  "from":0,
  "size":50,
  "query" : {
    "multi_match" : {
      "query" : "$context.arguments.state",
      "fields" : ["city", "state"]
    }
  }
}
```

For a list of utilities you can include in the VTL, see [Access Request Headers \(p. 282\)](#).

## Resolver mapping template reference for Lambda

You can use the AWS AppSync resolver mapping templates for AWS Lambda to shape requests from AWS AppSync to Lambda functions located in your account, and responses from your Lambda functions back to AWS AppSync. You can also use mapping templates to give hints to AWS AppSync about the nature of the operation to be invoked. This section describes the different mapping templates for the supported Lambda operations.

### Request mapping template

The Lambda request mapping template is fairly simple and allows as much context information as possible to pass to your Lambda function.

```
{
  "version": string,
  "operation": Invoke|BatchInvoke,
  "payload": any type
}
```

Here is the JSON schema representation of the Lambda request mapping template, when resolved.

```
{
  "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": {}
    },
    "required": [
      "version",
      "operation"
    ],
    "additionalProperties": false
  }
}
```

Here is an example where we pass the `field` value and the GraphQL field arguments from the context.

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $util.toJson($context.arguments)
  }
}
```

The entire mapping document is passed as input to your Lambda function, so that the previous example would now look like the following:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "id": "postId1"
    }
  }
}
```

## Version

Common to all request mapping templates, `version` defines the version that the template uses. `version` is required.



```
"version": "2018-05-29"
```

## Operation

The Lambda data source lets you define two operations: `Invoke` and `BatchInvoke`. The `Invoke` operation lets AWS AppSync know to call your Lambda function for every GraphQL field resolver. `BatchInvoke` instructs AWS AppSync to batch requests for the current GraphQL field.

`operation` is required.

For `Invoke`, the resolved request mapping template exactly matches the input payload of the Lambda function. So the following example template:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "arguments": $util.toJson($context.arguments)
  }
}
```

is resolved and passed to the Lambda function, as follows:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

For `BatchInvoke`, the mapping template is applied for every field resolver in the batch. For conciseness, AWS AppSync merges all the resolved mapping template `payload` values into a list under a single object matching the mapping template.

The following example template shows the merge:

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": $util.toJson($context)
}
```

This template is resolved into the following mapping document:

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

where each element of the `payload` list corresponds to a single batch item. The Lambda function is also expected to return a list-shaped response, matching the order of the items sent in the request, as follows:

```
[
  { "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
]
```

operation is required.

## Payload

The `payload` field is a container that you can use to pass any well-formed JSON to the Lambda function.

If the `operation` field is set to `BatchInvoke`, AWS AppSync wraps the existing `payload` values into a list.

`payload` is optional.

## Response mapping template

As with other data sources, your Lambda function sends a response to AWS AppSync that must be converted to a GraphQL type.

The result of the Lambda function is set on the `context` object that is available via the Velocity Template Language (VTL) `$context.result` property.

If the shape of your Lambda function response exactly matches the shape of the GraphQL type, you can forward the response using the following response mapping template:

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

There are no required fields or shape restrictions that apply to the response mapping template. However, because GraphQL is strongly typed, the resolved mapping template must match the expected GraphQL type.

## Lambda function batched response

If the `operation` field is set to `BatchInvoke`, AWS AppSync expects a list of items back from the Lambda function. In order for AWS AppSync to map each result back to the original request item, the response list must match in size and order. It is OK to have `null` items in the response list; `$ctx.result` is set to `null` accordingly.

## Direct Lambda Resolvers

If you wish to circumvent the use of mapping templates entirely, AWS AppSync can provide a default payload to your Lambda function and a default of a Lambda function's response to a GraphQL type. You can choose to provide a request template, a response template, or neither, and AWS AppSync handles it accordingly.

## Direct Lambda request mapping template

When the request mapping template is not provided, AWS AppSync will send the `Context` object directly to your Lambda function as an `Invoke` operation. For more information about the structure of the `Context` object, see [Resolver mapping template context reference \(p. 279\)](#).

## Direct Lambda response mapping template

When the response mapping template is not provided, AWS AppSync does one of two things upon receiving your Lambda function's response. If you did not provide a request mapping template, or if you provided a request mapping template with the version "2018-05-29", then the response logic functions equivalent to the following response mapping template:

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end
$util.toJson($ctx.result)
```

If you provided a template with the version "2017-02-28", the response logic functions equivalent to the following response mapping template:

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

Superficially, the mapping template bypass operates similarly to using certain mapping templates, as shown in the preceding examples. However, behind the scenes, the evaluation of the mapping templates is circumvented entirely. Because the template evaluation step is bypassed, in some scenarios applications might experience less overhead and latency during the response when compared to a Lambda function with a response mapping template that needs to be evaluated.

## Custom error handling in Direct Lambda Resolver responses

You can customize error responses from Lambda functions that Direct Lambda Resolvers invoke by raising a custom exception. The following example demonstrates how to create a custom exception using JavaScript:

```
class CustomException extends Error {
    constructor(message) {
        super(message);
        this.name = "CustomException";
    }
}

throw new CustomException("Custom message");
```

When exceptions are raised, the `errorType` and `errorMessage` are the name and message, respectively, of the custom error that is thrown.

If `errorType` is `UnauthorizedException`, AWS AppSync returns the default message ("You are not authorized to make this call.") instead of a custom message.

The following is an example GraphQL response that demonstrates a custom `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 Resolvers: Batching enabled

You can enable batching for your Direct Lambda Resolver by configuring the `maxBatchSize` on your resolver. When `maxBatchSize` is set to a value greater than 0 for a Direct Lambda resolver, AWS AppSync sends requests in batches to your Lambda function in sizes up to `maxBatchSize`.

Setting `maxBatchSize` to 0 on a Direct Lambda resolver turns off batching.

For more information on how batching with Lambda resolvers works, see [Advanced use case: Batching \(p. 94\)](#).

### Request mapping template

When batching is enabled and the request mapping template is not provided, AWS AppSync sends a list of Context objects as a `BatchInvoke` operation directly to your Lambda function.

### Response mapping template

When batching is enabled and the response mapping template is not provided, the response logic is equivalent to the following response mapping template:

```
#if( $context.result && $context.result.errorMessage )
  $utils.error($context.result.errorMessage, $context.result.errorType,
    $context.result.data)
#else
  $utils.toJson($context.result.data)
#end
```

The Lambda function must return a list of results in the same order as the list of Context objects that were sent. You can return individual errors by providing an `errorMessage` and `errorType` for a specific result. Each result in the list has the following format:

```
{
  "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

Other fields in the result object are currently ignored.

### Handling errors from Lambda

You can return an error for all results by throwing an exception or an error in your Lambda function. If the payload request or response size for your batch request is too large, Lambda returns an error. In that case, you should consider reducing your `maxBatchSize` or reducing the size of the response payload.

For information on handling individual errors, see [Returning individual errors \(p. 96\)](#).

## Sample Lambda functions

Using the schema below, you can create a Direct Lambda Resolver for the `Post.relatedPosts` field resolver and enable batching by setting `maxBatchSize` to greater than 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]
}
```

In the following query, the Lambda function will be called with batches of requests to resolve `relatedPosts`:

```
query getAllPosts {
  allPosts {
    id
    relatedPosts {
      id
    }
  }
}
```

A simple implementation of a Lambda function is provided below:

```
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\nSAMPLE 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' }
    }
  })
}
```

## Resolver mapping template reference for None data source

The AWS AppSync resolver mapping template used with the data source of type *None*, enables you to shape requests for AWS AppSync local operations.

## Request mapping template

The mapping template is simple and enables you to pass as much context information as possible via the payload field.

```
{
  "version": string,
  "payload": any type
}
```

Here is the JSON schema representation of the request mapping template, once resolved:

```
{
  "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
}
```

Here is an example where the field arguments are passed via the VTL context property `$context.arguments`:

```
{
  "version": "2018-05-29",
  "payload": $util.toJson($context.arguments)
}
```

The value of the payload field will be forwarded to the response mapping template and available on the VTL context property (`$context.result`).

This is an example representing the interpolated value of the payload field:

```
{
  "id": "postId1"
}
```

## Version

Common to all request mapping templates, the `version` field defines the version used by the template.

The `version` field is required.

Example:

```
"version": "2018-05-29"
```

## Payload

The `payload` field is a container that can be used to pass any well-formed JSON to the response mapping template.

The `payload` field is optional.

## Response mapping template

Because there is no data source, the value of the `payload` field will be forwarded to the response mapping template and set on the `context` object that is available via the VTL `$context.result` property.

If the shape of the `payload` field value exactly matches the shape of the GraphQL type, you can forward the response using the following response mapping template:

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

There are no required fields or shape restrictions that apply to the response mapping template. However, because GraphQL is strongly typed, the resolved mapping template must match the expected GraphQL type.

# Resolver Mapping Template Reference for HTTP

The AWS AppSync HTTP resolver mapping templates enable you to send requests from AWS AppSync to any HTTP endpoint, and responses from your HTTP endpoint back to AWS AppSync. By using mapping templates, you can provide hints to AWS AppSync about the nature of the operation to be invoked. This section describes the different mapping templates for the supported HTTP resolver.

## Request Mapping Template

```
{
  "version": "2018-05-29",
  "method": "PUT|POST|GET|DELETE|PATCH",
  "params": {
    "query": Map,
    "headers": Map,
    "body": string
  },
  "resourcePath": string
}
```

After the HTTP request mapping template is resolved, the JSON schema representation of the request mapping template looks like the following:

```
{
  "$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"
  ]
}

```

```
}
```

Following is an example of an HTTP POST request, with a `text/plain` body:

```
{
  "version": "2018-05-29",
  "method": "POST",
  "params": {
    "headers": {
      "Content-Type": "text/plain"
    },
    "body": "this is an example of text body"
  },
  "resourcePath": "/"
}
```

## Version

### Request mapping template only

Defines the version that the template uses. `version` is common to all request mapping templates and is required.

```
"version": "2018-05-29"
```

## Method

### Request mapping template only

HTTP method or verb (GET, POST, PUT, PATCH, or DELETE) that AWS AppSync sends to the HTTP endpoint.

```
"method": "PUT"
```

## ResourcePath

### Request mapping template only

The resource path that you want to access. Along with the endpoint in the HTTP data source, the resource path forms the URL that the AWS AppSync service makes a request to.

```
"resourcePath": "/v1/users"
```

When the mapping template is evaluated, this path is sent as part of the HTTP request, including the HTTP endpoint. For example, the previous example might translate to the following:

```
PUT <endpoint>/v1/users
```

## Params Field

### Request mapping template only

Used to specify what action your search performs, most commonly by setting the **query** value inside the **body**. However, there are several other capabilities that can be configured, such as the formatting of responses.

## headers

The header information, as key-value pairs. Both the key and the value must be strings.

For example:

```
"headers" : {  
  "Content-Type" : "application/json"  
}
```

Currently supported Content-Type headers are:

```
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
```

**Note:** You can't set the following HTTP headers:

```
HOST  
CONNECTION  
USER-AGENT  
EXPECTATION  
TRANSFER_ENCODING  
CONTENT_LENGTH
```

## query

Key-value pairs that specify common options, such as code formatting for JSON responses. Both the key and the value must be a string. The following example shows how you can send a query string as `?type=json`:

```
"query" : {  
  "type" : "json"  
}
```

## body

The body contains the HTTP request body that you choose to set. The request body is always a UTF-8 encoded string unless the content type specifies the charset.

```
"body": "body string"
```

# Certificate Authorities (CA) Recognized by AWS AppSync for HTTPS Endpoints

### Note

Let's Encrypt is accepted via the *identrust* and *isrgrootx1* certificates. No action on your part is required if you use Let's Encrypt.

At this time, self-signed certificates are not supported by HTTP resolvers when using HTTPS. AWS AppSync recognizes the following Certificate Authorities when resolving SSL/TLS certificates for HTTPS:

### Known root certificates in AWS AppSync

Name	Date	SHA1 Fingerprint
digicertassuredidrootca	Apr 21, 2018	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4
trustcenterclass2caii	Apr 21, 2018	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:3
thawtepremiumserverca	Apr 21, 2018	E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:6
cia-crt-g3-02-ca	Nov 23, 2016	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6
swisssignplatinumg2ca	Apr 21, 2018	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:2
swisssignsilverg2ca	Apr 21, 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:3
thawteserverca	Apr 21, 2018	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:4
equifaxsecureebusinessca	Apr 21, 2018	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:1
securetrustca	Apr 21, 2018	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:1
utnuserfirstclientauthca	Apr 21, 2018	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:7
thawtepersonalfreemailca	Apr 21, 2018	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:1
affirmtrustnetworkingca	Apr 21, 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:0
entrustevca	Apr 21, 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:5
utnuserfirsthardwareca	Apr 21, 2018	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:1
certumca	Apr 21, 2018	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:8
addtrustclass1ca	Apr 21, 2018	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:9
entrustrootcag2	Apr 21, 2018	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:7
equifaxsecureca	Apr 21, 2018	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:6
quovadisrootca3	Apr 21, 2018	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:5
quovadisrootca2	Apr 21, 2018	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7
digicertglobalrootg2	Apr 21, 2018	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:8
digicerthighassuranceevrootca	Apr 21, 2018	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:1
secomvalicertclass1ca	Apr 21, 2018	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:1
equifaxsecureglobalebusca	Apr 21, 2018	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:3
geotrustuniversalca	Apr 21, 2018	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:1
deprecateditsecca	Jan 27, 2012	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:7
verisignclass3ca	Apr 21, 2018	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:A
thawteprimaryrootcag3	Apr 21, 2018	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6
thawteprimaryrootcag2	Apr 21, 2018	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:1
deutschetelekomrootca2	Apr 21, 2018	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:3

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
buypassclass3ca	Apr 21, 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:6
utnuserfirstobjectca	Apr 21, 2018	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2
geotrustprimaryca	Apr 21, 2018	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:1
buypassclass2ca	Apr 21, 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:4
baltimorecodesigningca	Apr 21, 2018	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7
verisignclass1ca	Apr 21, 2018	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:0
baltimorecybertrustca	Apr 21, 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:1
starfieldclass2ca	Apr 21, 2018	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:1
camerfirmachamberscommerceca	Apr 21, 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:1
ttelesecglobalrootclass4ca	Apr 21, 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:8
verisignclass3g5ca	Apr 21, 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:2
ttelesecglobalrootclass4ca	Apr 21, 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:9
trustcenteruniversalca1	Apr 21, 2018	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:3
verisignclass3g4ca	Apr 21, 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:0
verisignclass3g3ca	Apr 21, 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5
xrampglobalca	Apr 21, 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:5
amzninternalrootca	Dec 12, 2008	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:1
certplusclass3pprimaryca	Apr 21, 2018	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:1
certumtrustednetworkca	Apr 21, 2018	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:0
verisignclass3g2ca	Apr 21, 2018	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:7
globalsignr3ca	Apr 21, 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:7
utndatacorpsgcca	Apr 21, 2018	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:1
secomscrootca2	Apr 21, 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:0
gtecybertrustglobalca	Apr 21, 2018	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:1
secomscrootca1	Apr 21, 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:1
affirmtrustcommercialca	Apr 21, 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:7
trustcenterclass4caii	Apr 21, 2018	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:3
verisignuniversalrootca	Apr 21, 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0
globalsignr2ca	Apr 21, 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2
certplusclass2primaryca	Apr 21, 2018	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:1

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
digicertglobalrootca	Apr 21, 2018	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:5
globalsignca	Apr 21, 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:8
thawteprimaryrootca	Apr 21, 2018	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7
starfieldrootg2ca	Apr 21, 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0
geotrustglobalca	Apr 21, 2018	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:8
soneraclass2ca	Apr 21, 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9
verisigntsaca	Apr 21, 2018	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:0
soneraclass1ca	Apr 21, 2018	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:3
quovadisrootca	Apr 21, 2018	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:7
affirmtrustpremiumeccca	Apr 21, 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:0
starfieldservicesrootg2ca	Apr 21, 2018	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:0
valicertclass2ca	Apr 21, 2018	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:1
comodoaaca	Apr 21, 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:F
aolrootca2	Apr 21, 2018	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:9
keynectisrootca	Apr 21, 2018	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:0
addtrustqualifiedca	Apr 21, 2018	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8
aolrootca1	Apr 21, 2018	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:5
verisignclass2g3ca	Apr 21, 2018	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:0
addtrustexternalca	Apr 21, 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:3
verisignclass2g2ca	Apr 21, 2018	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:E
geotrustprimarycag3	Apr 21, 2018	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2
geotrustprimarycag2	Apr 21, 2018	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:1
swisssigngoldg2ca	Apr 21, 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:2
entrust2048ca	Apr 21, 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:3
chunghwaepkirootca	Apr 21, 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:0
camerfirmachambersignca	Apr 21, 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:A
camerfirmachambersca	Apr 21, 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:0
godaddyclass2ca	Apr 21, 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:E
affirmtrustpremiumca	Apr 21, 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:2
verisignclass1g3ca	Apr 21, 2018	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:4
secomevrootca1	Apr 21, 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:0

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
verisignclasslg2ca	Apr 21, 2018	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:1E:0A:00
amzninternalinfoseccag3	Feb 27, 2015	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2D:0A:00
cia-crt-g3-01-ca	Nov 23, 2016	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:1E:0A:00
godaddyrootg2ca	Apr 21, 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:1E:0A:00
digicertassuredidrootca	Apr 21, 2018	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:0A:00
microsecszignorootca200	Apr 21, 2018	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:3D:0A:00
affirmtrustcommercial	Apr 21, 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:7D:0A:00
comodoecccertificationa	Apr 21, 2018	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:0D:0A:00
cadisigrootr2	Apr 21, 2018	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:1E:0A:00
swisssignsilvercag2	Apr 21, 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:3D:0A:00
securetrustca	Apr 21, 2018	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:1E:0A:00
cadisigrootr1	Apr 21, 2018	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:5D:0A:00
accvraiz1	Apr 21, 2018	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:6D:0A:00
entrustrootcertificatio	Apr 21, 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:5D:0A:00
camerfirmaglobalchamber	Apr 21, 2018	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:1E:0A:00
dstacescax6	Apr 21, 2018	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:5D:0A:00
identrustpublicsectorro	Apr 21, 2018	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:4D:0A:00
starfieldrootcertificat	Apr 21, 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0D:0A:00
secureglobalca	Apr 21, 2018	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3D:0A:00
eecertificationcentrero	Apr 21, 2018	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:0D:0A:00
opentrustrootcag3	Apr 21, 2018	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8D:0A:00
teliasonerarootcav1	Apr 21, 2018	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:8D:0A:00
autoridaddecertificacio	Apr 21, 2018	AE:05:1F:1F:5F:26:84:B6:8B:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:1E:0A:00
opentrustrootcag2	Apr 21, 2018	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:6D:0A:00
opentrustrootcag1	Apr 21, 2018	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:1E:0A:00
globalsigneccrootcar5	Apr 21, 2018	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:0D:0A:00
globalsigneccrootcar4	Apr 21, 2018	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:0D:0A:00
izenpecom	Apr 21, 2018	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:1E:0A:00
turktrustelektronikser	Apr 21, 2018	54:1A:7E:1D:46:D1:DF:00:3D:27:30:13:72:43:A9:12:11:C6:7D:0A:00
gdcatrustauthr5root	Apr 21, 2018	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:7D:0A:00

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
dtrustrootclass3ca22009	Apr 21, 2018	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:0
quovadisrootca3	Apr 21, 2018	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:5
quovadisrootca2	Apr 21, 2018	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:5
geotrustprimarycertificat	Apr 21, 2018	18:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2
geotrustprimarycertificat	Apr 21, 2018	8D:47:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:1
oistewisekeyglobalrootg	Apr 21, 2018	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:0
addtrustexternalroot	Apr 21, 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:3
chambersofcommerceroo	Apr 21, 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:0
digicertglobalrootg3	Apr 21, 2018	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:3
comodoaaaservicesroot	Apr 21, 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:F
digicertglobalrootg2	Apr 21, 2018	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:8
certinomisrootca	Apr 21, 2018	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:8
oistewisekeyglobalrootg	Apr 21, 2018	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0
dstrootcax3	Apr 21, 2018	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7
certigna	Apr 21, 2018	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:5
digicerthighassuranceev	Apr 21, 2018	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:1
soneraclass2rootca	Apr 21, 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9
trustcorrootcertca2	Apr 21, 2018	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:5
usertrustrsacertificati	Apr 21, 2018	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:2
trustcorrootcertca1	Apr 21, 2018	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:0
geotrustuniversalca	Apr 21, 2018	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:F
certsignrootca	Apr 21, 2018	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2
amazonrootca4	Apr 21, 2018	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:0
amazonrootca3	Apr 21, 2018	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:1
amazonrootca2	Apr 21, 2018	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4
verisignuniversalrootce	Apr 21, 2018	17:03:0A:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0
amazonrootca1	Apr 21, 2018	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:1
networksolutionscertifi	Apr 21, 2018	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:1
thawteprimaryrootcag3	Apr 21, 2018	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:0
affirmtrustnetworking	Apr 21, 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:0
thawteprimaryrootcag2	Apr 21, 2018	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:1



Name	Date	SHA1 Fingerprint
trustcoreca1	Apr 21, 2018	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:5A
deutschetelekomrootca2	Apr 21, 2018	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:8A
godaddyrootcertificateauthority1	Apr 21, 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:5A
entrustrootcertificationaltd	Apr 21, 2018	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A
szafirrootca2	Apr 21, 2018	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:8A
tubitakkamusmssllkokserti	Apr 21, 2018	11:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:5A
buypassclass3rootca	Apr 21, 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:5A
comodorsacertificationaltd	Apr 21, 2018	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:5A
netlockaranyclassgoldfor	Apr 21, 2018	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:5A
securitycommunicationrootca	Apr 21, 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:5A
dtrustrootclass3ca2ev200	Apr 21, 2018	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:5A
starfieldclass2ca	Apr 21, 2018	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:5A
pscpocert	Apr 21, 2018	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:5A
actalisauthenticationrootca	Apr 21, 2018	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:5A
staatdernederlandenrootca	Apr 21, 2018	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:5A
cfcaeavroot	Apr 21, 2018	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:5A
digicerttrustedrootg4	Apr 21, 2018	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:5A
staatdernederlandenrootca	Apr 21, 2018	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:5A
securitycommunicationeurope	Apr 21, 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:5A
globalsignrootcar3	Apr 21, 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5A
globalsignrootcar2	Apr 21, 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:5A
certumtrustednetworkca2	Apr 21, 2018	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:5A
acraizfnmtrcm	Apr 21, 2018	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:5A
hellenicacademicandresearch	Apr 21, 2018	91:05:7C:8D:0A:2D:55:9A:F3:7D:74:97:B4:BC:6F:8A
certplusrootcag2	Apr 21, 2018	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:5A
twcarootcertificationaltd	Apr 21, 2018	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:5A
twcaglobalrootca	Apr 21, 2018	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:5A
certplusrootcag1	Apr 21, 2018	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:5A
geotrustuniversalca2	Apr 21, 2018	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:5A
baltimorecybertrustroot	Apr 21, 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:5A
buypassclass2rootca	Apr 21, 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:5A

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
certumtrustednetworkca	Apr 21, 2018	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:0
digicertassuredidrootg3	Apr 21, 2018	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:3
digicertassuredidrootg2	Apr 21, 2018	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5
isrgrootx1	Apr 21, 2018	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A
entrustnetpremium2048se	Apr 21, 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:3
certplusclass2primaryca	Apr 21, 2018	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E
digicertglobalrootca	Apr 21, 2018	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:5
entrustrootcertificatio	Apr 21, 2018	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:7
starfieldservicesrootce	Apr 21, 2018	92:5A:95:28D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6
thawteprimaryrootca	Apr 21, 2018	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7
atostrustedroot2011	Apr 21, 2018	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:A
geotrustglobalca	Apr 21, 2018	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:8
luxtrustglobalroot2	Apr 21, 2018	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5
etugracertificationauth	Apr 21, 2018	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:5
visaecommerceroot	Apr 21, 2018	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:7
quovadisrootca	Apr 21, 2018	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:7
identrustcommercialroot	Apr 21, 2018	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3
staatdernederlandenevr	Apr 21, 2018	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:1
ttelesecglobalrootclass	Apr 21, 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:8
ttelesecglobalrootclass	Apr 21, 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:9
comodocertificationauth	Apr 21, 2018	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:F
securitycommunicationro	Apr 21, 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:1
quovadisrootca3g3	Apr 21, 2018	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2
xrampglobalcaroot	Apr 21, 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:5
securesignrootca11	Apr 21, 2018	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:3
affirmtrustpremium	Apr 21, 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:2
globalsignrootca	Apr 21, 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:8
swisssigngoldcag2	Apr 21, 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:2
quovadisrootca2g3	Apr 21, 2018	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:0
affirmtrustpremiumecc	Apr 21, 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:0
geotrustprimarycertific	Apr 21, 2018	11:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:1

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
quovadisrootcalg3	Apr 21, 2018	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:57:0A:00
hongkongpostrootca1	Apr 21, 2018	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:80:00:00
usertrustecccertificationauthority	Apr 21, 2018	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:57:0A:00
cybertrustglobalroot	Apr 21, 2018	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:33:00:00
godaddyclass2ca	Apr 21, 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:FE:00:00
hellenicacademicandresearchinstitution	Apr 21, 2018	01:05:06:05:2A:59:8:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:57:0A:00
ecacc	Apr 21, 2018	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:1E:00:00
hellenicacademicandresearchinstitution	Apr 21, 2018	FE:45:65:0B:20:1103:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:20:00:00
verisignclass3publicpriortocertificationauthority	Apr 21, 2018	4E:16:25:76:14:99:B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A0:00:00
verisignclass3publicpriortocertificationauthority	Apr 21, 2018	2A:1D:5A:8D:15:1E:92:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:00:00:00
verisignclass3publicpriortocertificationauthority	Apr 21, 2018	4E:16:25:76:14:99:B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A0:00:00
trustisfpsrootca	Apr 21, 2018	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:00:00:00
epkirootcertificationauthority	Apr 21, 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:00:00:00
globalchambersignroot2008	Apr 21, 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:A0:00:00
camerfirmachambersofcommerce	Apr 21, 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:FE:00:00
mozillacert81.pem	Mar 13, 2014	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:00:00:00
mozillacert99.pem	Mar 13, 2014	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:00:00
mozillacert145.pem	Mar 13, 2014	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:30:00:00
mozillacert37.pem	Mar 13, 2014	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:90:00:00
mozillacert4.pem	Mar 13, 2014	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:0A:00
mozillacert70.pem	Mar 13, 2014	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:00:00:00
mozillacert88.pem	Mar 13, 2014	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:20:00:00
mozillacert134.pem	Mar 13, 2014	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:00:00:00
mozillacert26.pem	Mar 13, 2014	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:1E:00:00
mozillacert77.pem	Mar 13, 2014	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:57:0A:00

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
mozillacert123.pem	Mar 13, 2014	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10:DD:6B:DF:99:72:2C:9
mozillacert15.pem	Mar 13, 2014	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:F
mozillacert66.pem	Mar 13, 2014	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:4
mozillacert112.pem	Mar 13, 2014	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:8
mozillacert55.pem	Mar 13, 2014	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:1
mozillacert101.pem	Mar 13, 2014	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:1
mozillacert119.pem	Mar 13, 2014	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2
mozillacert44.pem	Mar 13, 2014	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:3
mozillacert108.pem	Mar 13, 2014	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:8
mozillacert95.pem	Mar 13, 2014	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:6
mozillacert141.pem	Mar 13, 2014	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:F
mozillacert33.pem	Mar 13, 2014	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:6
mozillacert0.pem	Mar 13, 2014	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F
mozillacert84.pem	Mar 13, 2014	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9
mozillacert130.pem	Mar 13, 2014	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:F
mozillacert148.pem	Mar 13, 2014	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F
mozillacert22.pem	Mar 13, 2014	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F
mozillacert7.pem	Mar 13, 2014	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:F
mozillacert73.pem	Mar 13, 2014	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0
mozillacert137.pem	Mar 13, 2014	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
mozillacert11.pem	Mar 13, 2014	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4
mozillacert29.pem	Mar 13, 2014	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:1
mozillacert62.pem	Mar 13, 2014	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:A
mozillacert126.pem	Mar 13, 2014	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:9
mozillacert18.pem	Mar 13, 2014	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4
mozillacert51.pem	Mar 13, 2014	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2
mozillacert69.pem	Mar 13, 2014	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:1
mozillacert115.pem	Mar 13, 2014	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:9
mozillacert40.pem	Mar 13, 2014	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:1
mozillacert58.pem	Mar 13, 2014	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:1
mozillacert104.pem	Mar 13, 2014	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5
mozillacert91.pem	Mar 13, 2014	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:0
mozillacert47.pem	Mar 13, 2014	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:3
mozillacert80.pem	Mar 13, 2014	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:0
mozillacert98.pem	Mar 13, 2014	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:0
mozillacert144.pem	Mar 13, 2014	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9
mozillacert36.pem	Mar 13, 2014	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:3
mozillacert3.pem	Mar 13, 2014	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:7
mozillacert87.pem	Mar 13, 2014	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:0
mozillacert133.pem	Mar 13, 2014	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:9

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
mozillacert25.pem	Mar 13, 2014	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A
mozillacert76.pem	Mar 13, 2014	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:7
mozillacert122.pem	Mar 13, 2014	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:3
mozillacert14.pem	Mar 13, 2014	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:D
mozillacert65.pem	Mar 13, 2014	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:3
mozillacert111.pem	Mar 13, 2014	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:A
mozillacert129.pem	Mar 13, 2014	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:E
mozillacert54.pem	Mar 13, 2014	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2
mozillacert100.pem	Mar 13, 2014	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:0
mozillacert118.pem	Mar 13, 2014	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:1
mozillacert151.pem	Mar 13, 2014	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:7
mozillacert43.pem	Mar 13, 2014	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:1
mozillacert107.pem	Mar 13, 2014	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:5
mozillacert94.pem	Mar 13, 2014	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:4
mozillacert140.pem	Mar 13, 2014	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7
mozillacert32.pem	Mar 13, 2014	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:3
mozillacert83.pem	Mar 13, 2014	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:E
mozillacert147.pem	Mar 13, 2014	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:1
mozillacert21.pem	Mar 13, 2014	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:3
mozillacert39.pem	Mar 13, 2014	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:3

Name	Date	SHA1 Fingerprint
mozillacert6.pem	Mar 13, 2014	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:3
mozillacert72.pem	Mar 13, 2014	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:3
mozillacert136.pem	Mar 13, 2014	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:3
mozillacert10.pem	Mar 13, 2014	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:3
mozillacert28.pem	Mar 13, 2014	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:3
mozillacert61.pem	Mar 13, 2014	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:3
mozillacert79.pem	Mar 13, 2014	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:3
mozillacert125.pem	Mar 13, 2014	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:3
mozillacert17.pem	Mar 13, 2014	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:3
mozillacert50.pem	Mar 13, 2014	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:3
mozillacert68.pem	Mar 13, 2014	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:3
mozillacert114.pem	Mar 13, 2014	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:3
mozillacert57.pem	Mar 13, 2014	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:3
mozillacert103.pem	Mar 13, 2014	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:3
mozillacert90.pem	Mar 13, 2014	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:3
mozillacert46.pem	Mar 13, 2014	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:3
mozillacert97.pem	Mar 13, 2014	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:3
mozillacert143.pem	Mar 13, 2014	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:3
mozillacert35.pem	Mar 13, 2014	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:3
mozillacert2.pem	Mar 13, 2014	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:3

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
mozillacert86.pem	Mar 13, 2014	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:7
mozillacert132.pem	Mar 13, 2014	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:5
mozillacert24.pem	Mar 13, 2014	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:F
mozillacert9.pem	Mar 13, 2014	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:4
mozillacert75.pem	Mar 13, 2014	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:6
mozillacert121.pem	Mar 13, 2014	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:9
mozillacert139.pem	Mar 13, 2014	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:7
mozillacert13.pem	Mar 13, 2014	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:0
mozillacert64.pem	Mar 13, 2014	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:F
mozillacert110.pem	Mar 13, 2014	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:6
mozillacert128.pem	Mar 13, 2014	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:9
mozillacert53.pem	Mar 13, 2014	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:F
mozillacert117.pem	Mar 13, 2014	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:F
mozillacert150.pem	Mar 13, 2014	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:1
mozillacert42.pem	Mar 13, 2014	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:3
mozillacert106.pem	Mar 13, 2014	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1
mozillacert93.pem	Mar 13, 2014	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:3
mozillacert31.pem	Mar 13, 2014	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:0
mozillacert49.pem	Mar 13, 2014	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:2
mozillacert82.pem	Mar 13, 2014	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:8



AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
mozillacert146.pem	Mar 13, 2014	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0
mozillacert20.pem	Mar 13, 2014	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:2
mozillacert38.pem	Mar 13, 2014	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:1
mozillacert5.pem	Mar 13, 2014	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:5
mozillacert71.pem	Mar 13, 2014	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:A
mozillacert89.pem	Mar 13, 2014	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:7
mozillacert135.pem	Mar 13, 2014	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:8
mozillacert27.pem	Mar 13, 2014	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3
mozillacert60.pem	Mar 13, 2014	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:1
mozillacert78.pem	Mar 13, 2014	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:0
mozillacert124.pem	Mar 13, 2014	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8
mozillacert16.pem	Mar 13, 2014	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7
mozillacert67.pem	Mar 13, 2014	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:7
mozillacert113.pem	Mar 13, 2014	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:3
mozillacert56.pem	Mar 13, 2014	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6
mozillacert102.pem	Mar 13, 2014	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:1
mozillacert45.pem	Mar 13, 2014	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:0
mozillacert109.pem	Mar 13, 2014	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:1
mozillacert96.pem	Mar 13, 2014	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:8
mozillacert142.pem	Mar 13, 2014	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:5

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
mozillacert34.pem	Mar 13, 2014	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0
mozillacert1.pem	Mar 13, 2014	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8
mozillacert85.pem	Mar 13, 2014	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:2
mozillacert131.pem	Mar 13, 2014	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:2
mozillacert149.pem	Mar 13, 2014	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:1
mozillacert23.pem	Mar 13, 2014	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7
mozillacert8.pem	Mar 13, 2014	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:0
mozillacert74.pem	Mar 13, 2014	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:0
mozillacert120.pem	Mar 13, 2014	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8
mozillacert138.pem	Mar 13, 2014	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:0
mozillacert12.pem	Mar 13, 2014	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:5
mozillacert63.pem	Mar 13, 2014	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:3
mozillacert127.pem	Mar 13, 2014	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:8
mozillacert19.pem	Mar 13, 2014	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:8
mozillacert52.pem	Mar 13, 2014	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4
mozillacert116.pem	Mar 13, 2014	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:A
mozillacert41.pem	Mar 13, 2014	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:3
mozillacert59.pem	Mar 13, 2014	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0
mozillacert105.pem	Mar 13, 2014	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:4
mozillacert92.pem	Mar 13, 2014	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:1

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
mozillacert30.pem	Mar 13, 2014	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E
mozillacert48.pem	Mar 13, 2014	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:I
verisignc4g2.pem	Mar 20, 2014	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9
verisignc2g3.pem	Mar 20, 2014	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:0
verisignc1g6.pem	Dec 31, 2014	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:0
verisignc2g2.pem	Mar 20, 2014	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:1
verisignroot.pem	Mar 20, 2014	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0
verisignc2g1.pem	Mar 20, 2014	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:1
verisignc3g5.pem	Mar 20, 2014	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A
verisignc1g3.pem	Mar 20, 2014	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:4
verisignc3g4.pem	Mar 20, 2014	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:0
verisignc1g2.pem	Mar 20, 2014	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:1
verisignc3g3.pem	Mar 20, 2014	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5
verisignc1g1.pem	Mar 20, 2014	90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:5
verisignc3g2.pem	Mar 20, 2014	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:7
verisignc3g1.pem	Mar 20, 2014	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:A
verisignc2g6.pem	Dec 31, 2014	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1
verisignc4g3.pem	Mar 20, 2014	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:7
gdroot-g2.pem	Dec 31, 2014	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:1
gd-class2-root.pem	Dec 31, 2014	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:1

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
gd_bundle-g2.pem	Dec 31, 2014	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:3
dstacescax6	Jun 18, 2018	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:9
gd_bundle-g2.pem	Jun 18, 2018	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:3
verisignc4g3.pem	Jun 18, 2018	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:7
swisssignplatinumg2ca	Apr 21, 2018	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:A
geotrustprimarycertificat	Jun 18, 2018	18:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2
geotrustprimarycertificat	Jun 18, 2018	18:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2
buypassclass2rootca	Jun 18, 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:4
camerfirmachambersofcom	Jun 18, 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:1
mozillacert20.pem	Jun 18, 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:2
mozillacert12.pem	Jun 18, 2018	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:5
mozillacert90.pem	Jun 18, 2018	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9
mozillacert82.pem	Jun 18, 2018	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:8
mozillacert140.pem	Jun 18, 2018	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7
mozillacert74.pem	Jun 18, 2018	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6
mozillacert132.pem	Jun 18, 2018	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:5
mozillacert66.pem	Jun 18, 2018	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:4
mozillacert124.pem	Jun 18, 2018	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8
mozillacert58.pem	Jun 18, 2018	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:1
securitycommunicationroo	Jun 18, 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:0
mozillacert116.pem	Jun 18, 2018	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:A
mozillacert108.pem	Jun 18, 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:8
certigna	Jun 18, 2018	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:5
mozillacert3.pem	Jun 18, 2018	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:7
verisignc1g1.pem	Jun 18, 2018	90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:5
verisignc4g2.pem	Jun 18, 2018	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9
deutschetelekomrootca2	Jun 18, 2018	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:3
starfieldrootg2ca	Apr 21, 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0
comodoecccertificationa	Jun 18, 2018	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:0
digicertglobalrootg3	Jun 18, 2018	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:3

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
digicertglobalrootg2	Jun 18, 2018	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:8
mozillacert11.pem	Jun 18, 2018	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4
mozillacert81.pem	Jun 18, 2018	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:0
mozillacert73.pem	Jun 18, 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0
szafirrootca2	Jun 18, 2018	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3
mozillacert131.pem	Jun 18, 2018	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:2
ecacc	Jun 18, 2018	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:1
mozillacert65.pem	Jun 18, 2018	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:3
turktrustelektronikserturktr	Jun 18, 2018	64:1A:12:2E:5A:1F:4D:46:D1:DF:00:3D:27:30:13:72:43:A9:12:11:C6:7
mozillacert123.pem	Jun 18, 2018	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10:DD:6B:DF:99:72:2C:9
mozillacert57.pem	Jun 18, 2018	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8
mozillacert115.pem	Jun 18, 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:9
mozillacert49.pem	Jun 18, 2018	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:2
mozillacert107.pem	Jun 18, 2018	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:5
verisignclass3g4ca	Apr 21, 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6
securetrustca	Jun 18, 2018	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:1
mozillacert2.pem	Jun 18, 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6
buypassclass2ca	Apr 21, 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:4
secomscrootca2	Apr 21, 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:0
secomscrootca1	Apr 21, 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:1
trustisfpsrootca	Jun 18, 2018	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:0
hongkongpostrootca1	Jun 18, 2018	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8
certsignrootca	Jun 18, 2018	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2
geotrustprimaryca	Apr 21, 2018	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:1
twcaglobalrootca	Jun 18, 2018	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:A
camerfirmachambersca	Apr 21, 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:0
mozillacert10.pem	Jun 18, 2018	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7
mozillacert80.pem	Jun 18, 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C
mozillacert72.pem	Jun 18, 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:F
comodoaaca	Apr 21, 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:1
mozillacert130.pem	Jun 18, 2018	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:1

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
mozillacert64.pem	Jun 18, 2018	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:1
mozillacert122.pem	Jun 18, 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:3
mozillacert56.pem	Jun 18, 2018	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6
equifaxsecurebusinessca	Apr 21, 2018	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:1
camerfirmachambersignca	Apr 21, 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:A
mozillacert114.pem	Jun 18, 2018	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:5
mozillacert48.pem	Jun 18, 2018	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:1
pscprocert	Jun 18, 2018	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:5
mozillacert106.pem	Jun 18, 2018	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:3
mozillacert1.pem	Jun 18, 2018	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8
eecertificationcentrercn	Jun 18, 2018	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:0
digicertglobalrootca	Jun 18, 2018	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:5
thawteprimaryrootcag3	Jun 18, 2018	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6
thawteprimaryrootcag2	Jun 18, 2018	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:1
entrustrootcertificatio	Jun 18, 2018	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:F
valicertclass2ca	Apr 21, 2018	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:1
globalchambersignroot20	Jun 18, 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:A
amazonrootca4	Jun 18, 2018	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:6
gd-class2-root.pem	Jun 18, 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:1
amazonrootca3	Jun 18, 2018	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:1
amazonrootca2	Jun 18, 2018	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4
securitycommunicationro	Jun 18, 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:1
amazonrootca1	Jun 18, 2018	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:1
acraizfnmtrcm	Jun 18, 2018	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2
quovadisrootca3g3	Jun 18, 2018	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2
certplusrootcag2	Jun 18, 2018	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:0
certplusrootcag1	Jun 18, 2018	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:1
mozillacert71.pem	Jun 18, 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:A
mozillacert63.pem	Jun 18, 2018	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:3
mozillacert121.pem	Jun 18, 2018	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:5
ttelesecglobalrootclass	Apr 21, 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:8

Name	Date	SHA1 Fingerprint
mozillacert55.pem	Jun 18, 2018	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:
mozillacert113.pem	Jun 18, 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:
baltimorecybertrustca	Apr 21, 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:
mozillacert47.pem	Jun 18, 2018	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:
mozillacert105.pem	Jun 18, 2018	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:
mozillacert39.pem	Jun 18, 2018	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:
usertrustecccertificationauthority	Jun 18, 2018	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:
mozillacert0.pem	Jun 18, 2018	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:
securitycommunicationevrootca	Jun 18, 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:
verisignc3g5.pem	Jun 18, 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:
globalsignr3ca	Apr 21, 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:
trustcorecal	Jun 18, 2018	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:
equifaxsecureglobalebusapreca	Apr 21, 2018	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:
geotrustuniversalca	Jun 18, 2018	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:
affirmtrustpremiumca	Apr 21, 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:
staatdernederlandenrootca1	Jun 18, 2018	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:
staatdernederlandenrootca2	Jun 18, 2018	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:
mozillacert70.pem	Jun 18, 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:
secomevrootca1	Apr 21, 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:
geotrustglobalca	Jun 18, 2018	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:
mozillacert62.pem	Jun 18, 2018	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:
mozillacert120.pem	Jun 18, 2018	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:
mozillacert54.pem	Jun 18, 2018	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:
mozillacert112.pem	Jun 18, 2018	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:
mozillacert46.pem	Jun 18, 2018	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:
swissisigngoldcag2	Jun 18, 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:
mozillacert104.pem	Jun 18, 2018	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:
mozillacert38.pem	Jun 18, 2018	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:
certplusclass3pprimarycap	Apr 21, 2018	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:
entrustrootcertificationaltprimeca	Jun 18, 2018	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:
godaddyrootg2ca	Apr 21, 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:



AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
cfcaeavroot	Jun 18, 2018	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:8
verisignc3g4.pem	Jun 18, 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:C
geotrustuniversalca2	Jun 18, 2018	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:2
starfieldservicesrootg2Apr 21, 2018	Apr 21, 2018	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6
digicerthighassuranceevJun 18, 2018	Jun 18, 2018	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:D
entrustnetpremium2048seJun 18, 2018	Jun 18, 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:3
camerfirmaglobalchamberJun 18, 2018	Jun 18, 2018	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D
verisignclass3g3ca	Apr 21, 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5
godaddyclass2ca	Jun 18, 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:F
mozillacert61.pem	Jun 18, 2018	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:4
mozillacert53.pem	Jun 18, 2018	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:D
atostrustedroot2011	Jun 18, 2018	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:A
mozillacert111.pem	Jun 18, 2018	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:A
staatdernederlandenevroJun 18, 2018	Jun 18, 2018	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:D
mozillacert45.pem	Jun 18, 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C
mozillacert103.pem	Jun 18, 2018	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:5
mozillacert37.pem	Jun 18, 2018	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:9
mozillacert29.pem	Jun 18, 2018	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:D
izenpecom	Jun 18, 2018	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:D
comodorsacertificationaJun 18, 2018	Jun 18, 2018	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8
mozillacert99.pem	Jun 18, 2018	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:C
mozillacert149.pem	Jun 18, 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:D
utnuserfirstobjectca	Apr 21, 2018	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2
verisignc3g3.pem	Jun 18, 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5
dstrootcax3	Jun 18, 2018	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7
addtrustexternalroot	Jun 18, 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:3
certumtrustednetworkca	Jun 18, 2018	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:C
affirmtrustpremiumecc	Jun 18, 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C
starfieldclass2ca	Jun 18, 2018	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:F
actalisauthenticationroJun 18, 2018	Jun 18, 2018	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9
verisignclass2g3ca	Apr 21, 2018	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:6



AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
isrgrootx1	Jun 18, 2018	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A
godaddyrootcertificateauthority1	Jun 18, 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E
mozillacert60.pem	Jun 18, 2018	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:3
chunghwaepkirootca	Apr 21, 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:0
mozillacert52.pem	Jun 18, 2018	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4
microsecszignorootca2009	Jun 18, 2018	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:3
securesignrootca11	Jun 18, 2018	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:3
mozillacert110.pem	Jun 18, 2018	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:6
mozillacert44.pem	Jun 18, 2018	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:3
mozillacert102.pem	Jun 18, 2018	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:3
mozillacert36.pem	Jun 18, 2018	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:3
mozillacert28.pem	Jun 18, 2018	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:E
baltimorecybertrustroot	Jun 18, 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:F
amzninternalrootca	Dec 12, 2008	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:F
mozillacert98.pem	Jun 18, 2018	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:0
mozillacert148.pem	Jun 18, 2018	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F
verisignc3g2.pem	Jun 18, 2018	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:7
quovadisrootca2g3	Jun 18, 2018	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:0
geotrustprimarycertificateauthority	Jun 18, 2018	12:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:1
opentrustrootcag3	Jun 18, 2018	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8
opentrustrootcag2	Jun 18, 2018	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:6
opentrustrootcag1	Jun 18, 2018	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:1
verisignclass3ca	Apr 21, 2018	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:7
globalsignca	Apr 21, 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:8
ttelesecglobalrootclass3ca	Apr 21, 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:9
verisignclass1g3ca	Apr 21, 2018	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:4
verisignuniversalrootca	Apr 21, 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0
soneraclass2ca	Apr 21, 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9
starfieldservicesrootcertificateauthority	Jun 18, 2018	12:5A:95:28D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6
mozillacert51.pem	Jun 18, 2018	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
mozillacert43.pem	Jun 18, 2018	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:1
mozillacert101.pem	Jun 18, 2018	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:3
mozillacert35.pem	Jun 18, 2018	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7
globalsignr2ca	Apr 21, 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2
mozillacert27.pem	Jun 18, 2018	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3
affirmtrustpremium	Jun 18, 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:2
mozillacert19.pem	Jun 18, 2018	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:8
mozillacert97.pem	Jun 18, 2018	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:7
netlockaranyclassgoldforunitsizeany	Jun 18, 2018	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:0
mozillacert89.pem	Jun 18, 2018	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:7
verisignroot.pem	Jun 18, 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0
mozillacert147.pem	Jun 18, 2018	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:1
aolrootca2	Apr 21, 2018	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:9
cia-crt-g3-01-ca	Nov 23, 2016	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:1
aolrootca1	Apr 21, 2018	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:5
verisignc3g1.pem	Jun 18, 2018	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:2
mozillacert139.pem	Jun 18, 2018	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:7
soneraclass2rootca	Jun 18, 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9
swisssignsilverg2ca	Apr 21, 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:3
thawteprimaryrootca	Jun 18, 2018	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7
gdcatrustauthr5root	Jun 18, 2018	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:7
trustcenterclass4caii	Apr 21, 2018	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:3
usertrustsacertificationalbum12018	Jun 18, 2018	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:A
digicertassuredidrootg3	Jun 18, 2018	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:3
digicertassuredidrootg2	Jun 18, 2018	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5
mozillacert50.pem	Jun 18, 2018	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:A
mozillacert42.pem	Jun 18, 2018	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:3
mozillacert100.pem	Jun 18, 2018	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:0
mozillacert34.pem	Jun 18, 2018	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0
affirmtrustcommercialca	Apr 21, 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:7

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
mozillacert26.pem	Jun 18, 2018	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:1B:0E:0A
globalsigneccrootcar5	Jun 18, 2018	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:0E:0A
globalsigneccrootcar4	Jun 18, 2018	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:0E:0A
buypassclass3rootca	Jun 18, 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:0E:0A
mozillacert18.pem	Jun 18, 2018	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4E:0A
mozillacert96.pem	Jun 18, 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:8E:0A
verisignc2g6.pem	Jun 18, 2018	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:3E:0A
secomvalicertclass1ca	Apr 21, 2018	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:1E:0A
mozillacert88.pem	Jun 18, 2018	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:2E:0A
accvraiz1	Jun 18, 2018	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:0E:0A
mozillacert146.pem	Jun 18, 2018	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0E:0A
mozillacert138.pem	Jun 18, 2018	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:0E:0A
verisignclass3g2ca	Apr 21, 2018	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:7E:0A
dtrustrootclass3ca2ev200	Jun 18, 2018	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:3E:0A
xrampglobalca	Apr 21, 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:5E:0A
mozillacert9.pem	Jun 18, 2018	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:4E:0A
verisignuniversalrootcert	Jun 18, 2018	35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0E:0A
tubitakkamusmsslkokserturkiasisurum	Jun 18, 2018	01:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:FE:0A
mozillacert41.pem	Jun 18, 2018	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:3E:0A
mozillacert33.pem	Jun 18, 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:6E:0A
mozillacert25.pem	Jun 18, 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A8:0A
mozillacert17.pem	Jun 18, 2018	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:9E:0A
mozillacert95.pem	Jun 18, 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:0E:0A
affirmtrustpremiumeccca	Apr 21, 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:0E:0A
mozillacert87.pem	Jun 18, 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:0E:0A
mozillacert145.pem	Jun 18, 2018	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:5E:0A
mozillacert79.pem	Jun 18, 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:2E:0A
mozillacert137.pem	Jun 18, 2018	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A8:0A
digicertassuredidrootca	Jun 18, 2018	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4E:0A
addtrustqualifiedca	Apr 21, 2018	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8E:0A
mozillacert129.pem	Jun 18, 2018	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:FE:0A

Name	Date	SHA1 Fingerprint
verisignclass2g2ca	Apr 21, 2018	B3:E A:C4:47:76:C9:C8:1C:E A:F2:9D:95:B6:CC:A0:
baltimorecodesigningca	Apr 21, 2018	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:
luxtrustglobalroot2	Jun 18, 2018	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:
visaecommercerooot	Jun 18, 2018	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:
oistewisekeyglobalrootgma	Jun 18, 2018	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:
mozillacert8.pem	Jun 18, 2018	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:
comodocertificationauthnunitr	Jun 18, 2018	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:
cia-cert-g3-02-ca	Nov 23, 2016	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:
verisignc1g6.pem	Jun 18, 2018	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:
trustcenterclass2cai	Apr 21, 2018	A E:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:
quovadisrootcalg3	Jun 18, 2018	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:
mozillacert40.pem	Jun 18, 2018	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:
cadisigrootr2	Jun 18, 2018	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:
cadisigrootr1	Jun 18, 2018	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:
mozillacert32.pem	Jun 18, 2018	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:
utndatacorpsgcc	Apr 21, 2018	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:
mozillacert24.pem	Jun 18, 2018	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:
addtrustclass1ca	Apr 21, 2018	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:
mozillacert16.pem	Jun 18, 2018	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:
affirmtrustnetworkingca	Apr 21, 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:
mozillacert94.pem	Jun 18, 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:
mozillacert86.pem	Jun 18, 2018	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:
mozillacert144.pem	Jun 18, 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:
mozillacert78.pem	Jun 18, 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:
mozillacert136.pem	Jun 18, 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:
mozillacert128.pem	Jun 18, 2018	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:OD:
verisignclasslg2ca	Apr 21, 2018	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:
hellenicacademicandresearc	Jun 18, 2018	E1:45:65:0A:20:91:03:5B:98:A1:61:B5:51:2E:AC:DA:
soneraclclasslca	Apr 21, 2018	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:
hellenicacademicandresearc	Jun 18, 2018	E1:45:65:0A:20:91:03:5B:98:A1:61:B5:51:2E:AC:DA:

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
certumtrustednetworkca	Jun 18, 2018	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:3
equifaxsecureca	Apr 21, 2018	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:6
thawteserverca	Apr 21, 2018	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:4
mozillacert7.pem	Jun 18, 2018	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:1
affirmtrustnetworking	Jun 18, 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:0
deprecateditsecca	Jan 27, 2012	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:7
globalsignrootcar3	Jun 18, 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:7
globalsignrootcar2	Jun 18, 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2
quovadisrootca	Jun 18, 2018	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:7
mozillacert31.pem	Jun 18, 2018	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:0
entrustrootcertificatio	Jun 18, 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:5
mozillacert23.pem	Jun 18, 2018	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7
mozillacert15.pem	Jun 18, 2018	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:F
verisignc2g3.pem	Jun 18, 2018	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:6
mozillacert93.pem	Jun 18, 2018	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:3
mozillacert151.pem	Jun 18, 2018	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:7
mozillacert85.pem	Jun 18, 2018	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:2
certplusclass2primaryca	Jun 18, 2018	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:F
mozillacert143.pem	Jun 18, 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:1
mozillacert77.pem	Jun 18, 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5
mozillacert135.pem	Jun 18, 2018	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:8
mozillacert69.pem	Jun 18, 2018	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:F
mozillacert127.pem	Jun 18, 2018	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:8
mozillacert119.pem	Jun 18, 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2
geotrustprimarycag3	Apr 21, 2018	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2
identrustpublicsectorro	Jun 18, 2018	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:4
geotrustprimarycag2	Apr 21, 2018	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:1
trustcorrootcertca2	Jun 18, 2018	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:3
mozillacert6.pem	Jun 18, 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:F
trustcorrootcertca1	Jun 18, 2018	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:0
networksolutionscertifi	Jun 18, 2018	75:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:1

AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
twcarootcertificationauthority	Jun 18, 2018	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:2
addtrustexternalca	Apr 21, 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:3
verisignclass3g5ca	Apr 21, 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A
autoridaddecertificaci	Jun 18, 2018	AE:05:1F:3F:52:68:4B:48:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:1
hellenicacademicandre	Jun 18, 2018	9F:05:07:6D:0A:2D:55:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:1
verisigntsaca	Apr 21, 2018	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:0
utnuserfirsthardwareca	Apr 21, 2018	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:1
identrustcommercialro	Jun 18, 2018	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3
dtrustrootclass3ca2200	Jun 18, 2018	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:0
epkirootcertificationau	Jun 18, 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:0
mozillacert30.pem	Jun 18, 2018	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:1
teliasonerarootcav1	Jun 18, 2018	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:8
buypassclass3ca	Apr 21, 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:6
mozillacert22.pem	Jun 18, 2018	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:1
mozillacert14.pem	Jun 18, 2018	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:1
verisignc2g2.pem	Jun 18, 2018	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:1
certumca	Apr 21, 2018	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:8
mozillacert92.pem	Jun 18, 2018	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:1
mozillacert150.pem	Jun 18, 2018	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:1
mozillacert84.pem	Jun 18, 2018	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9
ttelesecglobalrootclass	Jun 18, 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:8
globalsignrootca	Jun 18, 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:8
ttelesecglobalrootclass	Jun 18, 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:9
mozillacert142.pem	Jun 18, 2018	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:5
mozillacert76.pem	Jun 18, 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:7
mozillacert134.pem	Jun 18, 2018	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:0
mozillacert68.pem	Jun 18, 2018	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:1
etugracertificationauth	Jun 18, 2018	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:5
mozillacert126.pem	Jun 18, 2018	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:9
keynectisrootca	Apr 21, 2018	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:0
mozillacert118.pem	Jun 18, 2018	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:3



AWS AppSync Developer Guide  
Certificate Authorities (CA) Recognized  
by AWS AppSync for HTTPS Endpoints

Name	Date	SHA1 Fingerprint
quovadisrootca3	Jun 18, 2018	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:5
quovadisrootca2	Jun 18, 2018	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7
mozillacert5.pem	Jun 18, 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:5
verisignc1g3.pem	Jun 18, 2018	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:4
cybertrustglobalroot	Jun 18, 2018	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:3
amzninternalinfosecg3	Feb 27, 2015	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2
starfieldrootcertificat	Jan 18, 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0
entrust2048ca	Apr 21, 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:3
swisssignsilvercag2	Jun 18, 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:3
affirmtrustcommercial	Jun 18, 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:7
certinomisrootca	Jun 18, 2018	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:8
xrampglobalcaroot	Jun 18, 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:5
secureglobalca	Jun 18, 2018	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3
swisssigngoldg2ca	Apr 21, 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:2
mozillacert21.pem	Jun 18, 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:3
mozillacert13.pem	Jun 18, 2018	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:0
verisignc2g1.pem	Jun 18, 2018	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:3
mozillacert91.pem	Jun 18, 2018	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:0
oistewisekeyglobalrootg	Jun 18, 2018	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0
mozillacert83.pem	Jun 18, 2018	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:E
entrustevca	Apr 21, 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:5
mozillacert141.pem	Jun 18, 2018	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:E
mozillacert75.pem	Jun 18, 2018	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:6
mozillacert133.pem	Jun 18, 2018	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:9
mozillacert67.pem	Jun 18, 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:7
mozillacert125.pem	Jun 18, 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:5
mozillacert59.pem	Jun 18, 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0
thawtepremiumserverca	Apr 21, 2018	E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:6
mozillacert117.pem	Jun 18, 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E
utnuserfirstclientauth	Apr 21, 2018	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:7
entrustrootcag2	Apr 21, 2018	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:7

Name	Date	SHA1 Fingerprint
mozillacert109.pem	Jun 18, 2018	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:1
digicerttrustedrootg4	Jun 18, 2018	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:C
gdroot-g2.pem	Jun 18, 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:1
comodoaaaservicesroot	Jun 18, 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:1
mozillacert4.pem	Jun 18, 2018	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:5
verisignclass3publicpri	Jun 18, 2018	4E:8B:56:7E:1A:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A
chambersofcommerceroot	Jun 18, 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:C
verisignclass3publicpri	Jun 18, 2018	2A:1D:5A:5D:1E:9B:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6
verisignclass3publicpri	Jun 18, 2018	2A:1D:5A:5D:1E:9B:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6
thawtepersonalfreemail	Apr 21, 2018	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:1
verisignclg2.pem	Jun 18, 2018	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:1
gtecybertrustglobalca	Apr 21, 2018	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:1
trustcenteruniversalca	Apr 21, 2018	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:3
camerfirmachamberscom	Apr 21, 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:1
verisignclass1ca	Apr 21, 2018	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:C

## Resolver mapping template changelog

Resolver and function mapping templates are versioned. The mapping template version, such as 2018-05-29 dictates the following:

- \* The expected shape of the data source request configuration provided by the request template
- \* The execution behavior of the request mapping template and the response mapping template

Versions are represented using the YYYY-MM-DD format, a later date corresponds to a more recent version. This page lists the differences between the mapping template versions currently supported in AWS AppSync.

### Topics

- [Datasource Operation Availability Per Version Matrix \(p. 416\)](#)
- [Changing the Version on a Unit Resolver Mapping Template \(p. 417\)](#)
- [Changing the Version on a Function \(p. 418\)](#)
- [2018-05-29 \(p. 418\)](#)
- [2017-02-28 \(p. 423\)](#)

## Datasource Operation Availability Per Version Matrix

Operation/Version Supported	2017-02-28	2018-05-29
AWS Lambda Invoke	Yes	Yes



Operation/Version Supported	2017-02-28	2018-05-29
AWS Lambda BatchInvoke	Yes	Yes
None Datasource	Yes	Yes
Amazon OpenSearch GET	Yes	Yes
Amazon OpenSearch POST	Yes	Yes
Amazon OpenSearch PUT	Yes	Yes
Amazon OpenSearch DELETE	Yes	Yes
Amazon OpenSearch GET	Yes	Yes
DynamoDB GetItem	Yes	Yes
DynamoDB Scan	Yes	Yes
DynamoDB Query	Yes	Yes
DynamoDB DeleteItem	Yes	Yes
DynamoDB PutItem	Yes	Yes
DynamoDB BatchGetItem	No	Yes
DynamoDB BatchPutItem	No	Yes
DynamoDB BatchDeleteItem	No	Yes
HTTP	No	Yes
Amazon RDS	No	Yes

**Note:** Only **2018-05-29** version is currently supported in functions.

## Changing the Version on a Unit Resolver Mapping Template

For Unit resolvers, the version is specified as part of the body of the request mapping template. To update the version, simply update the `version` field to the new version.

For example, to update the version on the AWS Lambda template:

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $utils.toJson($context.arguments)
  }
}
```

You need to update the version field from 2017-02-28 to 2018-05-29 as follows:

```
{
  "version": "2018-05-29", ## Note the version
```

```
"operation": "Invoke",
"payload": {
  "field": "getPost",
  "arguments": $utils.toJson($context.arguments)
}
```

## Changing the Version on a Function

For functions, the version is specified as the `functionVersion` field on the function object. To update the version, simply update the `functionVersion`. *Note:* Currently, only 2018-05-29 is supported for function.

The following is an example of a CLI command to update an existing function version:

```
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)"
```

**Note:** It is recommended to omit the version field from the function request mapping template as it will not be honored. If you do specify a version inside a function request mapping template, the version value will be overridden by the value of the `functionVersion` field.

## 2018-05-29

### Behavior Change

- If the datasource invocation result is `null`, the response mapping template is executed.
- If the datasource invocation yields an error, it is now up to you to handle the error, the response mapping template evaluated result will **always** be placed inside the GraphQL response data block.

### Reasoning

- A `null` invocation result has meaning, and in some application use cases we might want to handle `null` results in a custom way. For example, an application might check if a record exists in an Amazon DynamoDB table to perform some authorization check. In this case, a `null` invocation result would mean the user might not be authorized. Executing the response mapping template now provides the ability to raise an unauthorized error. This behavior provides greater control to the API designer.

Given the following response mapping template:

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

Previously with 2017-02-28, if `$ctx.result` came back `null`, the response mapping template was not executed. With 2018-05-29, we can now handle this scenario. For example, we can choose to raise an authorization error as follows:

```
# throw an unauthorized error if the result is null
#if ( $util.isNull($ctx.result) )
  $util.unauthorized()
```

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

**Note:** Errors coming back from a data source are sometimes not fatal or even expected, that is why the response mapping template should be given the flexibility to handle the invocation error and decide whether to ignore it, re-raise it, or throw a different error.

Given the following response mapping template:

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

Previously, with 2017-02-28, in case of an invocation error, the response mapping template was evaluated and the result was placed automatically in the `errors` block of the GraphQL response. With 2018-05-29, we can now choose what to do with the error, re-raise it, raise a different error, or append the error while return data.

## Re-raise an Invocation Error

In the following response template, we raise the same error that came back from the data source.

```
#if ( $ctx.error )
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

In case of an invocation error (for example, `$ctx.error` is present) the response looks like the following:

```
{
  "data": {
    "getPost": null
  },
  "errors": [
    {
      "path": [
        "getPost"
      ],
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "message": "Conditional check failed exception...",
      "locations": [
        {
          "line": 5,
          "column": 5
        }
      ]
    }
  ]
}
```

## Raise a Different Error

In the following response template, we raise our own custom error after processing the error that came back from the data source.

```
#if ( $ctx.error )
    #if ( $ctx.error.type.equals("ConditionalCheckFailedException") )
        ## we choose here to change the type and message of the error for
        ConditionalCheckFailedExceptions
    #end
#end
```

```
        $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)
```

In case of an invocation error (for example, `$ctx.error` is present) the response looks like the following:

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

## Append an Error to Return Data

In the following response template, we append the same error that came back from the data source while returning data back inside the response. This is also known as a partial response.

```
#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
```

In case of an invocation error (for example, `$ctx.error` is present) the response looks like the following:

```
{
  "data": {
    "getPost": {
      "id": "1",
      "title": "A post"
    }
  },
  "errors": [
    {
      "path": [
        "getPost"
      ],

```

```
        "errorType": "ConditionalCheckFailedException",
        "message": "Conditional check failed exception...",
        "locations": [
            {
                "line": 5,
                "column": 5
            }
        ]
    }
}
```

## Migrating from 2017-02-28 to 2018-05-29

Migrating from **2017-02-28** to **2018-05-29** is straightforward. Change the version field on the resolver request mapping template or on the function version object. However, note that **2018-05-29** execution behaves differently from **2017-02-28**, changes are outlined [here \(p. 418\)](#).

## Preserving the same execution behavior from 2017-02-28 to 2018-05-29

In some cases, it is possible to retain the same execution behavior as the **2017-02-28** version while executing a **2018-05-29** versioned template.

## Example: DynamoDB PutItem

Given the following **2017-02-28** DynamoDB PutItem request template:

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "attributeValues" : {
    "baz" : ... typed value
  },
  "condition" : {
    ...
  }
}
```

And the following response template:

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

Migrating to **2018-05-29** changes these templates as follows:

```
{
  "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" : {
```

```
    ...  
  }  
}
```

And changes the response template as follows:

```
## 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)
```

Now that it is your responsibility to handle errors, we chose to raise the same error using `$util.error()` that was returned from DynamoDB. You can adapt this snippet to convert your mapping template to **2018-05-29**, note that if your response template is different you will have to take account of the execution behavior changes.

## Example: DynamoDB GetItem

Given the following **2017-02-28** DynamoDB GetItem request template:

```
{  
  "version" : "2017-02-28",  
  "operation" : "GetItem",  
  "key" : {  
    "foo" : ... typed value,  
    "bar" : ... typed value  
  },  
  "consistentRead" : true  
}
```

And the following response template:

```
## map table attribute postId to field Post.id  
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))  
  
$util.toJson($ctx.result)
```

Migrating to **2018-05-29** changes these templates as follows:

```
{  
  "version" : "2018-05-29", ## Note the new 2018-05-29 version  
  "operation" : "GetItem",  
  "key" : {  
    "foo" : ... typed value,  
    "bar" : ... typed value  
  },  
  "consistentRead" : true  
}
```

And changes the response template as follows:

```
## 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)
```

In the **2017-02-28** version, if the datasource invocation was `null`, meaning there is no item in the DynamoDB table that matches our key, the response mapping template would not execute. It might be fine for most of the cases, but if you expected the `$ctx.result` to not be `null`, you now have to handle that scenario.

## 2017-02-28

### Characteristics

- If the datasource invocation result is `null`, the response mapping template is **not** executed.
- If the datasource invocation yields an error, the response mapping template is executed and the evaluated result is placed inside the GraphQL response `errors.data` block.

# Troubleshooting and Common Mistakes

This section discusses some common errors and how to troubleshoot them.

## Incorrect DynamoDB Key Mapping

If your GraphQL operation returns the following error message, it may be because your request mapping template structure doesn't match the Amazon DynamoDB key structure:

```
The provided key element does not match the schema (Service: AmazonDynamoDBv2; Status Code: 400; Error Code
```

For example, if your DynamoDB table has a hash key called "id" and your template says "PostID", as in the following example, this results in the preceding error, because "id" doesn't match "PostID".

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

## Missing Resolver

If you execute a GraphQL operation, such as a query, and get a null response, this may be because you don't have a resolver configured.

For example, if you import a schema that defines a `getCustomer(userId: ID!): field`, and you haven't configured a resolver for this field, then when you execute a query such as `getCustomer(userId: "ID123"){...}`, you'll get a response such as the following:

```
{
  "data": {
    "getCustomer": null
  }
}
```

## Mapping Template Errors

If your mapping template isn't properly configured, you'll receive a GraphQL response whose `errorType` is `MappingTemplate`. The message field should indicate where the problem is in your mapping template.



For example, if you don't have an operation field in your request mapping template, or if the operation field name is incorrect, you'll get a response like the following:

```
{
  "data": {
    "searchPosts": null
  },
  "errors": [
    {
      "path": [
        "searchPosts"
      ],
      "errorType": "MappingTemplate",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "message": "Value for field '$[operation]' not found."
    }
  ]
}
```

## Incorrect Return Types

The return type from your data source must match the defined type of an object in your schema, otherwise you may see a GraphQL error like:

```
"errors": [
  {
    "path": [
      "posts"
    ],
    "locations": null,
    "message": "Can't resolve value (/posts) : type mismatch error, expected type LIST, got OBJECT"
  }
]
```

For example this could occur with the following query definition:

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

Which expects a LIST of [Posts] objects. For example if you had a Lambda function in Node.JS with something like the following:

```
const result = { data: data.Items.map(item => { return item ; }) };
callback(err, result);
```

This would throw an error as `result` is an object. You would need to either change the callback to `result.data` or alter your schema to not return a LIST.