AWS AppSync

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

Welcome ........................................................................................................................................... 1
Quickstart ......................................................................................................................................... 2
Launch a Sample Schema ................................................................................................................ 2
- Launch a Sample Schema ........................................................................................................... 2
- Taking a Tour of the Console ...................................................................................................... 2
- Schema Designer ....................................................................................................................... 3
- Resolver Configuration .................................................................................................................. 3
- Settings ....................................................................................................................................... 3
- Queries ....................................................................................................................................... 3
Run Queries and Mutations ............................................................................................................. 3
- Add Data with a GraphQL Mutation ........................................................................................... 3
- Retrieve Data with a GraphQL Query ............................................................................................ 4
- Running an Application ................................................................................................................. 4
- Next Steps .................................................................................................................................... 5
System Overview and Architecture ................................................................................................. 6
- Architecture .................................................................................................................................. 6
- Concepts ...................................................................................................................................... 6
- GraphQL Proxy ............................................................................................................................. 6
- Operation ..................................................................................................................................... 6
- Action .......................................................................................................................................... 6
- Data Source ................................................................................................................................. 6
- Resolver ....................................................................................................................................... 7
- Identity ......................................................................................................................................... 7
- AWS AppSync Client .................................................................................................................... 7
GraphQL Overview .......................................................................................................................... 8
Designing a GraphQL API .................................................................................................................. 10
- Designing Your Schema ............................................................................................................... 11
- Creating an Empty Schema ......................................................................................................... 11
- Adding a Root Query Type ......................................................................................................... 11
- Defining a Todo Type ................................................................................................................... 11
- Adding a Mutation Type ............................................................................................................. 12
- Modifying the Todo with a Status .............................................................................................. 12
- Subscriptions ............................................................................................................................... 13
- Further Reading ........................................................................................................................... 13
- Advanced – Relations and Pagination .......................................................................................... 13
- Interfaces and Unions in GraphQL ............................................................................................... 14
Attaching a Data Source ................................................................................................................... 21
- (Recommended) Automatic Provision ....................................................................................... 21
- Adding a Data Source .................................................................................................................. 22
Configuring Resolvers ...................................................................................................................... 23
- Create Your First Resolver ......................................................................................................... 23
- Adding a Resolver for Mutations ................................................................................................. 23
- Advanced Resolvers ................................................................................................................... 23
- Test and Debug Resolvers .......................................................................................................... 25
Using Your API ................................................................................................................................. 27
- (Optional) Provision from Schema ............................................................................................. 27
- Schema ......................................................................................................................................... 27
- Provision from Schema .................................................................................................................. 27
- (Optional) Import from Amazon DynamoDB .............................................................................. 28
- Import a DynamoDB Table .......................................................................................................... 29
- Example Schema from Import ...................................................................................................... 29
Building a Client App ...................................................................................................................... 32
- Building a ReactJS Client App ..................................................................................................... 32
- Before You Begin .......................................................................................................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Sources and Resolvers</td>
<td>91</td>
</tr>
<tr>
<td>Tutorial: DynamoDB Resolvers</td>
<td>91</td>
</tr>
<tr>
<td>Setting up Your DynamoDB Tables</td>
<td>91</td>
</tr>
<tr>
<td>Creating Your GraphQL API</td>
<td>92</td>
</tr>
<tr>
<td>Defining a Basic &quot;Post&quot; API</td>
<td>92</td>
</tr>
<tr>
<td>Configuring the Data Source for the DynamoDB Tables</td>
<td>93</td>
</tr>
<tr>
<td>Setting up the &quot;addPost&quot; resolver (DynamoDB PutItem)</td>
<td>93</td>
</tr>
<tr>
<td>Setting up the &quot;getPost&quot; Resolver (DynamoDB GetItem)</td>
<td>96</td>
</tr>
<tr>
<td>Create an updatePost mutation (DynamoDB UpdateItem)</td>
<td>98</td>
</tr>
<tr>
<td>Modifying the &quot;updatePost&quot; resolver (DynamoDB UpdateItem)</td>
<td>100</td>
</tr>
<tr>
<td>Create upvotePost and downvotePost mutations (DynamoDB UpdateItem)</td>
<td>104</td>
</tr>
<tr>
<td>Setting up the &quot;deletePost&quot; resolver (DynamoDB DeletePost)</td>
<td>107</td>
</tr>
<tr>
<td>Setting up the &quot;allPost&quot; resolver (DynamoDB Scan)</td>
<td>112</td>
</tr>
<tr>
<td>Setting up the &quot;allPostsByAuthor&quot; resolver (DynamoDB Query)</td>
<td>115</td>
</tr>
<tr>
<td>Setting up the &quot;allPost&quot; resolver (DynamoDB Scan)</td>
<td>112</td>
</tr>
<tr>
<td>Building an Android Client App</td>
<td>73</td>
</tr>
<tr>
<td>Building an iOS Client App</td>
<td>59</td>
</tr>
<tr>
<td>Building a React Native Client App</td>
<td>45</td>
</tr>
<tr>
<td>Building a JavaScript Client App</td>
<td>55</td>
</tr>
<tr>
<td>Building a Java Client App</td>
<td>33</td>
</tr>
<tr>
<td>Building an iOS Client App</td>
<td>59</td>
</tr>
<tr>
<td>Building a React Native Client App</td>
<td>45</td>
</tr>
<tr>
<td>Building a JavaScript Client App</td>
<td>55</td>
</tr>
<tr>
<td>Building an iOS Client App</td>
<td>33</td>
</tr>
<tr>
<td>Creating Your GraphQL API</td>
<td>33</td>
</tr>
<tr>
<td>Creating Your GraphQL API</td>
<td>33</td>
</tr>
<tr>
<td>Create an API</td>
<td>59</td>
</tr>
<tr>
<td>Code Generation for the API</td>
<td>60</td>
</tr>
<tr>
<td>Set up Dependency on the AWS AppSync SDK</td>
<td>62</td>
</tr>
<tr>
<td>Convert the App to Use AWS AppSync for the Backend</td>
<td>63</td>
</tr>
<tr>
<td>Make Your App Real Time</td>
<td>67</td>
</tr>
<tr>
<td>Complex Objects</td>
<td>41</td>
</tr>
<tr>
<td>Offline Settings</td>
<td>38</td>
</tr>
<tr>
<td>Conflict Resolution</td>
<td>44</td>
</tr>
<tr>
<td>Conflict Resolution</td>
<td>44</td>
</tr>
<tr>
<td>Conflict Resolution</td>
<td>33</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Using Sets</td>
<td>119</td>
</tr>
<tr>
<td>Using Lists and Maps</td>
<td>124</td>
</tr>
<tr>
<td>Conclusion</td>
<td>126</td>
</tr>
<tr>
<td>Tutorial: Lambda Resolvers</td>
<td>126</td>
</tr>
<tr>
<td>Create a Lambda Function</td>
<td>127</td>
</tr>
<tr>
<td>Configure data source for AWS Lambda</td>
<td>128</td>
</tr>
<tr>
<td>Creating a GraphQL Schema</td>
<td>129</td>
</tr>
<tr>
<td>Configuring resolvers</td>
<td>23</td>
</tr>
<tr>
<td>Testing your GraphQL API</td>
<td>131</td>
</tr>
<tr>
<td>Returning Errors</td>
<td>132</td>
</tr>
<tr>
<td>Advanced Use Case: Batching</td>
<td>134</td>
</tr>
<tr>
<td>Tutorial: Amazon Elasticsearch Service Resolvers</td>
<td>139</td>
</tr>
<tr>
<td>One-Click Setup</td>
<td>139</td>
</tr>
<tr>
<td>Create a New Amazon ES Domain</td>
<td>139</td>
</tr>
<tr>
<td>Configure Data Source for Amazon ES</td>
<td>139</td>
</tr>
<tr>
<td>Connecting a Resolver</td>
<td>141</td>
</tr>
<tr>
<td>Modifying Your Searches</td>
<td>142</td>
</tr>
<tr>
<td>Adding Data to Amazon ES</td>
<td>143</td>
</tr>
<tr>
<td>Retrieving a Single Document</td>
<td>143</td>
</tr>
<tr>
<td>Perform Queries and Mutations</td>
<td>144</td>
</tr>
<tr>
<td>Best Practices</td>
<td>144</td>
</tr>
<tr>
<td>Tutorial: Local Resolvers</td>
<td>144</td>
</tr>
<tr>
<td>Create the Paging Application</td>
<td>145</td>
</tr>
<tr>
<td>Send and subscribe to pages</td>
<td>146</td>
</tr>
<tr>
<td>Tutorial: Combining GraphQL Resolvers</td>
<td>146</td>
</tr>
<tr>
<td>Example Schema</td>
<td>146</td>
</tr>
<tr>
<td>Alter data through resolvers</td>
<td>147</td>
</tr>
<tr>
<td>DynamoDB and Amazon ES</td>
<td>148</td>
</tr>
<tr>
<td>Tutorial: DynamoDB Batch Resolvers</td>
<td>150</td>
</tr>
<tr>
<td>Permissions</td>
<td>151</td>
</tr>
<tr>
<td>Data Source</td>
<td>6</td>
</tr>
<tr>
<td>Single Table Batch</td>
<td>152</td>
</tr>
<tr>
<td>Multi-Table Batch</td>
<td>154</td>
</tr>
<tr>
<td>Error Handling</td>
<td>159</td>
</tr>
<tr>
<td>Real-Time Data</td>
<td>164</td>
</tr>
<tr>
<td>GraphQL Schema Subscription Directives</td>
<td>164</td>
</tr>
<tr>
<td>Using Subscription Arguments</td>
<td>165</td>
</tr>
<tr>
<td>Security</td>
<td>167</td>
</tr>
<tr>
<td>API_KEY Authorization</td>
<td>77</td>
</tr>
<tr>
<td>AWS_IAM Authorization</td>
<td>168</td>
</tr>
<tr>
<td>OPENID_CONNECT Authorization</td>
<td>169</td>
</tr>
<tr>
<td>AMAZON_COGNITO_USER_POOLS Authorization</td>
<td>169</td>
</tr>
<tr>
<td>Fine-Grained Access Control</td>
<td>170</td>
</tr>
<tr>
<td>Filtering Information</td>
<td>172</td>
</tr>
<tr>
<td>Authorization Use Cases</td>
<td>173</td>
</tr>
<tr>
<td>Overview</td>
<td>173</td>
</tr>
<tr>
<td>Reading Data</td>
<td>174</td>
</tr>
<tr>
<td>Writing Data</td>
<td>176</td>
</tr>
<tr>
<td>Public and Private Records</td>
<td>178</td>
</tr>
<tr>
<td>Real-Time Data</td>
<td>164</td>
</tr>
<tr>
<td>Resolver Mapping Template Reference</td>
<td>182</td>
</tr>
<tr>
<td>Resolver Mapping Template Overview</td>
<td>182</td>
</tr>
<tr>
<td>Example Template</td>
<td>183</td>
</tr>
<tr>
<td>Resolver Mapping Template Programming Guide</td>
<td>184</td>
</tr>
<tr>
<td>Setup</td>
<td>184</td>
</tr>
<tr>
<td>Variables</td>
<td>185</td>
</tr>
<tr>
<td>Calling Methods</td>
<td>187</td>
</tr>
</tbody>
</table>
Welcome

- Are you a first-time AWS AppSync user? (p. 2)
- Are you developing a mobile application? (p. 10)
- Are you adding a GraphQL API to existing AWS resources? (p. 91)

This is the AWS AppSync Developer Guide.

AWS AppSync is an enterprise-level, fully managed GraphQL service with real-time data synchronization and offline programming features.

This guide focuses on using AWS AppSync to create and interact with data sources by using GraphQL from your application. Developers who want to build applications using GraphQL with robust database, search, and compute capabilities, will find the information they need to build an application or integrate existing data sources with AWS AppSync.
Quickstart

This section describes how to use the AWS AppSync console to launch a sample schema, automatically provision a DynamoDB data source and connect resolvers, write GraphQL queries and mutations, and use the API in a sample app.

You can also start 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. 11). Alternatively, you can import from an existing Amazon DynamoDB table to create a real-time and offline GraphQL API (p. 28).

Topics
• Launch a Sample Schema (p. 2)
• Run Queries and Mutations (p. 3)

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 sample schema enables users to create an application where events ("Going to the movies" or "Dinner at Mom & Dad's") can be entered. Application users can also comment on the events ("See you at 7!"). This app demonstrates using GraphQL operations where state is persisted in Amazon DynamoDB.

To start, you'll create a sample schema and provision it.

To create the API

   <step>
   Choose Create API from the Dashboard.
   </step>
2. Type a friendly application name.
   <step>
   At the bottom of the console window, select the Sample schema.
   </step>
3. Choose Create and wait for the provisioning process to complete.

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 Getting Started, which has information such as your endpoint URL and authorization mode.

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 or user pools. For more information, see Security (p. 167).
This page has a listing of sample client applications (JavaScript, iOS, etc.) for testing an end-to-end experience. You can clone and download these, as well as the configuration file that has the necessary information (such as your endpoint URL) to get started. Then, follow the instructions on the page to run your app.

Schema Designer

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

On the right side of the console are the GraphQL types that have been created, as well as 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 click the Create Resources button at the top and select one of these user-defined types from the drop-down menu, you can see how selecting different field options populates a schema design. Don’t select anything now, but try this in the future when you design a schema (p. 11).

Resolver Configuration

From 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 (with a full listing on the Data Sources tab of the console) for a resolver, as well as 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 Elasticsearch Service.

Settings

The Settings tab is where you configure things like the authorization method for your API. For more information on these options, see the security overview (p. 167).

Queries

A built-in designer for writing and running GraphQL queries and mutations, including introspection and documentation, is included in the console. We’ll cover that next.

Run Queries and Mutations

In the AWS AppSync console, choose the Queries tab on the left to open the GraphQL operations interface. First, note the pane on the right side that enables you to click through the operations, including queries, mutations, and subscriptions that your schema has exposed. Choose the Mutation node, and you see a mutation and can add a new event to it: createEvent(....):Event. Use this to add something to your database with GraphQL.

Add Data with a GraphQL Mutation

Because there isn’t any data yet, the first step is to add some with a GraphQL mutation. You do this with the mutation keyword, passing in the appropriate arguments. This works similarly to a function. You can also select which data you want to be returned in the response by putting the fields inside curly braces. Paste the following into the query editor and choose Run:
mutation {
    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`; however, one was generated and returned in the results specified between the curly braces. This is because the sample demonstrates an `autoId()` function in a GraphQL resolver as a best practice for the partition key set on your DynamoDB resources. For now, just make a note of the returned `id` value for use in the next section.

**Retrieve Data with a GraphQL Query**

Now that there is a record in your database, running a query returns some results. 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 press the Run button at the top:

```graphql
query {
    getEvent(id:"XXXXXX-XXXX-XXXXXXX-XXXX-XXXXXXXXX"){
        name
        where
        description
    }
}
```

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

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

This time the query shows nested types as well as giving the query a friendly name (`getAllEvents`), which is optional. Experiment by adding or removing and then rerunning the query. When you're done, it's time to connect a client application.

**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. In the AWS AppSync console, at the root of the navigation, select the name of your API, and you will see a list of platforms. Clone the appropriate
sample to your local workstation, download the configuration file and, if necessary, the GraphQL introspection schema.json file (used on some platforms like iOS and Android for code generation). 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. See Security (p. 167) for more information.

**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. 10)
- Connecting Data Sources and Resolvers (p. 91)
- Building Client Applications (p. 32)
System Overview and Architecture

AWS AppSync allows developers to interact with their data via 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.
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.

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. Learn more about this process [here](p. 11).

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:

```graphql
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 "co-located" 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. 10)
- Data Sources and Resolvers Tutorial (p. 91)
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 describe building a schema from scratch, provisioning resources automatically, manually defining a data source, and connecting 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.

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, and Amazon Elasticsearch Service 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. 27) from a schema or create a schema and connect resolvers from an existing table (p. 28) 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. 184) and helper utilities available to use when programming can be found in aws-appsync-resolver-context-reference. AWS AppSync also has built-in test & debug flows when editing or authoring from scratch which you can read about in Test and Debug Resolvers (p. 25).

Topics

- Designing Your Schema (p. 11)
- Attaching a Data Source (p. 21)
- Configuring Resolvers (p. 23)
- Using Your API (p. 27)
- (Optional) Provision from Schema (p. 27)
- (Optional) Import from Amazon DynamoDB (p. 28)
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:

```graphql
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:

```graphql
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:

```graphql
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. 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. You’ll see this later.

There are similarities between the Query and Todo types. In GraphQL, the root types (Query, Mutation, and Subscription) are just types like the ones you define. They’re special, though, 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
    status: TodoStatus
}
```

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

Modifying the Todo with a Status

At this point, your GraphQL API is structurally functioning 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
}
```
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.

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. Please see Real-Time Data (p. 164) for more information on configuring subscriptions.

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:

- Add a new TodoConnection type, which has todos and nextToken fields.
- Add two input arguments, first and after, to the getTodos field.
- Change getTodos so that it returns TodoConnection.

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

type Query {
  getTodos(first: Int = 20, after: 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. In AWS AppSync, this is connected to Amazon DynamoDB with a mapping template.
This converts the value of the first argument to the maxResults parameter and the after argument to the exclusiveStartKey parameter. See Resolver Mapping Template Reference (p. 182) for examples or the built in template samples in the AWS AppSync console.

Next, suppose your todos have comments, and you want to run a query that returns all the comments for a todo. 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:

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

type Query {
    getTodos(first: Int = 20, after: String): TodoConnection
}

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

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

type Comment {
    id: ID!
    content: String
}

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

type TodoStatus {
    done
    pending
}
```

The application graph on top of your existing data sources in AWS AppSync allows 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 this is done in Configuring Resolvers (p. 23).

## Interfaces and Unions in GraphQL

### Interfaces

GraphQL's type system features 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, skip this section and come back to it at a later time when you want to evolve your schema or add more features, and instead look at automatically create and connect resolvers (p. 27) 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, they all have 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 SDL, our `Event` interface would be:

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

And each of the types implements the `Event` interface:

```graphql
type Concert implements Event {
  id: ID!
  title: String!
  startsAt: String!
  endsAt: String!
  venue: Venue
  minAgeRestriction: Int
  performingBand: String
}

type Festival implements Event {
  id: ID!
  title: String!
  startsAt: String!
  endsAt: String!
  venue: Venue
  minAgeRestriction: Int
  performers: [String]
}

type Conference implements Event {
  id: ID!
  title: 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 on the schema:

```graphql
schema {
  query: Query
}

type Query {
  # Retrieve Events at a specific Venue
  findEventsAtVenue(venueId: ID!): [Event]
}
```
findEventsByVenue returns a list of Event. Because GraphQL interface fields are common to all the implementing types, you probably guessed it is possible to select any fields on the Event interface (id, title, startsAt, endsAt, venue, and minAgeRestriction). Additionally, we can access the fields on any implementing type, as long as we specify the type, using GraphQL fragments.

Let’s look at an example of a GraphQL query that uses our interface.

```graphql
query {
  findEventsAtVenue(venueId: "Madison Square Garden") {
    id
    name
    minAgeRestriction
    startsAt
    ... on Festival {
      performers
    }
  }
}
```
... on Concert {
    performingBand
}

... on Conference {
    speakers
    workshops
}
}

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

```
{
    "data": {
        "findEventsAtVenue": [
            {
                "id": "Festival-2",
                "name": "Festival 2",
                "minAgeRestriction": 21,
                "startsAt": "2018-10-05T14:48:00.000Z",
                "performers": [
                    "The Singers",
                    "The Screamers"
                ]
            },
            {
                "id": "Concert-3",
                "name": "Concert 3",
                "minAgeRestriction": 18,
                "startsAt": "2018-10-07T14:48:00.000Z",
                "performingBand": "The Jumpers"
            },
            {
                "id": "Conference-4",
                "name": "Conference 4",
                "minAgeRestriction": null,
                "startsAt": "2018-10-09T14:48:00.000Z",
                "speakers": [
                    "The Storytellers"
                ],
                "workshops": [
                    "Writing",
                    "Reading"
                ]
            }
        ]
    }
}
```

As you can see, results are returned as a single collection of events. Using interfaces to represent common characteristics will be very handy for sorting results.

**Unions**

GraphQL's type system also features **Unions**. Unions are identical to Interfaces, except they do not 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 our `Event` schema, we can define a `SearchResult` union:
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, we must use fragments. Let's look at an example:

```graphql
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 our SearchResult union defined. (i.e., 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 our 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, select Attach Resolver, create a new Data Source of type NONE, and then name it StubDataSource. For the sake of this example, we will pretend we fetched results from an external source, and hardcode the fetched results in our request mapping template.
In the request mapping template pane, enter:

```json
{
  "version": "2017-02-28",
  "payload":
  ## We are effectively mocking our search results for this example
  [
    {
      "id": "Venue-1",
      "name": "Venue 1",
      "address": "2121 7th Ave, Seattle, WA 98121",
      "maxOccupancy": 1000
    },
    {
      "id": "Festival-2",
      "name": "Festival 2",
      "performers": ["The Singers", "The Screamers"]
    },
    {
      "id": "Concert-3",
      "name": "Concert 3",
      "performingBand": "The Jumpers"
    },
    {
      "id": "Conference-4",
      "name": "Conference 4",
      "speakers": ["The Storytellers"],
      "workshops": ["Writing", "Reading"]
    }
  ]
}
```

In our application, we chose to return the type name as part of the id field. Our type resolution logic only consists of parsing the id field to extract the type name and adding the __typename field to each of the results. We can easily perform that logic in the response mapping template (you can also perform this task as part of your Lambda function, if you are using the Lambda Data source):

```template
#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)
```

Running our query,

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }
    ... on Festival {
      id
      name
      performers
    }
    ... on Concert {
      id
    }
  }
}
```
Attaching a Data Source

(Recommended) Automatic Provision

Continuing on from Designing Your Schema (p. 11), you can have AWS AppSync automatically create tables based on your schema definition. This is an optional but recommended step. AWS AppSync will

```json
{  
  "data": {  
    "search": [  
      {  
        "id": "Venue-1",  
        "name": "Venue 1",  
        "address": "2121 7th Ave, Seattle, WA 98121"  
      },  
      {  
        "id": "Festival-2",  
        "name": "Festival 2",  
        "performers": [  
          "The Singers",  
          "The Screamers"  
        ]  
      },  
      {  
        "id": "Concert-3",  
        "name": "Concert 3",  
        "performingBand": "The Jumpers"  
      },  
      {  
        "speakers": [  
          "The Storytellers"  
        ],  
        "workshops": [  
          "Writing",  
          "Reading"  
        ]  
      }  
    ]  
  }  
}
```

Naturally, the type resolution logic will vary depending on the application. For example, we could have a different identifying logic that checks for the existence of certain fields or even a combination of fields. That is, we 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 what the logic will be.
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 hash key called "id" (and if you're doing the advanced section with Relations, you also need a table named "Comments" with a hash key of "todoid" and a sort key of "content").

To add your data source

1. Choose the Data Sources tab in the console, and choose New.
   <step>Give your data source a friendly name, such as "Todos table".</step>
2. Choose Amazon DynamoDB Table as the type.
   <step>Choose the appropriate region.</step>
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 will need a trust policy outlined below.

If you're doing the advanced section, repeat the process. Note that you need IAM permissions of PutItem and Query on the "Comments" table.

Now that you've connected a data source to an AWS service, you can connect it to your schema with a resolver. See Configuring Resolvers (p. 23).

IAM Trust Policy

If you are using an existing IAM role for your data source, you will need to grant that role the appropriate permissions to perform operations on your AWS resource, such as PutItem on an Amazon DynamoDB table. You will also need to modify the Trust Policy on that role to allow AWS AppSync to leverage it for resource access as shown in the below example policy.

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

Create Your First Resolver

Navigate back to the Schema page in the AWS AppSync console and find the query type on the right side. Choose the Attach resolver button next to the getTodos field, which opens the Add Resolver page. Select the data source you just created and either use 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 (all item queries, individual lookups, etc.). For example, on the simple version of the schema from Designing Your Schema (p. 11) where getTodos didn't have pagination, the mapping template is as follows:

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

A response mapping template is always needed. The console provides a default with the following passthrough value:

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

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:

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

AWS AppSync will automatically convert arguments defined in the addTodo field from your GraphQL schema into DynamoDB operations. The above example will store 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).

Use the same passthrough template from earlier.

AWS AppSync also supports test & debug workflows for editing resolvers. You can use a mock context object to see the transformed value of the template before invoking, or optionally you can view the full request execution to a data source interactively when you run a query. To learn more about this see Test and Debug Resolvers (p. 25) and Monitoring and Logging (p. 249).

Advanced Resolvers

If you are following the Advanced section of building a sample schema in Designing Your Schema (p. 11), to do a paginated scan, you should use the following template:
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 would be:

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

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.

**Note:** The fact that this uses a query operation against a second table is only for illustrative purposes. It could also be another operation against DynamoDB. Further, the data could be pulled from another data source, such as AWS Lambda or Amazon Elasticsearch Service because the relation is controlled by your GraphQL schema.

From the **Schema** page in the console, click the comments field on the Todo type, and then choose **Attach resolver**. Use the following request mapping template:

```
{
  "version" : "2017-02-28",
  "operation" : "Scan",
  "key": {
    "todoid" : { "S" : '${context.source.id}' }
  },
}
```

Pay attention to "context.source". This references the parent object of the current field being resolved. In this example, "source" is referring to the Todo object, which contains the comments you are fetching.

You can use the passthrough response mapping template.

Finally, create an **addComment** resolver from the schema page in the console, just like you did for the preceding fields. The request mapping template in this case is a simple **PutItem**:

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "todoid" : { "S" : '${context.arguments.todoid}' },
    "content" : { "S" : '${context.arguments.content}' }
  }
}
```

In the preceding example, the key corresponds to the arguments from the **addComment** mutation. Use a passthrough response.
Test and Debug Resolvers

AWS AppSync executes resolvers on a GraphQL field against a data source. As described in Resolver Mapping Template Overview (p. 182), 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. You can find an introductory tutorial-style programming guide for writing resolvers in the Resolver Mapping Template Programming Guide (p. 184).

To help developers write, test, and debug these resolvers, the AWS AppSync console also provides tools to "mock" a GraphQL request and response, 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. Find more details about this structure and the available helper utilities to use when programming in the Resolver Mapping Template Context Reference (p. 195).

When writing or editing a resolver function, you can pass a "mock" or "test" context object into the console editor and 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, open a resolver by navigating to the Schema page, and selecting an existing resolver on the right to edit. Or click Attach to add a new resolver. At the top of the page, choose Select test context, and then choose Create new context. Provide a memorable name. Next, either select from an existing sample context object or populate the JSON manually, then press Save. You can then press the Test button to evaluate your resolver using this mocked context object.

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 a response to be seen only for a certain user. Here’s what the schema might look like:

```typescript
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 below. This has arguments from the client of name and age, as well as a username populated in the identity object:

```json
{
  "arguments" : {
    "firstname": "Shaggy",
```
You can test this using the following request and response mapping templates:

**Request Template**

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

**Response Template**

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

The evaluated template will have 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 would not be returned because the authorization check would fail. Read more about fine-grained authorization in Authorization Use Cases (p. 173).

**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 displayed logs are the connection-time information.

To perform this, you need to have Amazon CloudWatch logs enabled in advance, as described in Monitoring and Logging (p. 249). Next, navigate to the Queries tab of the AWS AppSync console, and type in a valid GraphQL query. In the lower-right section, you'll see a Logs check box, which you can select to drag the view up. Using the "play" Arrow at the top of the page, run your GraphQL query. In a few moments, your full request and response logs for the operation will be streamed to this section of the console for you to view.
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. Navigate to the Queries tab in the console and enter the following text in the editor:

```graphql
mutation add {
        id
        name
        description
        priority
    }
}
```

Press the button at the top to run your mutation. After it completes, the result from your selection set (id, name, description, and priority) are displayed 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:

```graphql
query {
    allTodo {
        id
        name
    }
}
```

This should return your data, but just the two fields (id and name) from your selection set.

(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.

Schema

These instructions start with the schema outlined in Designing Your Schema (p. 11), as shown next:

```graphql
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!
}
```
Provision from Schema

After you save a schema, a Create resources button appears in the upper right of the page. Click this to go to the Create resources page. You can select any user-defined GraphQL types from the screen, and your Todo type should be available. Select this type and you'll see a form you can use to configure the table details. You can change your DynamoDB primary or sort keys here, as well as add additional indexes. At the bottom of the page is a corresponding section for the GraphQL queries and mutations that are then 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. An index selection is also available. You can use it for different query options, which set up a DynamoDB Local Secondary Index or Global Secondary Indexes, as appropriate.

For the preceding example schema, you can simply have id selected as the primary key and press the Create button. 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 the Using Your API (p. 27) section. Note that there will be a GraphQL input type for the arguments of the created schema. For example if you provision from schema with a GraphQL type Books {...} then there might be an input type like so:

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

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

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

(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 for which 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, navigate to the Data Sources page and select the New button. Give your data source a friendly name, and select Amazon DynamoDB as the data source type. Select the appropriate table, then toggle the switch under Automatically generate GraphQL.

You'll see two code editors with GraphQL schema:

- The top editor can be manipulated 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).
- The bottom editor is read-only and contains generated GraphQL schema snippets, showing what types, queries, and mutations will be merged into your schema. If you edit the type in the top editor, this will change as appropriate.

Press Create at the bottom and your schema is merged and resolvers are created. After this is complete, you can run mutations and queries as described in the Using Your API (p. 27) section. **Note** that there will be a GraphQL input type for the arguments of the created schema. For example if you import a table called "Books" then there might be an input type like so:

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

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

```graphql
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:

```json
{
  Table: {
    AttributeDefinitions: [
      {
        AttributeName: 'authorId',
        AttributeType: 'S'
      },
      {
        AttributeName: 'bookId',
```
Example Schema from Import

```
{
  AttributeName: 'authorId',
  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 will show 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 the non-key attributes at the bottom like `isPublished` need to be added manually as they cannot be inferred from DynamoDB automatically. For instance if you had another attribute on an item in your DynamoDB table called `rating` you would need to add it under `isPublished` to have it populated in the GraphQL schema. 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

The following sections are tutorials for building a client application with GraphQL on different platforms. Each tutorial starts with an application running with local data, and then adds in the AWS AppSync SDK to communicate with your GraphQL API. The tutorials assume you have a basic schema created using the schema from the DynamoDB resolvers tutorial (p. 91) as a reference starting point, which you can optionally complete first.

Topics
- Building a ReactJS Client App (p. 32)
- Building a React Native Client App (p. 45)
- Building a JavaScript Client App (p. 55)
- Building an iOS Client App (p. 59)
- Building an Android Client App (p. 73)

Building a ReactJS Client App

AWS AppSync integrates with the Apollo GraphQL client for building client applications. AWS provides Apollo plugins for offline support, authorization, and subscription handshaking. You can use the Apollo client directly, or you can use it with some of the client helpers provided in the AWS AppSync SDK. This
This tutorial shows you how to use AWS AppSync with React Apollo, which uses ReactJS constructs and patterns with GraphQL.

**Before You Begin**

This tutorial is set up for a sample API using the schema from the DynamoDB resolvers tutorial (p. 91). To follow along with the complete flow, you can optionally walk through that tutorial first. If you want to do more customization of GraphQL resolvers, such as those that use DynamoDB, see the Resolver Mapping Template Reference (p. 182). The application will use the following starting schema:

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

type Mutation {
  addPost(id: ID! author: String! title: String content: String! url: String!): Post!
  deletePost(id: ID!, expectedVersion: Int): Post
}

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

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

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

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

**Get the GraphQL API Endpoint**

After you create your GraphQL API, you’ll need to get the API endpoint (URL) so you can use it in your client application. You can get the API endpoint in two ways.

In the AWS AppSync console, choose **Home** and then choose **GraphQL URL** to see the API endpoint.

Alternatively, you can get it by running the following CLI command:

```
aws appsync get-graphql-api --api-id $GRAPHQL_API_ID
```

The following instructions show how you can use **AWS_IAM** for client authorization. In the AWS AppSync console, choose **Settings** on the left, and then choose **AWS_IAM**. For more information about authorization modes, see Security (p. 167).
Download a Client Application

To show you how to use AWS AppSync, we first review a React application with just a local array of data. Then we add AWS AppSync capabilities to it. To begin, download a sample application where we can add, update, and delete posts.

Understanding the React Sample App

The React sample app has three major files:

- `./src/App.js`: The main entry point of the application. It renders the main application shell with two components named AddPost and AllPosts, and has a local array of data named posts which is passed as a prop to the other components.
- `./src/Components/AddPost`: A React component that contains a form that enables a user to enter new information about a post, such as the author and title.
- `./src/Components/AllPosts`: A React component that lists all existing posts from the posts array that App.js created. It enables you to edit or delete existing posts.

Run your app as follows, and test it to be sure it works:

```
yarn && yarn start
```

Import the AWS AppSync SDK into Your App

In this section, you'll add AWS AppSync to your existing app.

Add the following dependencies to your application:

```
yarn add react-apollo graphql-tag aws-sdk
```

Next, add in the AWS AppSync SDK, including the React extensions:

```
yarn add aws-appsync
cyarn add aws-appsync-react
```

From the AWS AppSync console, navigate to your GraphQL API landing page where the API URL is listed. At the bottom of the page, choose Web. Next, click the Download button and save the AppSync.js configuration file into ./src.

To interact with AWS AppSync, your client needs to define GraphQL queries and mutations. This is commonly done in separate files, as follows:

```
mkdir ./src/Queries
touch ./src/Queries/AllPostsQuery.js
touch ./src/Queries/DeletePostMutation.js
touch ./src/Queries/NewPostMutation.js
touch ./src/Queries/UpdatePostMutation.js
```

Edit and save AllPostsQuery.js:

```
import gql from 'graphql-tag';

export default gql`
query AllPosts {  
`
allPost {
  posts {
    __typename
    id
title
author
version
  }
}
};

Edit and save DeletePostMutation.js:

```javascript
import gql from 'graphql-tag';

export default gql`
mutation DeletePostMutation($id: ID!, $expectedVersion: Int!) {
  deletePost(id: $id, expectedVersion: $expectedVersion) {
    __typename
    id
author
title
version
  }
}
`;
```

Edit and save NewPostMutation.js:

```javascript
import gql from 'graphql-tag';

export default gql`
mutation AddPostMutation($id: ID!, $author: String!, $title: String!) {
  addPost(
    id: $id
    author: $author
    title: $title
    content: ""
    url: ""
  ) {
    __typename
    id
    author
    title
    version
  }
}
`;
```

Edit and save UpdatePostMutation.js:

```javascript
import gql from 'graphql-tag';

export default gql`
mutation UpdatePostMutation($id: ID!, $author: String, $title: String, $expectedVersion: Int!) {
  updatePost(
    id: $id
    author: $author
    title: $title
    expectedVersion: $expectedVersion
  ) {
    __typename
    id
  }
}
```

35
Edit your App.js file, as follows:

```javascript
import AWSAppSyncClient from "aws-appsync";
import { Rehydrated } from 'aws-appsync-react';
import { AUTH_TYPE } from "aws-appsync/lib/link/auth-link";
import { graphql, ApolloProvider, compose } from 'react-apollo';
import * as AWS from 'aws-sdk';
import AppSync from './AppSync.js';
import AllPostsQuery from './Queries/AllPostsQuery';
import NewPostMutation from './Queries/NewPostMutation';
import DeletePostMutation from './Queries/DeletePostMutation';
import UpdatePostMutation from './Queries/UpdatePostMutation';

After all the import statements, add the following code:

```javascript
const client = new AWSAppSyncClient({
  url: AppSync.graphqlEndpoint,
  region: AppSync.region,
  auth: {
    type: AUTH_TYPE.API_KEY,
    apiKey: AppSync.apiKey,
    // type: AUTH_TYPE.AWS_IAM,
    // Note - Testing purposes only
    // credentials: new AWS.Credentials({
    //   accessKeyId: AWS_ACCESS_KEY_ID,
    //   secretAccessKey: AWS_SECRET_ACCESS_KEY
    // })*/

    // Amazon Cognito Federated Identities using AWS Amplify
    // credentials: () => Auth.currentCredentials(),

    // Amazon Cognito user pools using AWS Amplify
    // type: AUTH_TYPE.AMAZON_COGNITO_USER_POOLS,
    // jwtToken: async () => (await Auth.currentSession()).getIdToken().getJwtToken()
  },
});
```

You can switch the AUTH_TYPE value use API keys, IAM (including short-term credentials from Amazon Cognito Federated Identities), or Amazon Cognito user pools. We recommend you use either IAM or Amazon Cognito user pools after onboarding with an API key. The previous code shows how to use the default configuration of AWS AppSync with an API key, referencing the AppSync.js file you downloaded. When you're ready to add other authorization methods to your application, you can use the AWS Amplify library to quickly add these capabilities to your application. The corresponding AWS Amplify methods for the AWS AppSync client constructor are included above. An import of the library with configuration would look similar to the following:

```javascript
import Amplify, { Auth } from 'aws-amplify';
import { withAuthenticator } from 'aws-amplify-react';
Amplify.configure(awsmobile);
//...code

const AppWithAuth = withAuthenticator(App, true);
```
For more information on using AWS Amplify, see the library documentation.

Replace the `App` component entirely, so it looks like this:

```javascript
class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
        <NewPostWithData />
        <AllPostsWithData />
      </div>
    );
  }
}
```

You can also delete the `posts` variable in your code, because the app state will be coming from AWS AppSync.

At the bottom of your `App.js` file, define the following higher-order component (HOC):

```javascript
const AllPostsWithData = compose(
  graphql(AllPostsQuery, {
    options: {
      fetchPolicy: 'cache-and-network'
    },
    props: (props) => ({
      posts: props.data.allPost && props.data.allPost.posts,
    })
  }),
  graphql(DeletePostMutation, {
    props: (props) => {
      onDelete: (post) => props.mutate({
        variables: { id: post.id, expectedVersion: post.version },
        optimisticResponse: () => ({ deletePost: { ...post, __typename: 'Post' } })
      })
    }
  }),
  options: {
    refetchQueries: [{ query: AllPostsQuery }],
    update: (proxy, { data: { deletePost: { id } } }) => {
      const query = AllPostsQuery;
      const data = proxy.readQuery({ query });
      data.allPost.posts = data.allPost.posts.filter(post => post.id !== id);
      proxy.writeQuery({ query, data });
    }
  },
  graphql(UpdatePostMutation, {
    props: (props) => {
      onEdit: (post) => {
        props.mutate({
          variables: { ...post, expectedVersion: post.version },
          optimisticResponse: () => ({ updatePost: { ...post, __typename: 'Post', version: post.version + 1 } })
        })
      }
    }
  })
),
```
Finally, replace export default App with the ApolloProvider:

```javascript
const WithProvider = () => {
  <ApolloProvider client={client}>
    <Rehydrated>
      <App />
    </Rehydrated>
  </ApolloProvider>
};

export default WithProvider;
```

Test Your Application

```bash
yarn start
```

Open a webpage and add, remove, edit, and delete data. If you're using Chrome developer tools, you can use the network conditioning tool for offline testing.

Offline Settings

There are important considerations that you'll need to account for if you want an optimistic UI for an application, where data can be manipulated when the device is in an offline state. Many of these settings
are documented in the official Apollo documentation, however, we call out several of them here that you should configure.

First, know that the AWS AppSync client allows you to disable offline capabilities if you simply want to use GraphQL in an always-online scenario. To do this, you pass an additional option when instantiating your client, named `disableOffline`, as follows:

```javascript
const client = new AWSAppSyncClient({
  url: AppSync.graphqlEndpoint,
  region: AppSync.region,
  auth: {
    type: AUTH_TYPE.API_KEY,
    apiKey: AppSync.apiKey,
  },
  disableOffline: true
});
```

- `fetchPolicy`: This option allows you to specify how a query interacts with the network versus local in-memory caching. AWS AppSync persists this cache to a platform-specific storage medium. If you are using the AWS AppSync client in offline scenarios (`disableOffline: false`), you **MUST** set this value to `cache-and-network`:

  ```javascript
  options: {
    fetchPolicy: 'cache-and-network'
  }
  ```

- `optimisticResponse`: This option allows you to pass a function or an object to a mutation for updating your UI before the server responds with the result. This is needed in offline scenarios (and for slower networks) to ensure that the UI is updated when the device has no connectivity. Optionally, you can also use this if you have set `disableOffline: true`. For example, if you were adding a new object to a list, you might use the following:

  ```javascript
  onAdd: post => props.mutate({
    variables: post,
    optimisticResponse: () => ({ addPost: { __typename: 'Post', ups: 1, downs: 1, content: '', url: '', version: 1, ...post } }),
  })
  ```

Normally, you use `optimisticResponse` in conjunction with the `update` option for React Apollo's component, which can trigger during an offline mutation. If you want the UI to update offline for a specific query, you need to specify that query as part of the `readQuery` and `writeQuery` options on the cache, as shown below:

```javascript
options: {
  refetchQueries: [{ query: AllPostsQuery }],
  update: (dataProxy, { data: { addPost } }) => {
    const query = AllPostsQuery;
    const data = dataProxy.readQuery({ query });
    data.allPost.posts.push(addPost);
    dataProxy.writeQuery({ query, data });
  }
}
```

When this happens, the AWS AppSync persistent store update automatically in response to the Apollo cache update. Upon network reconnection, it will synchronize with your GraphQL endpoint. You could also modify more than one query when offline, in which case you could run the above process multiple times in the same `update` block.
Make Your Application Real Time

Edit your schema with the subscription type, as follows:

```graphql
schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}
type Mutation {
  addPost(id: ID! author: String! title: String content: String! url: String!): Post!
  deletePost(id: ID!, expectedVersion: Int): Post!
}
type Post {
  id: ID!
  author: String!
  title: String
  content: String
  url: String
  ups: Int
  downs: Int
  version: Int!
}
type PaginatedPosts {
  posts: [Post!]
  nextToken: String
}
type Query {
  allPost(count: Int, nextToken: String): PaginatedPosts!
  getPost(id: ID!): Post
}
type Subscription {
  newPost: Post
  @aws_subscribe(mutations: ["addPost"])
}
```

Notice that the `@aws_subscribe` specifies which mutations trigger a subscription. You can add more mutations in this array to meet your application needs.

The subscription type `newPost` needs to be passed into an option (named `updateQuery`) of the React Apollo client to update your UI dynamically when a subscription is received. Ensure that this field name matches the subscription type in the following example code.

In your `App.js` file, edit the `AllPostsWithData` HOC to include `subscribeToNewPosts` in the `props` field, as follows:

```javascript
const AllPostsWithData = compose(
  graphql(AllPostsQuery, {
    options: {
      fetchPolicy: 'cache-and-network'
    },
    props: (props) => ({
      posts: props.data.allPost && props.data.allPost.posts,
      // START - NEW PROP :
      subscribeToNewPosts: params => {
```
Complex Objects

Many times you might want to create logical objects that have more complex data, such as images or videos, as part of their structure. For instance you might create a "Person" type with a profile picture or a "Post" type that has an associated image. With AWS AppSync, you can model these as GraphQL types, referred to as "Complex Objects". If any of your mutations have a variable with bucket, key, region, mimeType and localUri fields, the SDK will upload the file to Amazon S3 for you.

A complete working example of this feature can be found at https://github.com/aws-samples/aws-amplify-graphql.

First, note that to use Complex Objects you need AWS IAM credentials for reading and writing to S3. These can be separate from the other auth credentials you use in your AppSync client. Credentials for

```javascript
props.data.subscribeToMore({
  document: NewPostsSubscription,
  updateQuery: (prev, { subscriptionData: { data : { newPost } } }) => {
    ...prev,
    allPost: { posts: [newPost, ...prev.allPost.posts.filter(post => post.id !== newPost.id)], __typename: 'PaginatedPosts' }
  }
}),
// END - NEW PROP
});
...
//more code
```

import gql from 'graphql-tag';

export default gql`
  subscription NewPostSub {
    newPost {
      __typename
      id
      title
      author
      version
    }
  }
`;
Complex Objects are set using the `complexObjectsCredentials` parameter which can be used with AWS Amplify like so:

```javascript
const client = new AWSAppSyncClient({
    url: ENDPOINT,
    region: REGION,
    auth: { ... },
    complexObjectsCredentials: () => Auth.currentCredentials(),
});
```

Edit your schema to add the `S3Object` and `S3ObjectInput` types, as follows:

```graphql
enum S3Object
  input S3ObjectInput
  type Query
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID!): Post
  type Subscription
    newPost: Post
```
@aws_subscribe(mutations: ["addPost"],
}
)

Edit your ./src/Components/AddPost.jsx file, as follows:

```javascript
import React, { Component } from "react";
import { v4 as uuid } from 'uuid';
export default class AddPost extends Component {
    constructor(props) {
        super(props);
        this.state = this.getInitialState();
    }
    static defaultProps = {
        onAdd: () => null
    }
    getInitialState = () => ({
        id: '',
        title: '',
        author: '',
        file: null,
    });
    handleChange = (field, event) => {
        const { target: { value } } = event;
        this.setState({
            [field]: value
        });
    }
    handleAdd = () => {
        const { title, author, file: selectedFile } = this.state;
        let file;
        if (selectedFile) {
            const { name, type: mimeType } = selectedFile;
            const [, , , extension] = /(^.+)(\.(\w+))?$/.exec(name);
            const bucket = '[YOUR BUCKET]';
            const key = [uuid(), extension].filter(x => !!x).join('.');
            const region = '[YOUR REGION]';
            file = {
                bucket,
                key,
                region,
                mimeType,
                localUri: selectedFile,
            };  
        }
        this.setState(this.getInitialState(), () => {
            this.props.onAdd({
                title, author, content: 'hardcoded', file
            });
        });
    }
    handleCancel = () => {
        this.setState(this.getInitialState());
    }
```
Conflict Resolution

When clients make a mutation, either online or offline, they can send a version number with the payload (named `expectedVersion`) for AWS AppSync to check before writing to Amazon DynamoDB. A DynamoDB resolver mapping template can be configured to perform conflict resolution in the cloud, which you can learn about in Resolver Mapping Template Reference for DynamoDB (p. 209). If the service determines it needs to reject the mutation, data is sent to the client and you can optionally run an additional callback to perform client-side conflict resolution.

For example, suppose you had a mutation with DynamoDB set for checking the version, and the client sent `expectedVersion: 0`, as in this example:

```javascript
graphql(UpdatePostMutation, {
  props: (props) => ({
    onEdit: (post) => {
      props.mutate({
        variables: { ...post, expectedVersion: 0 },
        optimisticResponse: () => ({ updatePost: { ...post, __typename: 'Post',
          version: post.version + 1 } }),
      })
    },
  }),...more code
```

This would fail the version check because `0` would be lower than any of the current values. You can then define a custom callback conflict resolver. A custom conflict resolver will receive the following variables:

- **mutation**: GraphQL statement of a mutation
- **mutationName**: Optional if a name of a mutation is set on a GraphQL statement
- **variables**: Input parameters of the mutation

Now try running your app again by typing `yarn start`. Add a new post via the console, with a mutation on `addPost`. Your file should be uploaded to Amazon S3 before doing your mutation.
Building a React Native Client App

AWS AppSync integrates with the Apollo GraphQL client for building client applications. AWS provides Apollo plugins for offline support, authorization, and subscription handshaking. You can use the Apollo client directly, or you can use it with some of the client helpers provided in the AWS AppSync SDK. This tutorial shows you how to use AWS AppSync with React Apollo, which uses ReactJS constructs and patterns with GraphQL.

Before You Begin

This tutorial is set up for a sample API using the schema from the DynamoDB resolvers tutorial (p. 91). To follow along with the complete flow, you can optionally walk through that tutorial first. If you want to do more customization of GraphQL resolvers, such as those that use DynamoDB, see the Resolver Mapping Template Reference (p. 182). The application will use the following starting schema:

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

type Mutation {
  addPost(id: ID! author: String! title: String content: String! url: String!): Post!
}
```
Get the GraphQL API Endpoint

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

Get the GraphQL API Endpoint

After you create your GraphQL API, you’ll need to get the API endpoint (URL) so you can use it in your client application. There are two ways to get the API endpoint.

In the AWS AppSync console, choose Home, and then choose GraphQL URL to see the API endpoint.

Alternatively, you can get it by running the following CLI command:

```bash
aws appsync get-graphql-api --api-id #GRAPHQL_API_ID
```

Download a Client Application

To show you how to use AWS AppSync, we first review a React Native application (bootstrapped with create-react-native-app) with just a local array of data. Then we add AWS AppSync capabilities to it. To begin, download a sample application where we can add, update, and delete posts.

Understanding the React Native Sample App

The React Native sample app has three major files:

- `/src/App.js`: The main entry point of the application. Renders the main application shell with two components named AddPost and AllPosts, and has a local array of data named posts that is passed as a prop to the other components.

- `/src/Components/AddPost`: A React Native component that contains a form that enables a user to enter new information about a post, such as the author and title.

- `/src/Components/AllPosts`: A React Native component that lists all existing posts from the posts array that App.js created. It enables you to edit or delete existing posts.

Run your app as follows, and test it to be sure it works:
yarn && yarn start

## Import the AWS AppSync SDK into Your App

In this section, you'll add AWS AppSync to your existing React Native app.

For Android, you need to eject and add a permission to access network state. First, run the following:

```bash
yarn eject
```

After ejecting, edit `android/app/src/main/AndroidManifest.xml` with the following:

```xml
<manifest...
  android:versionCode="1"
  android:versionName="1.0">
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
</manifest>
```

Add the following dependencies to your application:

```bash
yarn add react-apollo graphql-tag aws-sdk
```

Next, add in the AWS AppSync SDK, including the React extensions:

```bash
yarn add aws-appsync
yarn add aws-appsync-react
```

From the AWS AppSync console, navigate to your GraphQL API landing page where the **API URL** is listed. At the bottom of the page, choose **Web**. Next, click the **Download** button and save the **AppSync.js** configuration file into `./`.

To interact with AWS AppSync, your client needs to define GraphQL queries and mutations. This is commonly done in separate files, as follows:

```bash
mkdir ./Queries
touch ./Queries/AllPostsQuery.js
touch ./Queries/DeletePostMutation.js
touch ./Queries/NewPostMutation.js
touch ./Queries/UpdatePostMutation.js
```

Edit and save `AllPostsQuery.js`:

```javascript
import gql from 'graphql-tag';

export default gql`
query AllPosts {
  allPost {
    __typename
    id
    title
    author
    version
  }
}
```

47
Import the AWS AppSync SDK into Your App

```javascript
import gql from 'graphql-tag';

export default gql`
mutation DeletePostMutation($id: ID!, $expectedVersion: Int!) {
  deletePost(id: $id, expectedVersion: $expectedVersion) {
    __typename
    id
    author
    title
    version
  }
}
```

**Edit and save DeletePostMutation.js:**

```javascript
import gql from 'graphql-tag';

export default gql`
mutation AddPostMutation($id: ID!, $author: String!, $title: String!) {
  addPost(
    id: $id
    author: $author
    title: $title
    content: ""
    url: ""
  ) {
    __typename
    id
    author
    title
    version
  }
}
```

**Edit and save NewPostMutation.js:**

```javascript
import gql from 'graphql-tag';

export default gql`
mutation UpdatePostMutation($id: ID!, $author: String, $title: String, $expectedVersion: Int!) {
  updatePost(
    id: $id
    author: $author
    title: $title
    expectedVersion: $expectedVersion
  ) {
    __typename
    id
    author
    title
    version
  }
}
```

**Edit and save UpdatePostMutation.js:**

```

Edit your App.js file, as follows:

```
import AWSAppSyncClient from "aws-appsync";
import { Rehydrated } from 'aws-appsync-react';
import { AUTH_TYPE } from "aws-appsync/lib/link/auth-link";
import { graphql, ApolloProvider, compose } from 'react-apollo';
import * as AWS from 'aws-sdk';
import AppSync from './AppSync.js';
import AllPostsQuery from './Queries/AllPostsQuery';
import NewPostMutation from './Queries/NewPostMutation';
import DeletePostMutation from './Queries/DeletePostMutation';
import UpdatePostMutation from './Queries/UpdatePostMutation';

After all the import statements, add the following code:

const client = new AWSAppSyncClient({
  url: AppSync.graphqlEndpoint,
  region: AppSync.region,
  auth: {
    type: AUTH_TYPE.API_KEY,
    apiKey: AppSync.apiKey,
    /*credentials: new AWS.Credentials({
      accessKeyId: AWS_ACCESS_KEY_ID,
      secretAccessKey: AWS_SECRET_ACCESS_KEY
    })*/
    //IAM Cognito Identity using AWS Amplify
    /*credentials: () => Auth.currentCredentials(),
     //Cognito User Pools using AWS Amplify
     //type: AUTH_TYPE.AMAZON_COGNITO_USER_POOLS,
     //jwtToken: async () => (await Auth.currentSession()).getIdToken().getJwtToken(),
  },
});

Note that you can switch the AUTH_TYPE value to use API keys, IAM (including short-term credentials from Amazon Cognito Federated Identities), or Amazon Cognito user pools. We recommend you use either IAM or Amazon Cognito user pools after onboarding with an API key. The previous code shows how to use the default configuration of AWS AppSync with an API key, referencing the AppSync.js file you downloaded. When you're ready to add other authorization methods to your application, you can use the AWS Amplify library to quickly add these capabilities to your application. The corresponding AWS Amplify methods for the AWS AppSync client constructor are included above, and an import of the library would look similar to the following:

import Amplify, { Auth } from 'aws-amplify';
import { withAuthenticator } from 'aws-amplify-react';
Amplify.configure(awsmobile);

//...code
const AppWithAuth = withAuthenticator(App, true);

For more information on using AWS Amplify, see the library documentation.

Replace the App component entirely, so it looks like this:

class App extends Component {
  state = { posts: [] };

  render() {

Delete the `posts` variable in your code, because the app state will be coming from AWS AppSync. Also, change the initial state of the App component to this:

```javascript
state = { posts: [] };  
```

At the bottom of your `App.js` file, define the following higher-order component (HOC):

```javascript
const AllPostsWithData = compose(
    graphql(AllPostsQuery, {
        options: {
            fetchPolicy: 'cache-and-network',
        },
        props: (props) => {
            posts: props.data.allPost && props.data.allPost.posts,
        }
    }),
    graphql(DeletePostMutation, {
        props: (props) => {
            onDelete: (post) => props.mutate({
                variables: { id: post.id, expectedVersion: post.version },
                optimisticResponse: () => ({ deletePost: {...post, __typename: 'Post'} })
            })
        }
    }),
    graphql(UpdatePostMutation, {
        props: (props) => {
            onEdit: (post) => {
                props.mutate({
                    variables: { ...post, expectedVersion: post.version },
                    optimisticResponse: () => ({ updatePost: {...post, __typename: 'Post', version: post.version + 1} })
                })
            }
        }
    }),
    refetchQueries: [{ query: AllPostsQuery }],
    update: (proxy, { data: { deletePost: { id } } }) => {
        const query = AllPostsQuery;
        const data = proxy.readQuery({ query });
        data.allPost.posts = data.allPost.posts.filter(post => post.id !== id);
        proxy.writeQuery({ query, data });
    },
    options: {
        refetchQueries: [{ query: AllPostsQuery }],
        update: (dataProxy, { data: { updatePost } }) => {
            const query = AllPostsQuery;
            const data = dataProxy.readQuery({ query });
            data.allPost.posts = data.allPost.posts.map(post => post.id !== updatePost.id ? post : { ...updatePost });
        }
    }
);
```
Test Your Application

yarn start

Offline Settings

There are important considerations that you'll need to account for if you want an optimistic UI for an application, where data can be manipulated when the device is in an offline state. Many of these settings are documented in the official Apollo documentation, however, we call out several of them here that you should configure.

First, note that the AWS AppSync client allows you to disable offline capabilities if you simply want to use GraphQL in an always-online scenario. To do this, you pass an additional option when instantiating your client, named disableOffline, as follows:

```javascript
const client = new AWSAppSyncClient({
    url: AppSync.graphqlEndpoint,
    region: AppSync.region,
    auth: {
```
• **fetchPolicy**: This option allows you to specify how a query interacts with the network versus local in-memory caching. AWS AppSync persists this cache to a platform-specific storage medium. If you are using the AWS AppSync client in offline scenarios (`disableOffline: false`), you **MUST** set this value to `cache-and-network`:

```javascript
options: {
  fetchPolicy: 'cache-and-network'
}
```

• **optimisticResponse**: This option allows you to pass a function or an object to a mutation for updating your UI before the server responds with the result. This is needed in offline scenarios (and for slower networks) to ensure that the UI is updated when the device has no connectivity. Optionally, you can use this if you have set `disableOffline: true`. For example, if you were adding a new object to a list, you might use the following:

```javascript
onAdd: post => props.mutate({
  variables: post,
  optimisticResponse: () => ({ addPost: { __typename: 'Post', ups: 1, downs: 1, content: '', url: '', version: 1, ...post } })
})
```

Normally, you use `optimisticResponse` in conjunction with the update option for React Apollo's component, which can trigger during an offline mutation. If you want the UI to update offline for a specific query, you need to specify that query as part of the `readQuery` and `writeQuery` options on the cache, as shown below:

```javascript
options: {
  refetchQueries: [{ query: AllPostsQuery }],
  update: (dataProxy, { data: { addPost } }) => {
    const query = AllPostsQuery;
    const data = dataProxy.readQuery({ query });
    data.allPost.posts.push(addPost);
    dataProxy.writeQuery({ query, data });
  }
}
```

When this happens, the AWS AppSync persistent store is automatically updated in response to the Apollo cache update. Upon network reconnection, it will synchronize with your GraphQL endpoint. You could also modify more than one query when offline, in which case you could run the above process multiple times in the same `update` block.

### Make Your Application Real Time

Edit your schema with the subscription type, as follows:

```javascript
schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}
```
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, expectedVersion: Int): Post!
  deletePost(id: ID!, expectedVersion: Int): Post
}

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

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

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

type Subscription {
  newPost: Post
  @aws_subscribe(mutations: ["addPost"])
}

Notice that the @aws_subscribe specifies which mutations trigger a subscription. You can add more mutations in this array to meet your application needs.

The subscription type newPost needs to be passed into an option named updateQuery of the React Apollo client to update your UI dynamically when a subscription is received. Ensure that this field name matches the subscription type in the following example code.

In your App.js file, edit the AllPostsWithData HOC to include subscribeToNewPost in the props field, as follows:

```javascript
const AllPostsWithData = compose(
  graphql(AllPostsQuery, {
    options: {
      fetchPolicy: 'cache-and-network'
    },
    props: (props) => ({
      posts: props.data.allPost && props.data.allPost.posts,
      // START - NEW PROP :
      subscribeToNewPosts: params => {
        props.data.subscribeToMore({
          document: NewPostsSubscription,
          updateQuery: (prev, { subscriptionData: { data: { newPost } } }) => {
            ...prev,
            allPost: { posts: [newPost, ...prev.allPost.posts.filter(post => post.id !== newPost.id)], __typename: 'PaginatedPosts' }
          }
        });
        // END - NEW PROP
      }
    }
  }
));
```
Conflict Resolution

When clients make a mutation, either online or offline, they can send a version number with the payload (named `expectedVersion`) for AWS AppSync to check before writing to Amazon DynamoDB. A DynamoDB resolver mapping template can be configured to perform conflict resolution in the cloud, which you can learn about in Resolver Mapping Template Reference for DynamoDB (p. 209). If the service determines it needs to reject the mutation, data is sent to the client and you can optionally run an additional callback to perform client-side conflict resolution.

For example, suppose you had a mutation with DynamoDB set for checking the version, and the client sent `expectedVersion: 0`, as in this example:

```
graphql(UpdatePostMutation, {
  props: (props) => ({
    onEdit: (post) => {
      props.mutate({
        ... // more code
      })
    },
    onDelete: () => null,
    onEdit: () => null,
    subscribeToNewPosts: () => null,
  })
})
```

Import the following statement at the top of your `App.js` file:

```
import NewPostsSubscription from './Queries/NewPostsSubscription';
```

Modify the `defaultProps` in the `AllPosts.js` component, as follows:

```
static defaultProps = {
  posts: [],
  onDelete: () => null,
  onEdit: () => null,
  subscribeToNewPosts: () => null,
}
```

Add the following lifecycle method to your `AllPosts` component in `AllPosts.js`:

```
componentWillMount(){
  this.props.subscribeToNewPosts();
}
```

Now try running your app again by typing `yarn start`. Add a new post via the console, with a mutation on `addPost`. You should see real-time data appear in your client application.
This would fail the version check because 0 would be lower than any of the current values. You can then define a custom callback conflict resolver. A custom conflict resolver will receive the following variables:

- mutation:GraphQL statement of a mutation
- mutationName: Optional if a name of a mutation is set on a GraphQL statement
- variables: Input parameters of the mutation
- data: Response from AWS AppSync of actual data in DynamoDB
- retries: Number of times a mutation has been retried

For example, you could have the following custom callback conflict resolver:

```javascript
const conflictResolver = ({ mutation, mutationName, variables, data, retries }) => {
  switch (mutationName) {
    case 'UpdatePostMutation':
      return {
        ...variables,
        expectedVersion: data.version,
      };
    default:
      return false;
  }
}
```

In the previous example, you can do a logical check on the mutationName and then rerun the mutation with the correct version that AWS AppSync returned.

**Note:** We recommend doing this only in rare cases. Usually, you should let the AWS AppSync service define conflict resolution, or race conditions can occur. If you don’t want to retry, simply return `DISCARD`.

Now, to use this callback, pass it into the AWS AppSync client instantiation:

```javascript
const client = new AWSAppSyncClient({
  url: awsconfig.ENDPOINT,
  region: awsconfig.REGION,
  auth: authInfo,
  conflictResolver,
});
```

---

**Building a JavaScript Client App**

AWS AppSync integrates with the Apollo GraphQL client for building client applications. AWS provides Apollo plugins for offline support, authorization, and subscription handshaking. This tutorial shows how you can use the AWS AppSync SDK with the Apollo client directly in a Node.js application. You can follow a similar process with popular JavaScript frameworks.

**Before You Begin**

This tutorial expects a GraphQL schema with the following structure:
schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}

type Mutation {
  deletePost(id: ID!): Post!
}

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
}

type Subscription {
  newPost: Post
  @aws_subscribe(mutations: ["addPost"])
}

This schema is from the DynamoDB resolvers tutorial (p. 91), with a subscription added. To follow the complete flow, you can optionally walk through that tutorial first. If you would like to do more customization of GraphQL resolvers, such as those that use DynamoDB, see the Resolver Mapping Template Reference (p. 182).

Get the GraphQL API Endpoint

After you create your GraphQL API, you'll need to get the API endpoint (URL) so you can use it in your client application. There are two ways to get the API endpoint.

In the AWS AppSync console, choose Home, and then choose GraphQL URL to see the API endpoint.

Alternatively, you can get it by running the following CLI command:

```bash
aws appsync get-graphql-api --api-id $GRAPHQL_API_ID
```

The following instructions show how you can use AWS_IAM for client authorization. In the console, select Settings on the left, and then click AWS_IAM.

Create a Client Application

Create a new project and initialize it with npm, accepting the defaults:

```bash
mkdir appsync && cd appsync
touch index.js aws-exports.js
```
AWS AppSync supports several authorization types, which you can learn more about in Security (p. 167). We recommend using short-term credentials from Amazon Cognito Federated Identities or Amazon Cognito user pools. For example purposes, we show how you can use IAM keys. Your `aws-exports` file should look like the following:

```javascript
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
var config = {
    AWS_ACCESS_KEY_ID: ' ',
    AWS_SECRET_ACCESS_KEY: ' ',
    HOST: 'URL.YOURREGION.amazonaws.com',
    REGION: 'YOURREGION',
    PATH: '/graphql',
    ENDPOINT: '',
};
config.ENDPOINT = "https://" + config.HOST + config.PATH;
exports.default = config;
```

Edit your `package.json` dependencies file and be sure it includes the following:

```javascript
"dependencies": {
    "apollo-cache-inmemory": "^1.1.0",
    "apollo-client": "^2.0.3",
    "apollo-link": "^1.0.3",
    "apollo-link-http": "^1.2.0",
    "aws-sdk": "^2.41.0",
    "aws-appsync": "^1.0.0",
    "es6-promise": "^4.1.1",
    "graphql": "^0.11.7",
    "graphql-tag": "^2.5.0",
    "isomorphic-fetch": "^2.2.1",
    "ws": "^3.3.1"
}
```

From a command line, run the following:

`npm install`

Now add the following to your `index.js` file:

```javascript
"use strict";
/**
 * This shows how to use standard Apollo client on Node.js
 */
global.WebSocket = require('ws');
global.window = global.window || {
    setTimeout: setTimeout,
    clearTimeout: clearTimeout,
    WebSocket: global.WebSocket,
    ArrayBuffer: global.ArrayBuffer,
    addEventListener: function () { },
    navigator: { onLine: true }
};
global.localStorage = {
    store: {},
    getItem: function (key) {
        return this.store[key]
    }
```
setItem: function (key, value) {
  this.store[key] = value
},
removeItem: function (key) {
  delete this.store[key]
};

require('es6-promise').polyfill();
require('isomorphic-fetch');

// Require exports file with endpoint and auth info
const aws_exports = require('./aws-exports').default;

// Require AppSync module
const AUTH_TYPE = require('aws-appsync/lib/link/auth-link').AUTH_TYPE;
const AWSAppSyncClient = require('aws-appsync').default;

const url = aws_exports.ENDPOINT;
const region = aws_exports.REGION;
const type = AUTH_TYPE.AWS_IAM;

// If you want to use API key-based auth
const apiKey = 'xxxxxxxxx';
// If you want to use a jwtToken from Amazon Cognito identity:
const jwtToken = 'xxxxxxxxx';

// If you want to use AWS...
const AWS = require('aws-sdk');
AWS.config.update({
  region: aws_exports.REGION,
  credentials: new AWS.Credentials({
    accessKeyId: aws_exports.AWS_ACCESS_KEY_ID,
    secretAccessKey: aws_exports.AWS_SECRET_ACCESS_KEY
  })
});
const credentials = AWS.config.credentials;

// Import gql helper and craft a GraphQL query
const gql = require('graphql-tag');
const query = gql(`
query AllPosts {
  allPost {
    __typename
    id
    title
    content
    author
    version
  }
}`);

// Set up a subscription query
const subquery = gql(`
subscription NewPostSub {
  newPost {
    __typename
    id
    title
    author
    version
  }
}`);

// Set up Apollo client
const client = new AWSAppSyncClient({
url: url,
    region: region,
    auth: {
        type: type,
        credentials: credentials,
    }
});

client.hydrated().then(function (client) {
    // Now run a query
    client.query({ query: query })
        .then(function logData(data) {
            console.log('results of query: ', data);
        })
        .catch(console.error);

    // Now subscribe to results
    const observable = client.subscribe({ query: subquery });

    const realtimeResults = function realtimeResults(data) {
        console.log('realtime data: ', data);
    }

    observable.subscribe({
        next: realtimeResults,
        complete: console.log,
        error: console.log,
    });
});

Notice that in the previous example, if you want to use an API key or Amazon Cognito user pools, you could update the AUTH_TYPE:

```
const type = AUTH_TYPE.API_KEY
const type = AUTH_TYPE.AMAZON_COGNITO_USER_POOLS
```

You would need to provide the key or JWT token, as appropriate.

## Building an iOS Client App

AWS AppSync integrates with the Apollo GraphQL client when building client applications. AWS provides plugins for offline support, authorization, and subscription handshaking to make this process easier. You can use the Apollo client directly, or with some client helpers provided in the AWS AppSync SDK when you get started.

This tutorial describes how to build an iOS application, including code generation for Swift types, by using AWS AppSync. For the latest AWS AppSync SDK documentation, see [https://awslabs.github.io/aws-mobile-appsync-sdk-ios/](https://awslabs.github.io/aws-mobile-appsync-sdk-ios/).

### Create an API

Before getting started, you'll need an API. See Designing a GraphQL API (p. 10) for details, and use the following schema to work with the examples below:

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

type Query {
  singlePost(id: ID!): Post
}
schema {
  query: Query
}

If you want to do more customization of GraphQL resolvers, see the Resolver Mapping Template Reference (p. 182).

You'll need the endpoint for your client, which you can get from the AWS AppSync console. Under Home, look for GraphQL URL. Or run the following CLI command:

```bash
aws appsync get-graphql-api --api-id $GRAPHQL_API_ID
```

## Download a Client Application

To show you how to use AWS AppSync, we first review an iOS application with just a local array of data. Then we add AWS AppSync capabilities to it. Go to the following URL to download a sample application, where we can add, update, and delete posts.

### Understanding the iOS Sample App

The iOS sample app has three major files:

1. `PostListViewController` The `PostListViewController` shows the list of posts available in the app. It uses a simple `UITableView` to list all the posts. You can Add, Update, or Delete posts from this `ViewController`.
2. `AddPostViewController` The `AddPostViewController` adds a new post into the list of existing posts. It gives a call to the delegate in `PostListViewController` to update the list of posts.
3. `UpdatePostViewController` The `UpdatePostViewController` updates an existing post from the list of posts. It gives a call to the delegate in `PostListViewController` to update the values of existing posts.

### Running the iOS Sample App

1. Open the `PostsApp.xcodeproj` file from the download bundle, which you downloaded previously.
2. Build the project (COMMAND+B) and ensure that it completes without error.
3. Run the project (COMMAND+R) and try the Add, Update, and Delete (swipe left) operations on the post list.

### Code Generation for the API

The AppSync iOS client generates a strongly typed API for your backend based on the GraphQL schema that’s defined. This helps you create native Swift request and response data objects, making it easy to understand the input and output requirements. For example, if your GraphQL schema defines a variable
as optional, the generated API has that variable as Swift `optional`. This enables the client to expect the value to be `nil` at certain times.

Set up the Code Generation for GraphQL Operations

To interact with AWS AppSync, your client needs to define GraphQL queries, mutations, and subscriptions. These are converted to strongly typed Swift objects.

You can do this by creating a `posts.graphql` file in a folder with a name like `GraphQLOperations`. Put the following operations in `posts.graphql`:

```graphql
query GetPost($id: ID!) {
  getPost(id:$id) {
    id
    title
    author
    content
    url
    version
  }
}

query AllPosts {
  listPosts {
    items {
      id
      title
      author
      content
      url
      version
      ups
      downs
    }
  }
}

mutation AddPost($input: CreatePostInput!) {
  createPost(input: $input) {
    id
    title
    author
    url
    content
  }
}

mutation UpdatePost($input: UpdatePostInput!) {
  updatePost(input: $input) {
    id
    author
    title
    content
    url
    version
  }
}

mutation DeletePost($input: DeletePostInput!) {
  deletePost(input: $input){
    id
    title
    author
    url
  }
}
```
Generate Swift API Code for Your API

Run the following commands to install `aws-appsync-codegen`. Then use the code generator to generate an API to access the AWS AppSync backend:

```bash
npm install -g aws-appsync-codegen
aws-appsync-codegen generate GraphQLOperations/*.graphql --schema GraphQLOperations/schema.json --output API.swift
```

**Note:** You’ll need the `schema.json` file, which is the GraphQL schema for your backend API. You can get it from the AWS AppSync console. Put this file in the `GraphQLOperations` folder with the `posts.graphql` file.

Add the generated `API.swift` file into your Xcode project.

Set up Dependency on the AWS AppSync SDK

1. Open a terminal, navigate to the location of the project that you downloaded, and then run the following:

   ```bash
   pod init
   ```

   This should create a `Podfile` in the root directory of the project. We’ll use this `Podfile` to declare dependency on the AWS AppSync SDK and other required components.

   Open the `Podfile` and add the following lines in the application target:

   ```ruby
   target 'PostsApp' do
     use_frameworks!
     pod 'AWSAppSync' ~> '2.6.15'
   end
   ```

   From the terminal, run the following command:

   ```bash
   pod install --repo-update
   ```

   This should create a file named `PostsApp.xcworkspace`. DO NOT open `*.xcodeproj` going forward. If it's open, you can close `PostsApp.xcworkspace`.

   Open the `PostsApp.xcworkspace` with Xcode. Build the project (COMMAND+B) and ensure that it completes without error.
In the app, edit the `Constants.swift` file, and update the GraphQL endpoint and your authentication mechanism.

```swift
let CognitoIdentityPoolId = "COGNITO_POOL_ID"
let CognitoIdentityRegion: AWSRegionType = .REGION
let AppSyncRegion: AWSRegionType = .REGION
let AppSyncEndpointURL: URL = URL(string: "https://APPSYNCURL/graphql")!
let database_name = "appsync-local-db"
```

**Convert the App to Use AWS AppSync for the Backend**

Add the `AppSyncClient` as an instance member of the `AppDelegate` class. This enables us to access the same client easily across the app, and update the `didFinishLaunching` method in `AppDelegate.swift` with the following code:

```swift
import UIKit
import AWSAppSync

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

  var window: UIWindow?
  var appSyncClient: AWSAppSyncClient?

  func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    // Set up Amazon Cognito credentials
    let credentialsProvider = AWSCognitoCredentialsProvider(regionType: CognitoIdentityRegion, identityPoolId: CognitoIdentityPoolId)
    // You can choose your database location, accessible by the SDK
    let databaseURL = URL(fileURLWithPath:NSTemporaryDirectory()).appendingPathComponent(database_name)
    do {
      // Initialize the AWS AppSync configuration
      let appSyncConfig = try AWSAppSyncClientConfiguration(url: AppSyncEndpointURL, serviceRegion: AppSyncRegion, credentialsProvider: credentialsProvider, databaseURL:databaseURL)
      // Initialize the AWS AppSync client
      appSyncClient = try AWSAppSyncClient(appSyncConfig: appSyncConfig)
      // Set id as the cache key for objects
      appSyncClient?.apolloClient?.cacheKeyForObject = { $0["id"] }
      catch {
        print("Error initializing appsync client. \($0)"")
        return true
      }
    }
    // ... other intercept methods
  }
}
```

**Update the AddPostViewController.swift file with the following code:**

```swift
import Foundation
```
import UIKit
import AWSAppSync

class AddPostViewController: UIViewController {
    @IBOutlet weak var authorInput: UITextField!
    @IBOutlet weak var titleInput: UITextField!
    @IBOutlet weak var contentInput: UITextField!
    @IBOutlet weak var urlInput: UITextField!
    var appSyncClient: AWSAppSyncClient?

    override func viewDidLoad() {
        super.viewDidLoad()
        appDelegate = UIApplication.shared.delegate as! AppDelegate
        appSyncClient = appDelegate.appSyncClient!
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated
    }

    @IBAction func addNewPost(_ sender: Any) {
        // Create a GraphQL mutation
        let uniqueId = UUID().uuidString
        let mutationInput = CreatePostInput(id: uniqueId,
                                             author: authorInput.text!,
                                             title: titleInput.text,
                                             content: contentInput.text,
                                             url: urlInput.text,
                                             version: 1)

        let mutation = AddPostMutation(input: mutationInput)
        appSyncClient?.perform(mutation: mutation, optimisticUpdate: { (transaction) in
            do {
                // Update our normalized local store immediately for a responsive UI
                try transaction?.update(query: AllPostsQuery()) { (data: inout
                    AllPostsQuery.Data) in
                    data.listPosts?.items?.append(AllPostsQuery.Data.ListPost.Item.init(id:
                        uniqueId, title: mutationInput.title!, author: mutationInput.author, content:
                        mutationInput.content!, version: 0))
                }
            } catch {
                print("Error updating the cache with optimistic response.")
            }
        }) {(result, error) in
            if let error = error as? AWSAppSyncClientError {
                print("Error occurred: \
                    \n                print("Error occurred: \
                    return
            }
            self.dismiss(animated: true, completion: nil)
        }

        self.dismiss(animated: true, completion: nil)
    }

    @IBAction func onCancel(_ sender: Any) {
        self.dismiss(animated: true, completion: nil)
    }
}

Update the UpdatePostViewController.swift file with the following code:

import Foundation

...
import UIKit
import AWSAppSync

class UpdatePostViewController: UIViewController {

  var updatePostMutation: UpdatePostMutation?
  var updatePostInput: UpdatePostInput?

  @IBOutlet weak var authorInput: UITextField!
  @IBOutlet weak var titleInput: UITextField!
  @IBOutlet weak var contentInput: UITextField!
  @IBOutlet weak var urlInput: UITextField!

  var appSyncClient: AWSAppSyncClient?

  override func viewDidLoad() {
    super.viewDidLoad()
    authorInput.text = updatePostInput?.author!
    titleInput.text = updatePostInput?.title!
    contentInput.text = updatePostInput?.content!
    urlInput.text = updatePostInput?.url!

    let appDelegate = UIApplication.shared.delegate as! AppDelegate
    appSyncClient = appDelegate.appSyncClient!
  }

  @IBAction func updatePost(_ sender: Any) {
    updatePostInput?.author = authorInput.text!
    updatePostInput?.title = titleInput.text
    updatePostInput?.content = contentInput.text
    updatePostInput?.url = urlInput.text

    updatePostMutation = UpdatePostMutation(input: updatePostInput!)

    appSyncClient?.perform(mutation: updatePostMutation!) { (result, error) in
      if let error = error as? AWSAppSyncClientError {
        print("Error occurred while making request: \(error.localizedDescription )")
        return
      }
      if let resultError = result?.errors {
        print("Error saving the item on server: \(resultError)")
        return
      }
      self.dismiss(animated: true, completion: nil)
    }
  }

  @IBAction func onCancel(_ sender: Any) {
    self.dismiss(animated: true, completion: nil)
  }
}

Update the PostListViewController.swift file with the following code:

import UIKit
import AWSAppSync

class PostCell: UITableViewCell {
  @IBOutlet weak var authorLabel: UILabel!
  @IBOutlet weak var titleLabel: UILabel!
  @IBOutlet weak var contentLabel: UILabel!

  func updateValues(author: String, title: String?, content: String?) {
    authorLabel.text = author
    titleLabel.text = title
    contentLabel.text = content
  }
}

Update the PostListViewController.swift file with the following code:
AWS AppSync AWS AppSync Developer Guide
Convert the App to Use AWS AppSync for the Backend
class PostListViewController: UIViewController, UITableViewDelegate, UITableViewDataSource
{
var appSyncClient: AWSAppSyncClient?
@IBOutlet weak var tableView: UITableView!
var postList: [AllPostsQuery.Data.ListPost.Item?]? = [] {
didSet {
tableView.reloadData()
}
}
func loadAllPosts() {
appSyncClient?.fetch(query: AllPostsQuery(), cachePolicy: .returnCacheDataAndFetch)
{ (result, error) in
if error != nil {
print(error?.localizedDescription ?? "")
return
}
self.postList = result?.data?.listPosts?.items
}
}
func loadAllPostsFromCache() {
appSyncClient?.fetch(query: AllPostsQuery(), cachePolicy: .returnCacheDataDontFetch)
{ (result, error) in
if error != nil {
print(error?.localizedDescription ?? "")
return
}
self.postList = result?.data?.listPosts?.items
}
}
override func viewWillAppear(_ animated: Bool) {
super.viewWillAppear(animated)
loadAllPostsFromCache()
}
override func viewDidLoad() {
super.viewDidLoad()
// Do any additional setup after loading the view, typically from a nib
self.automaticallyAdjustsScrollViewInsets = false
let appDelegate = UIApplication.shared.delegate as! AppDelegate
appSyncClient = appDelegate.appSyncClient
loadAllPosts()
self.tableView.dataSource = self
self.tableView.delegate = self
navigationItem.rightBarButtonItem = UIBarButtonItem(title: "Add", style: .plain,
target: self, action: #selector(addTapped))
}
@objc func addTapped() {
let storyboard = UIStoryboard(name: "Main", bundle: nil)
let controller = storyboard.instantiateViewController(withIdentifier:
"NewPostViewController") as! AddPostViewController
self.present(controller, animated: true, completion: nil)
}
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

66


return postList?.count ?? 0
}

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "PostCell", for: indexPath) as! PostCell
    let post = postList![indexPath.row]!
    cell.updateValues(author: post.author, title: post.title, content: post.content)
    return cell
}

func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool {
    return true
}

func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    if (editingStyle == UITableViewCellEditingStyle.delete) {
        let id = postList![indexPath.row]!.id
        let deletePostMutation = DeletePostMutation(input: DeletePostInput(id: id!))
        appSyncClient?.perform(mutation: deletePostMutation) { result, err in
            self.postList?.remove(at: indexPath.row)
            self.tableView.reloadData()
        }
    }
}

func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    let post = postList![indexPath.row]!
    let storyboard = UIStoryboard(name: "Main", bundle: nil)
    let controller = storyboard.instantiateViewController(withIdentifier: "UpdatePostViewController") as! UpdatePostViewController
    controller.updatePostInput = UpdatePostInput(id: post.id, author: post.author, title: post.title, content: post.content, url: post.url, ups: 1, downs: 0, version: post.version)
    self.present(controller, animated: true, completion: nil)
}

Make Your App Real Time

AWS AppSync and GraphQL use the concept of subscriptions to deliver real-time updates of data to the application. We have defined subscriptions on the events of NewPost, UpdatePost, and DeletePost. This means we’ll get a real-time notification if app data is changed from another device, and we can update our application UI based on the updates.

Add a real-time subscription to receive events on a new post that is added by anyone. In the PostListViewController.swift file, add the following function: .. code-block:: swift

```swift
func startNewPostSubscription() {
    let subscription = OnCreatePostSubscription() do {
        _ = try appSyncClient?.subscribe(subscription: subscription, resultHandler: { (result, transaction, error) in
            if let result = result {
                // Store a reference to the new object let newPost = result.data!.onCreatePost!
                // Create a new object for the desired query, where the new object content should reside let postToAdd = AllPostsQuery.Data.ListPost.Item(id: newPost.id, title: newPost.title, author: newPost.author, content: newPost.content, version: 1)
                self.postList?.add(postToAdd)
            }
        })
    }
}
```
do {
    // Update the local store with the newly received data
    try transaction?.update(query: AllPostsQuery()) { (data: inout AllPostsQuery.Data) in
        data.listPosts?.items?.append(postToAdd)
    } self.loadAllPostsFromCache()
    } catch {
        print("Error updating store")
    }
} else if let error = error {
    print(error.localizedDescription)
}

Next, call the method we created from the viewDidLoad method of PostListViewController. This updates the list of posts every time a new post is added from any client.

### Complex Objects

Many times you might want to create logical objects that have more complex data, such as images or videos, as part of their structure. For example, you might create a "Person" type with a profile picture or a "Post" type that has an associated image. You can use AWS AppSync to model these as GraphQL types. If any of your mutations have a variable with bucket, key, region, mimeType, and localUri fields, the SDK will upload the file to Amazon S3 for you.

Update your schema as follows to add the S3Object and S3ObjectInput types for the file, and a new mutation named CreatePostWithFileInputMutation:

```graphql
input CreatePostInput {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
}

input CreatePostWithFileInput {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
}
```
Complex Objects

downs: Int
file: S3ObjectInput!
version: Int!
}

input DeletePostInput {
id: ID!
}

type Mutation {
createPost(input: CreatePostInput!): Post
createPostWithFile(input: CreatePostWithFileInput!): Post
updatePost(input: UpdatePostInput!): Post
deletePost(input: DeletePostInput!): Post
}

type Post {
id: ID!
author: String!
title: String
content: String
url: String
ups: Int
downs: Int
file: S3Object
version: Int!
}

type PostConnection {
items: [Post]
nextToken: String
}

type Query {
singlePost(id: ID!): Post
getPost(id: ID!): Post
listPosts(first: Int, after: String): PostConnection
}

type S3Object {
bucket: String!
key: String!
region: String!
}

input S3ObjectInput {
bucket: String!
key: String!
region: String!
localUri: String!
mimeType: String!
}

type Subscription {
onCreatePost(
  id: ID,
  author: String,
title: String,
  content: String,
  url: String
): Post
  @aws_subscribe(mutations: ["createPost"]) 
onUpdatePost(
  id: ID,
  author: String,
title: String,
  content: String,
  url: String
): Post
  @aws_subscribe(mutations: ["updatePost"]) 
}
content: String,
url: String
): Post
@aws_subscribe(mutations: ["updatePost")
onDeletePost(
  id: ID,
  author: String,
  title: String,
  content: String,
  url: String
): Post
@aws_subscribe(mutations: ["deletePost")
}

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
}

Note: If you’re using the sample schema specified at the start of this documentation, you can replace your schema with schema above.

Next, you need to add a resolver for createPostWithFile mutation. You can do that from the AWS AppSync console by selecting PostsTable as the datasource and the following mapping templates:

Request Mapping Template

{
  "version": "2017-02-28",
  "operation": "PutItem",
  "key": {
    "id": $util.dynamodb.toDynamoDBJson($ctx.args.input.id),
  },
  #set( $attribs = $util.dynamodb.toMapValues($ctx.args.input) )
  #if($util.isNull($ctx.args.input.file.version))
    #set( $attribs.file = $util.dynamodb.toS3Object($ctx.args.input.file.key,
      $ctx.args.input.file.bucket, $ctx.args.input.file.region))
  #else
    #set( $attribs.file = $util.dynamodb.toS3Object($ctx.args.input.file.key,
      $ctx.args.input.file.bucket, $ctx.args.input.file.region, $ctx.args.input.file.version))
  #end
  "attributeValues": $util.toJson($attribs),
  "condition": {
    "expression": "attribute_not_exists(#id)",
    "expressionNames": {
      "#id": "id",
    },
  },
}

Response Mapping Template
Once you have a resolver for the mutation, to ensure that our S3 Complex Object details are fetched correctly during any query operation, add a resolver for the `file` field of `Post`. You can do that from the AWS AppSync console by using the following mapping templates:

**Request Mapping Template**

```
{
  "version" : "2017-02-28",
  "operation" : "Query",
  "query" : {
    ## Provide a query expression. **
    "expression": "id = :id",
    "expressionValues" : {
      ":id" : {
        "S" : "${ctx.args.id}"
      }
    }
  }
}
```

**Response Mapping Template**

```
#util.toJson(#util.dynamodb.fromS3ObjectJson(#context.source.file))
```

The AWS AppSync SDK doesn't take a direct dependency on the AWS SDK for iOS for S3, but takes in `AWSS3TransferUtility` and `AWSS3PresignedURLClient` clients as part of `AWSAppSyncClientConfiguration`. The code generator used above for generating the API generates the S3 wrappers required to use the previous clients in the client code. To generate the wrappers, pass the `--add-s3-wrapper` flag while running the code generator tool. You also need to take a dependency on the `AWSS3` SDK. You can do that by updating your `Podfile` to the following:

```
target 'PostsApp' do
  use_frameworks!
  pod 'AWSAppSync' ~> '2.6.15'
  pod 'AWSS3' ~> '2.6.13'
end
```

Then run `pod install` to fetch the new dependency.

Download the updated `schema.json` from the `<AppSync console>` and put it in the `GraphQLOperations` folder in the root of the app.

Next, you have to add the new mutation, which is used to perform S3 uploads as part of mutation. Add the following mutation operation in your `posts.graphql` file:

```
mutation AddPostWithFile($input: CreatePostWithFileInput!) {
  createPostWithFile(input: $input) {
    id
title
author
url
content
ups
downs
version
```
Conflict Resolution

After adding the new mutation in our operations file, we run the code generator again with the new schema to generate mutations that support file uploads. This time, we also pass the `-add-s3-wrapper` flag, as follows:

```
aws-appsync-codegen generate GraphQLOperations/posts.graphql --schema GraphQLOperations/schema.json --output API.swift --add-s3-wrapper
```

Update the `AWSAppSyncClientConfiguration` object to provide the `AWSS3TransferUtility` client for managing the uploads and downloads:

```swift
let appSyncConfig = try AWSAppSyncClientConfiguration(url: AppSyncEndpointURL, 
serviceRegion: AppSyncRegion, 
credentialsProvider: 
credentialsProvider, 
databaseURL:databaseURL, 
s3ObjectManager: AWSS3TransferUtility.default())
```

The mutation operation doesn't require any specific changes in method signature. It requires only an `S3ObjectInput` with `bucket`, `key`, `region`, `localUri`, and `mimeType`. Now when you do a mutation, it automatically uploads the specified file to S3 using the `AWSS3TransferUtility` client internally.

### Conflict Resolution

When clients make a mutation, either online or offline, they can send a version number with the payload (named `expectedVersion`) for AWS AppSync to check before writing to Amazon DynamoDB. You can configure a DynamoDB resolver mapping template to perform conflict resolution in the cloud. See `Resolver Mapping Template Reference for DynamoDB` (p. 209). If the service determines it needs to reject the mutation, data is sent to the client. You can optionally run an additional callback to perform client-side conflict resolution.

For example, suppose you had a mutation with DynamoDB set for checking the version, and the client sent `expectedVersion: 0`, as in this example:

```swift
@IBAction func updatePost(_ sender: Any) {
    let updatePostMutation = UpdatePostMutation(id: "1", 
        author: "Mr. Abc", 
        content: "UpdatedContent", 
        expectedVersion: 0)

    appSyncClient?.perform(mutation: updatePostMutation) { (result, error) in 
        if let error = error as? AWSAppSyncClientError {
            print("Error occurred while making request: \(error.localizedDescription )")
            return
        }
        if let resultError = result?.errors {
```
Building an Android Client App

Create an API

Before getting started, you will need an API. See Designing a GraphQL API for details. For this tutorial use the following schema with the create resources flow (p. 27).

type Post {
  id: ID!
  author: String!
  title: String
  content: String
  url: String
  ups: Int
  downs: Int
  version: Int!
}
type Query {
  singlePost(id: ID!): Post
}
schema {
  query: Query
}
Note: After creating your API, use this schema in the console, then select create resources, select Post as your type and then press Create at the bottom.

If you want to do more customization of GraphQL resolvers, see the Resolver Mapping Template Reference (p. 182).

Download a Client Application

This tutorial will use the AWS AppSync Posts schema starter kit.

If you wish to see the entire app without following the steps, here is the whole sample.

Gradle Setup

Project's build.gradle

In the project's build.gradle file, add the following dependency in the build script:

```groovy
classpath 'com.amazonaws:aws-android-sdk-appsync-gradle-plugin:2.6.16'
```

Sample Project's build.gradle

```groovy
// Top-level build file where you can add configuration options common to all sub-projects/modules.
buildscript {
    // ..other code..
    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.1'
        classpath 'com.amazonaws:aws-android-sdk-appsync-gradle-plugin:2.6.16'
        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

App's build.gradle

In the app's build.gradle file, add the following plugin:

```groovy
apply plugin: 'com.amazonaws.appsync'
```

Add the following dependency:

```groovy
compile 'com.amazonaws:aws-android-sdk-appsync:2.6.16'
```

Sample App's build.gradle

```groovy
apply plugin: 'com.amazonaws.appsync'
apply plugin: 'com.android.application'
android {
    // Typical items
}
dependencies {
    // Typical dependencies
    compile 'com.amazonaws:aws-android-sdk-appsync:2.6.16'
}
App's AndroidManifest.xml

Add the permissions to access the network state to determine if the device is offline. You will need some of the other permissions for GraphQL subscriptions.

<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>

Code Generation for GraphQL Operations

To interact with AWS AppSync, your client needs to define GraphQL queries and mutations.

Create a file named ./app/src/main/graphql/com/amazonaws/demo/posts/posts.graphql:

```graphql
query GetPost($id:ID!) {
  getPost(id:$id) {
    id
    title
    author
    content
    url
    version
  }
}

query AllPosts {
  listPosts {
    items {
      id
      title
      author
      content
      url
      version
      ups
      downs
    }
  }
}

mutation AddPost($input: CreatePostInput!) {
  createPost(input: $input) {
    id
    title
    author
    url
    content
  }
}

mutation UpdatePost($input: UpdatePostInput!) {
  updatePost(input: $input) {
    id
    author
    title
  }
}
```
mutation DeletePost($input: DeletePostInput!) {
  deletePost(input: $input){
    id
    title
    author
    url
    content
  }
}

Next, fetch the schema.json file from the AWS AppSync console and place it alongside the posts.graphql file (/app/src/main/graphql/com/amazonaws/demo/posts/schema.json). You can find this in the console by clicking on your API name in the left-hand navigation, scrolling to the bottom, selecting Android, clicking the Export schema dropdown, and selecting schema.json.

Now build the project. The generated source files will be available to use within the app. They will not show up in your source directory, but are added in the build path.

Call the Service

In this section, we create a client and call the service.

Set up Constants.java

You will need the endpoint for your client, which you can get from the AWS AppSync console in the same place the you retrieved the schema.json in the previous step. Alternatively, you can run the following CLI command:

```bash
aws appsync get-graphql-api --api-id $GRAPHQL_API_ID
```

Edit Constants.java with your information. Not all of the constants will be used, depending on the authorization path chosen in the next steps. For example, if you’re using API key-based authorization, the identity pool ID and region can be ignored.

```java
public static final String APPSYNC_API_URL = "https://API-URL/graphql"; // TODO: Update the endpoint URL as specified on AppSync console

// API Key Authorization
public static final String APPSYNC_API_KEY = "API-KEY"; // TODO: Copy the API Key specified on the AppSync Console

// IAM based Authorization (Cognito Identity)
public static final String COGNITO_IDENTITY = "POOL-ID"; // TODO: Update the Cognito Identity Pool ID
public static final Regions COGNITO_REGION = Regions.US_WEST_2; // TODO: Update the region to match the Cognito Identity Pool region

// Cognito User Pools Authorization
public static final String USER_POOLS_POOL_ID = "";
public static final String USER_POOLS_CLIENT_ID = "";
public static final String USER_POOLS_CLIENT_SECRET = "";
public static final Regions USER_POOLS_REGION = Regions.US_WEST_2; // TODO: Update the region to match the Cognito User Pools region
```
For Cognito Identity, you will need to grant the IAM role that your client assumes for permissions to query, mutate, or subscribe to data with AWS AppSync. This will be on either the Unauthenticated or Authenticated role that you configure for your Cognito Identity Pool. Please note that the region of the Cognito Identity Pool can be different than the region of your AppSync API and your constants file should configure this appropriately. You can add this as an inline policy for the appropriate role as outlined here (p. 167).

For Cognito User Pools, you can configure default permissions on the schema and then restrict at the GraphQL type level using logical groups. Please note that the region of the Cognito User Pool can be different than the region of your AppSync API and your constants file should configure this appropriately. You can find more information on this here (p. 167).

To configure any of these settings (including API Keys) navigate to the Settings page of the AppSync console and select the Authorization type as appropriate.

Create the client

When making calls to AWS AppSync, there are 3 ways to authenticate those calls. The API Key Authorization is the most simple to on-board with, but it is recommended to use either IAM or Amazon Cognito User Pools after on-boarding with an API key.

API Key Authorization

For authorization using the API key, update the ClientFactory.getInstance(Context) method with the following code.

```java
public class ClientFactory {
    // ...other code...
    private static volatile AWSAppSyncClient client;

    public static AWSAppSyncClient getInstance(Context context) {
        if (client == null) {
            client = AWSAppSyncClient.builder()
                .context(context)
                .apiKey(new BasicAPIKeyAuthProvider(Constants.APPSYNC_API_KEY)) // API Key based authorization
                .region(Constants.APPSYNC_REGION)
                .serverUrl(Constants.APPSYNC_API_URL)
                .build();
        }
        return client;
    }
}
```

IAM-Based Authorization (Amazon Cognito Identity)

For IAM-based authorization, update the ClientFactory.getInstance(Context) method with the following code.

```java
public class ClientFactory {
    // ...other code...
    private static volatile AWSAppSyncClient client;

    public static AWSAppSyncClient getInstance(Context context) {
        if (client == null) {
            client = AWSAppSyncClient.builder()
                .context(context)
                .credentialsProvider(getCredentialsProvider(context)) // For use with IAM/Cognito authorization
                .build();
        }
        return client;
    }
}
```
Call the Service

```java
private static final AWSCredentialsProvider getCredentialsProvider(final Context context) {
    final CognitoCachingCredentialsProvider credentialsProvider = new CognitoCachingCredentialsProvider(
        context,
        Constants.COGNITO_IDENTITY,
        Constants.COGNITO_REGION
    );
    return credentialsProvider;
}
```

Cognito User Pools Authorization

For Cognito User Pools Authorization, update the `ClientFactory.getInstance(Context)` method with the following code.

```java
import com.amazonaws.mobileconnectors.cognitoidentityprovider.CognitoUserPool;

public class ClientFactory {
    // ...other code...
    private static volatile AWSAppSyncClient client;
    public static AWSAppSyncClient getInstance(Context context) {
        if (client == null) {
            client = AWSAppSyncClient.builder()
                .context(context)
                .cognitoUserPoolsAuthProvider(getUserPools(context)) // For use with User Pools authorization
                .region(Constants.APPSYNC_REGION)
                .serverUrl(Constants.APPSYNC_API_URL)
                .build();
        }
        return client;
    }

    private static CognitoUserPoolsAuthProvider getUserPools(final Context context) {
        final CognitoUserPoolsAuthProvider provider = new BasicCognitoUserPoolsAuthProvider(new CognitoUserPool(
            context,
            Constants.USER_POOLS_POOL_ID,
            Constants.USER_POOLS_CLIENT_ID,
            Constants.USER_POOLS_CLIENT_SECRET,
            Constants.USER_POOLS_REGION));
        return provider;
    }
}
```

Query the Posts

Add the `PostsActivity.queryData()` method and create a callback to receive the data, when available.

```java
public class PostsActivity extends AppCompatActivity {
    // ...other code...
}
```
Call the Service

```java
public void queryData() {
    if (mAWSAppSyncClient == null) {
        mAWSAppSyncClient = ClientFactory.getInstance(this);
    }
    mAWSAppSyncClient.query(AllPostsQuery.builder().build())
        .responseFetcher(AppSyncResponseFetchers.CACHE_AND_NETWORK)
        .enqueue(postsCallback);
}

private GraphQLCall.Callback<AllPostsQuery.Data> postsCallback = new
    GraphQLCall.Callback<AllPostsQuery.Data>() {
        @Override
        public void onResponse(@Nonnull final Response<AllPostsQuery.Data> response) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    PostsActivity.this.mAdapter.setPosts(response.data().listPosts().items());
                    PostsActivity.this.mSwipeRefreshLayout.setRefreshing(false);
                    PostsActivity.this.mAdapter.notifyDataSetChanged();
                }
            });
        }
    }

    @Override
    public void onFailure(@Nonnull ApolloException e) {
        Log.e(TAG, "Failed to perform AllPostsQuery", e);
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                PostsActivity.this.mSwipeRefreshLayout.setRefreshing(false);
            }
        });
    }
}
```

Mutate the Posts (Add a Post)

Add the `AddPostActivity.save()` method and create a callback to receive confirmation.

```java
public class AddPostActivity extends AppCompatActivity {

    // ...other code...

    private void save() {
        final String title = ((EditText) findViewById(R.id.updateTitle)).getText().toString();
        final String author = ((EditText) findViewById(R.id.updateAuthor)).getText().toString();
        final String url = ((EditText) findViewById(R.id.updateUrl)).getText().toString();
        final String content = ((EditText) findViewById(R.id.updateContent)).getText().toString();
        final String id = UUID.randomUUID().toString();
        AddPostMutation addPostMutation = AddPostMutation.builder()
            .input(CreatePostInput.builder()
                .id(id)
                .title(title)
                .author(author)
                .url(url)
            )
            .build();
        
```
Mutate the Posts (Update a Post)

Add the UpdatePostActivity.save() method and create a callback to receive confirmation.

```java
@override
public void save() {
    final String title = ((EditText) findViewById(R.id.updateTitle)).getText().toString();
    final String author = ((EditText) findViewById(R.id.updateAuthor)).getText().toString();
    final String url = ((EditText) findViewById(R.id.updateUrl)).getText().toString();
    final String content = ((EditText) findViewById(R.id.updateContent)).getText().toString();

    UpdatePostMutation updatePostMutation = UpdatePostMutation.builder()
        .input(UpdatePostInput.builder()
            .id(sPost.id())
            .title(title)
            .author(author)
            .url(url)
            .content(content)
            .ups(sPost.ups())
            .downs(sPost.downs())
            .build());

    ClientFactory.getInstance(this).mutate(updatePostMutation).enqueue(postsCallback);
}
```

```java
private GraphQLCall.Callback<UpdatePostMutation.Data> postsCallback = new
    GraphQLCall.Callback<UpdatePostMutation.Data>() {
        @Override
        public void onResponse(@Nonnull final Response<UpdatePostMutation.Data> response) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Toast.makeText(AddPostActivity.this, "Updated post",
                        Toast.LENGTH_SHORT).show();
                    AddPostActivity.this.finish();
                }
            });
        }

        @Override
        public void onFailure(@Nonnull final ApolloException e) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Log.e("","Failed to perform UpdatePostMutation", e);
                    Toast.makeText(AddPostActivity.this, "Failed to update post",
                        Toast.LENGTH_SHORT).show();
                    AddPostActivity.this.finish();
                }
            });
        }
    }
```
Optimistic Updates

Update a Post

This section makes changes to the `UpdatePostActivity.save()` method that was created in the previous step.

For optimistic updates, create the data expected to be returned after the mutation. The optimistic updates are written to the persistent SQL store.

Note that the `UpdatePostMutation` class is the same class used to create the expected object as well as make the mutation request.

```java
UpdatePostMutation.Data expected = new UpdatePostMutation.Data(new UpdatePostMutation.PutPost(  "Post",  sPost.id(),  author,  title,  content,  url,  sPost.version() + 1));
```

The mutation call will now become the following, which takes in the expected data:
UpdatePostMutation.Data expected = new UpdatePostMutation.Data(new
  UpdatePostMutation.PutPost(
    "Post",
    sPost.id(),
    author,
    title,
    content,
    url,
    sPost.version() + 1)
);  
UpdatePostMutation updatePostMutation = UpdatePostMutation.builder()
  .input(UpdatePostInput.builder()
    .id(sPost.id())
    .title(title)
    .author(author)
    .url(url)
    .content(content)
    .ups(sPost.ups())
    .downs(sPost.downs())
    .version(sPost.version() + 1)
    .build()
  )
  .build();

// Make mutation call
ClientFactory.getInstance(this).mutate(updatePostMutation,
  expected).enqueue(postsCallback);

Add a Post

This section makes changes to the AddPostActivity.save() method that was created in the previous step.

AddPostMutation.Data expected = new AddPostMutation.Data(new AddPostMutation.CreatePost(
  "Post",
  id,
  title,
  author,
  url,
  content
 ));
AddPostMutation addPostMutation = AddPostMutation.builder()
  .input(CreatePostInput.builder()
    .id(id)
    .title(title)
    .author(author)
    .url(url)
    .content(content)
    .ups(0)
    .downs(0)
    .version(1)
    .build()
  )
  .build();
ClientFactory.getInstance(this).mutate(addPostMutation, expected).enqueue(postsCallback);

Offline Mutations

Offline mutations currently work out of the box and are available in memory, as well as through app restarts.
The callback for onResponse is received when the network is available, and the request goes through if the app was not killed.

For mutations which are performed after an app restart, the PersistentMutationsCallback object will be called.

The PersistentMutationsCallback will have information about the mutation type and identifier. It can be specified while initializing the client.

```java
AWSAppSyncClient client = AWSAppSyncClient.builder()
    .context(context)
    .apiKey(new BasicAPIKeyAuthProvider(Constants.APPSYNC_API_KEY)) // API Key based auth
    .region(Constants.APPSYNC_REGION)
    .serverUrl(Constants.APPSYNC_API_URL)
    .persistentMutationsCallback(new PersistentMutationsCallback() {
        @Override
        public void onResponse(PersistentMutationsResponse response) {
            if (response.getMutationClassName().equals("AddPostMutation")) {
                // perform action here add post mutation
            }
        }

        @Override
        public void onFailure(PersistentMutationsError error) {
            // handle error feedback here
        }
    })
    .build();
```

**Subscriptions**

Subscriptions allow information to be pushed to the client when certain events happen such as a new post being added.

Append to the file named `./app/src/main/graphql/com/amazonaws/demo/posts/posts.graphql`:

```graphql
subscription OnCreatePost {
  onCreatePost {
    id
    author
    title
    content
    url
  }
}
```

**App's build.gradle**

In the app's build.gradle file, add the following dependency:

```groovy
compile 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.2.0'
compile 'org.eclipse.paho:org.eclipse.paho.android.service:1.1.1'
```

**Sample App's build.gradle**

```groovy
android {
    // Typical items
```
dependencies {
  // Typical dependencies
  compile 'com.amazonaws:aws-android-sdk-appsync:2.6.16'
  compile 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.2.0'
  compile 'org.eclipse.paho:org.eclipse.paho.android.service:1.1.1'
}

### AndroidManifest.xml

Add the following service into the application:

```xml
<service android:name="org.eclipse.paho.android.service.MqttService" />
```

#### Sample AndroidManifest

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.amazonaws.postsapp">
  <!--other code-->
  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <service android:name="org.eclipse.paho.android.service.MqttService" />
    <!--other code-->
  </application>
</manifest>
```

### Client call

This section makes changes to the `PostsActivity.onCreate()` and `PostsActivity.onPause()` methods.

```java
public class PostsActivity extends AppCompatActivity {
    // .. other code ..
    private AppSyncSubscriptionCall subscriptionWatcher;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // .. other code ..
        subscribe();
    }

    private void subscribe() {
        OnCreatePostSubscription subscription = OnCreatePostSubscription.builder().build();
        subscriptionWatcher = ClientFactory.getInstance(this).subscribe(subscription);
        subscriptionWatcher.execute(subCallback);
    }
```
Complex Objects

Logical objects can be created that have more complex data, such as images or videos, as part of their structure. For instance you might create a "Person" type with a profile picture or a "Post" type that has an associated image. With AWS AppSync, you can model these as GraphQL types.

Schema setup

If any mutations have an input type S3ObjectInput with fields bucket, key, region, mimeType and localUri fields, the SDK will upload the file to Amazon S3.

```java
input S3ObjectInput {
  bucket: String!
  key: String!
  region: String!
  localUri: String
  mimeType: String
}
```

For example, to add a photo field in the Post type. Update Post type, add the new S3ObjectInput type and add a new mutation, putPostWithPhoto.

```java
type Mutation {
  ...other mutations here...
  putPostWithPhoto(
    id: ID!,
    author: String!,
    title: String,
    content: String,
    url: String,
    ups: Int,
  }
}
Next, update the `putPostWithPhoto` mutation resolver to use `PutItemWithS3Object` template for request mapping and `Return single item` for response mapping from the AWS AppSync console.

Next, update the response mapping template for the `photo` field.

```java
#util.toJson(#util.dynamodb.fromS3ObjectJson(#context.source.file))
```

**Client Code**

Note that to use Complex Objects you need AWS IAM credentials for reading and writing to S3. These can be separate from the other authentication credentials used in the AppSync client. Credentials for Complex Objects are set in the `S3ObjectManagerImplementation` builder parameter which can be used like:

```java
public class ClientFactory {

    // ...other code...
    private static volatile AWSAppSyncClient client;
    private static volatile S3ObjectManagerImplementation s3ObjectManager;

    public static AWSAppSyncClient getInstance(Context context) {
        if (client == null) {
            client = AWSAppSyncClient.builder()
                .context(context)
                .region(Defaults.APPSYNC_REGION)
                .serverUrl(Defaults.APPSYNC_API_URL)
                .s3ObjectManager(getS3ObjectManager(context)) // Here we initialize the s3 object manager.
                .build();
        }
    }
```
public class AddPostActivity extends AppCompatActivity {

    // ...other code...

    // Photo selector application code.
    private static int RESULT_LOAD_IMAGE = 1;
    private String photoPath;

    public void choosePhoto(View view) {
        Intent i = new Intent(Intent.ACTION_PICK,android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
        startActivityForResult(i, RESULT_LOAD_IMAGE);
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        if (requestCode == RESULT_LOAD_IMAGE && resultCode == RESULT_OK && null != data) {

The SDK would upload the file found at the localUri when the bucket, key, region, localUri and mimeType are all provided. Now, the SDK would upload any field which has S3ObjectInput type in the mutation. The only requirement from a developer is to provide the correct bucket, key, region, localUri and mimeType.

Example:

Add the following permissions to AndroidManifest.xml:

<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

Then in your activity where you are adding a post, update the code to:
Uri selectedImage = data.getData();
String[] filePathColumn = {MediaStore.Images.Media.DATA};
Cursor cursor = getContentResolver().query(selectedImage, 
    filePathColumn, null, null, null);
cursor.moveToFirst();
int columnIndex = cursor.getColumnIndex(filePathColumn[0]);
String picturePath = cursor.getString(columnIndex);
cursor.close();
// String picturePath contains the path of selected Image
photoPath = picturePath;

// Actual mutation code
private void save() {
    final String title = ((EditText) 
        findViewById(R.id.updateTitle)).getText().toString();
    final String author = ((EditText) 
        findViewById(R.id.updateAuthor)).getText().toString();
    final String url = ((EditText) 
        findViewById(R.id.updateUrl)).getText().toString();
    final String content = ((EditText) 
        findViewById(R.id.updateContent)).getText().toString();

    S3ObjectInput s3ObjectInput = S3ObjectInput.builder()
        .bucket("YOUR_BUCKET_NAME")
        .key("public/"+ UUID.randomUUID().toString())
        .region("us-east-1")
        .localUri(photoPath)
        .mimeType("image/jpg").build();

    PutPostWithPhotoMutation addPostMutation = PutPostWithPhotoMutation.builder()
        .id(UUID.randomUUID().toString())
        .title(title)
        .author(author)
        .url(url)
        .content(content)
        .ups(0)
        .downs(0)
        .photo(s3ObjectInput)
        .expectedVersion(1)
        .build();

    ClientFactory.getInstance(this).mutate(addPostMutation).enqueue(postsCallback);
}

// Mutation callback code
private GraphQLCall.Callback<PutPostWithPhotoMutation.Data> postsCallback = new 
    GraphQLCall.Callback<PutPostWithPhotoMutation.Data>() {
        @Override
        public void onResponse(@Nonnull final Response<PutPostWithPhotoMutation.Data> 
            response) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Toast.makeText(AddPostActivity.this, "Added post", 
                        Toast.LENGTH_SHORT).show();
                    AddPostActivity.this.finish();
                }
            });
        }

        @Override
        public void onFailure(@Nonnull final ApolloException e) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Log.e("", "Failed to perform AddPostMutation", e);
                }
            });
        }
    }
Conflict Resolution

When clients make a mutation, either online or offline, they can send a version number with the payload (named `expectedVersion`) for AWS AppSync to check before writing to Amazon DynamoDB. A DynamoDB resolver mapping template can be configured to perform conflict resolution in the cloud, which you can learn about in the Resolver Mapping Template Reference for DynamoDB (p. 209). If the service determines it needs to reject the mutation, data is sent to the client and you can optionally run an additional callback to perform client-side conflict resolution.

For example, suppose you had a mutation with DynamoDB set for checking the version, and the client sent `expectedVersion:0`, as in this example:

```java
UpdatePostMutation updatePostMutation = UpdatePostMutation.builder()
    .input(UpdatePostInput.builder()
        .id("my-post-id")
        .title(title)
        .author(author)
        .url(url)
        .content(content)
        .ups(0)
        .downs(0)
        .expectedVersion(0)
        .build())
    .build();
ClientFactory.getInstance(this).mutate(updatePostMutation).enqueue(postsCallback);
```

This would fail the version check because `0` would be lower than any of the current values. You can then define a custom callback conflict resolver. A custom conflict resolver will receive the following variables:

- `handler`: the ConflictResolutionHandler which needs to be used to either update the value or let the mutation fail
- `serverState`: the JSONObject representing the object state in server (DynamoDB)
- `clientState`: Optional if a name of a mutation is set on a GraphQL statement
- `recordIdentifier`: the unique identifier for operation
- `operationType`: the string representation of operation type for switching between conflict resolving behavior

For example, you could have the following custom conflict resolver:

```java
class ClientConflictResolver implements ConflictResolverInterface {
    @Override
    public void resolveConflict(@Nonnull ConflictResolutionHandler handler,
        @NonNull JSONObject serverState,
        @NonNull JSONObject clientState,
        @NonNull String recordIdentifier,
        @NonNull String operationType) {

        if (operationType.equals("UpdatePostMutation")) {
            try {
```

Toast.makeText(AddPostActivity.this, "Failed to add post", Toast.LENGTH_SHORT).show();
AddPostActivity.this.finish();
```
UpdatePostMutation updatePostMutation = UpdatePostMutation.builder()
    .input(UpdatePostInput.builder()
      .id(clientState.getString("id"))
      .title(clientState.getString("title"))
      .author(clientState.getString("author"))
      .url(clientState.getString("url"))
      .content(clientState.getString("content"))
      .ups(0)
      .downs(0)
      .expectedVersion(serverState.getInt("version"))
      .build())
    .build();

handler.retryMutation(updatePostMutation, recordIdentifier);
}

} catch (JSONException je) {
  // in case of un-expected errors, we fail the mutation
  // we can also call the below method if we want server data to be accepted
  instead of client.
  handler.fail(recordIdentifier)
}

}

You will also have to provide an instance of ClientConflictResolver to the AWSAppSyncClient via a builder parameter.

client = AWSAppSyncClient.builder()
    .context(context)
    .region(Constants.APPSYNC_REGION)
    .serverUrl(Constants.APPSYNC_API_URL)
    .conflictResolver(new ClientConflictResolver()) // Here we are passing in the conflict resolver.
    .build();
Data Sources and Resolvers

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. 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 supports the automatic provisioning of DynamoDB tables from a GraphQL schema as described in (Optional) Provision from Schema (p. 27) and Launch a Sample Schema (p. 2). 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. 28).

Topics

- Tutorial: DynamoDB Resolvers (p. 91)
- Tutorial: Lambda Resolvers (p. 126)
- Tutorial: Amazon Elasticsearch Service Resolvers (p. 139)
- Tutorial: Local Resolvers (p. 144)
- Tutorial: Combining GraphQL Resolvers (p. 146)
- Tutorial: DynamoDB Batch Resolvers (p. 150)

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 provision AWS resources using the following AWS CloudFormation template:

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

You can launch this AWS CloudFormation stack in the US West 2 (Oregon) region in your AWS account by clicking this button:
This will create:

- 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:

```
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 the Create API button.
- Set the name of the API to AWSAppSyncTutorial.
- Select 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, replace the contents of the Schema pane with the following code, and then choose the Save.

```graphql
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:

- Go to the Data source tab.
- Choose New to create a new data source.
- Enter PostDynamoDBTable as the name of the data source.
- Choose Amazon DynamoDB table as the data source type.
- Choose US-WEST-2 for the region.
- Choose the AppSyncTutorial-PostDynamoDB table from the list of tables.
- Choose Existing role in the Create or use an existing role section.
- Select the arn:aws:iam::123456789012:role/AppSyncTutorialAmazonDynamoDBRole role from the list of available roles.
- 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:

• Go to the Schema tab.
• Find the addPost field on the Mutation type in the Data types pane on the right.
• Click on its Attach button.
• Select PostDynamoDBTable in the Data source name drop-down menu.
• Paste the following into the Configure the request mapping template section:

```json
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : { "S" : "${context.arguments.id}" }
    },
    "attributeValues" : {
        "author": { "S" : "${context.arguments.author}" },
        "title": { "S" : "${context.arguments.title}" },
        "content": { "S" : "${context.arguments.content}" },
        "url": { "S" : "${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 we'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 $utils.autoId() which you could have also used in the form of "id" : { "S" : "${context.arguments.id}" }. Then you could simply leave the id: ID! out of the schema definition of addPost() and it would be inserted automatically. We won't use this technique for this tutorial, but you should consider it as a good practice when writing to DynamoDB tables.

See the Resolver Mapping Template Overview (p. 182) reference documentation for more information about mapping templates, see the GetItem (p. 209) reference documentation for more information about GetItem request mapping, and see the Type System (Request Mapping) (p. 223) reference documentation for more info about types.

• Paste the following into the Configure the response mapping template section:

```json
$utils.toJson($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.

• Go to the **Queries** tab
• Paste the following mutation into the **Queries** pane

```
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
  }
}
```

• Then choose the **Execute query** button (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 something like this:

```json
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post."
    }
  }
}
```

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:

```json
{
  "version": "2017-02-28",
  "operation": "PutItem",
  "key": {
```
AWS AppSync used the request mapping document to generate and execute a DynamoDB PutItem 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 we're able to add data to the AppSyncTutorial-PostDynamoDB table, we need to set up the getPost query so it can retrieve that data from the AppSyncTutorial-Post table. To do this, we set up another resolver.

1. Go to the "Schema" tab.
2. Find the getPost field on the Query type in the "Data types" pane on the right.
3. Click on its Attach button.
4. Select PostDynamoDBTable in the "Data source name" dropdown.
5. Paste the following into the "Configure the request mapping template" section:

```json
{  
  "version" : "2017-02-28",  
  "operation" : "GetItem",  
  "key" : {  
    "id" : { "S" : "${context.arguments.id}" }  
  }
}
```

6. Paste the following into the "Configure the response mapping template" section:

```javascript
$utils.toJson($context.result)
```
• Click the Save button.

**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 DynamoDB GetItem operation. We can now run a query to retrieve the post we created earlier.

• Go to the "Queries" tab
• Paste the following mutation into the "Queries" pane.

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

• Then hit the "Execute query" button (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 something like this:

```
{
  "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 an `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.

```json
{
    "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 we can create and retrieve Post objects in DynamoDB. Next, we'll set up a new mutation to allow us to update object. We'll do this using the UpdateItem DynamoDB operation.

• Go to the "Schema" tab
• Modify the Mutation type in the "Schema" pane to add a new updatePost mutation:

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

• Click the Save button
• Find the newly created updatePost field on the Mutation type in the "Data types" pane on the right.
• Click on its Attach button.
• Select PostDynamoDBTable in the "Data source name" dropdown.
• Paste the following into the "Configure the request mapping template" section:

```json
{
    "version": "2017-02-28",
    "operation": "UpdateItem",
    "key": {
        "id": { "S": "${context.arguments.id}" }
    },
    "update": {
        "expression": "SET author = :author, title = :title, content = :content, #url = :url ADD version :one",
        "expressionNames": {
```
Create an updatePost mutation (DynamoDB UpdateItem)

```json
{
  "#url" : "url"
}
```

**Note:** This resolver is using the DynamoDB UpdateItem, which is significantly different from the PutItem operation. Instead of writing the entire item, we'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 we can use name placeholders and define them in the expressionNames field.

See the UpdateItem (p. 212) reference documentation for more info about UpdateItem request mapping, and the DynamoDB UpdateExpressions documentation for more information on how to write update expressions.

- Paste the following into the "Configure the response mapping template" section:

```java
$utils.toJson($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. We can now run a mutation to update the item we wrote earlier.

- Go to the "Queries" tab
- Paste the following mutation into the "Queries" pane. You'll also need to update the id argument to have the value we noted down earlier.

```graphql
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 
  } 
}
```

- Then hit the "Execute query" button (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 something like this:

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

Note that the `ups` and `downs` fields were not modified. This is because our request mapping template did not ask AWS AppSync and DynamoDB to do anything with those fields. Also note that the `version` field was incremented by 1. This is because we asked AWS AppSync and DynamoDB to add 1 to the `version` field.

### Modifying the "updatePost" resolver (DynamoDB UpdateItem)

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

• If I want to update just a single field, then I have to update all the fields.
• If two people are modifying the object, then we potentially lose information.

To address these issues, we'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.

• Go to the "Schema" tab.
• Modify the `updatePost` field in the `Mutation` type in the "Schema" pane 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.

```graphql
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!
}
```

• Click the Save button
• Find the `updatePost` field on the Mutation type in the "Data types" pane on the right.
• Click on the `PostDynamoDBTable` link to open the existing resolver.
• Modify the request mapping template in the "Configure the request mapping template" section:

```json
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : { "S" : "{context.arguments.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", ":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 we're going to SET **
  #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

  ## Continue building the update expression, adding attributes we're going to ADD **
  #if( !$!{expAdd.isEmpty()} )
    #set( $expression = "ADD" )
    #foreach( $entry in $expAdd.entrySet() )
      #set( $expression = "$expression ${entry.key} ${entry.value}" )
    #if ( $foreach.hasNext )
      #set( $expression = "$expression," )
    #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
```

** 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" : { "N" : ${context.arguments.expectedVersion} }
  }
}
```

- Click the **Save** button.

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

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

See the Condition Expressions (p. 230) reference documentation for more information about condition expressions.

### Call the API to update a Post

Lets try updating our Post object with the new resolver:

- Go to the "Queries" tab
- Paste the following mutation into the "Queries" pane. You'll also need to update the `id` argument to have the value we noted down earlier.

```python
mutation updatePost {
```
updatePost(
  id:123
  title: "An empty story"
  content: null
  expectedVersion: 2
)

• Then hit the "Execute query" button (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 something like this:

```
{
  "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
    }
  }
}
```

Note that in this request, we only asked AWS AppSync and DynamoDB to update the title and
content field. It left all the other fields alone (other than incrementing the version field). We 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 will see a response
similar to the following:

```
{
  "data": {
    "updatePost": null
  },
  "errors": []
}
```

```
Create upvotePost and downvotePost mutations (DynamoDB UpdateItem)

The request fails because the condition expression evaluates to false:

- The first time we 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 we 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 AWS AppSync’s 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 our case, we can see the version field of the object in DynamoDB is set to 3, so we could just update the expectedVersion argument to 3 and the request would succeed again.

See the Condition Expressions (p. 230) mapping template reference documentation for more information about handling condition check failures.

Create upvotePost and downvotePost mutations (DynamoDB UpdateItem)

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

- Go to the "Schema" tab
- Modify the Mutation type in the "Schema" pane to add new upvotePost and downvotePost mutations:

```graphql

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!,
```

Notice: The fortran code seems to be incomplete or perhaps was meant as an example of how to use the AWS AppSync. It's hard to tell due to the nature of the code being included in the text. The code appears to be a part of the AWS AppSync AWS AppSync Developer Guide, but it’s not fully visible in the image.
{  
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : { "S" : "${context.arguments.id}" }  
  },
  "update" : {
    "expression" : "ADD ups :plusOne, version :plusOne",
    "expressionValues" : {
      ":plusOne" : { "N" : 1 }
    }
  }
}

$utils.toJson($context.result)

{  
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : { "S" : "${context.arguments.id}" }  
  },
  "update" : {
    "expression" : "ADD downs :plusOne, version :plusOne",
    "expressionValues" : {
      ":plusOne" : { "N" : 1 }
    }
  }
}

$utils.toJson($context.result)
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 `downvotePost` mutation to DynamoDB UpdateItem operation. We can now run mutations to upvote or downvote the post we created earlier.

- Go to the "Queries" tab
- Paste the following mutation into the "Queries" pane. You'll also need to update the `id` argument to have the value we noted down earlier.

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

- Then hit the "Execute query" button (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 something like this:

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

- Click the "Execute query" button 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:

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

- Then hit the "Execute query" button (the orange play button). This time, you should see the `downs` and `version` field incrementing by 1 each time you execute the query.
Setting up the "deletePost" resolver (DynamoDB DeletePost)

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

- Go to the "Schema" tab
- Modify the Mutation type in the "Schema" pane to add a new deletePost mutation:

```graphql
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!
}
```

Note that this time we made the expectedVersion field optional. The reason for this will be explained when we add the request mapping template.

- Click the Save button
- Find the newly created delete field on the Mutation type in the "Data types" pane on the right.
- Click on its Attach button.
- Select PostDynamoDBTable in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```json
{
  "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
    }
  }
}
```
Setting up the "deletePost" resolver (DynamoDB DeletePost)

```json
"key": {
    "id": { "S" : ${context.arguments.id} }
}
#if( $context.arguments.containsKey("expectedVersion") )
    ,"condition" : {
        "expression": "attribute_not_exists(id) OR version = :expectedVersion",
        "expressionValues": {
            ":expectedVersion" : { "N" : ${context.arguments.expectedVersion} }
        }
    }
#end
```

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

- Paste the following into the "Configure the response mapping template" section:

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

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

- Click the Save button.

See the [DeleteItem](p. 216) reference documentation for more info about DeleteItem request mapping.

**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 DynamoDB DeleteItem operation. We can now run a mutation to delete something in the table.

- Go to the "Queries" tab
- Paste the following mutation into the "Queries" pane. You'll also need to update the `id` argument to have the value we noted down earlier.

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

- Then hit the "Execute query" button (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 something like this:
The value is only returned if this call to `deletePost` was the one that actually deleted it from DynamoDB.

- Try hitting the "Execute query" button again.
- The call still succeeds, but no value is returned.

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

- Paste the following mutation into the "Queries" pane

```graphql
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
  }
}
```

- Then hit the "Execute query" button (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, as we'll need it in just a moment. It should look something like this:

```json
{
}
```
"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 and delete that post, but we'll put in the wrong value for `expectedVersion`

- Paste the following mutation into the "Queries" pane. You'll also need to update the `id` argument to have the value we noted down earlier.

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

- Then hit the "Execute query" button (the orange play button).

```json
{
  "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
        }
      ]
    }
  ]
}
```
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`:

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

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

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

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

```json
{
  "data": {
    "deletePost": null
  }
}
```
Setting up the "allPost" resolver (DynamoDB Scan)

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

- Go to the "Schema" tab
- Modify the Query type in the "Schema" pane to add a new allPost query:

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

- Add a new PaginationPosts type in the "Schema pane:

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

- Click the Save button
- Find the newly created allPost field on the Query type in the "Data types" pane on the right.
- Click on its Attach button.
- Select PostDynamoDBTable in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```json
{
  "version": "2017-02-28",
  "operation": "Scan"
  #if( ${context.arguments.count} )
  "limit": ${context.arguments.count}
  #end
  #if( ${context.arguments.nextToken} )
  "nextToken": ${context.arguments.nextToken}
  #end
}
```

Note that 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 (we'll show where the value for nextToken comes from later).

- Paste the following into the "Configure the response mapping template" section:

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

- Click the Save button.
See the Scan (p. 220) reference documentation for more info about Scan request mapping.

Call the API to scan all Posts

Now the resolver has been set up, AWS AppSync knows how to translate an incoming `allPost` mutation to a DynamoDB Scan operation. We can now scan the table to retrieve all the posts.

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

- Go to the "Queries" tab
- Paste the following mutation into the "Queries" pane

```javascript
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 }
  post8: addPost(id:8 author: "AUTHORNAME" title: "A series of posts, Volume 8" content: "Some content" url: "https://aws.amazon.com/appsync/ " ) { title }
}
```

- Then hit the "Execute query" button (the orange play button).

Now, let's scan the table, returning 5 results at a time.

- Paste the following query in the "Queries" pane

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

- Then hit the "Execute query" button (the orange play button).
- The first 5 posts should appear in the results pane to the right of the query pane. It should look something like this:

```json
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        }
```
We can see that we got 5 results and also a nextToken that we can use to get the next set of results.

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

```graphql
query allPost {
  allPost(
    count: 5
    nextToken:
      "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI3Z0inRkJEdXJkOTUyRkNqZ293V2ZnRnBMZkFybWpOa2VtRndnU1d2dEhOd2NNcT9zaVJhck9BeGRoQ1pybXh2Smh2aWl3a2JzS2dLdHd1a3Zadz09I"
  ) {
    posts {
      id
      author
    }
    nextToken
  }
}
```

- Then hit the "Execute query" button (the orange play button).

- The remaining 4 posts should appear in the results pane to the right of the query pane. There is no nextToken in this set of results as we have paged through all 9 posts, with none remaining. It should look something like this:

```json
{
  "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": "5",
        "title": "A series of posts, Volume 5"
      },
      {
        "id": "6",
        "title": "A series of posts, Volume 6"
      },
      {
        "id": "7",
        "title": "A series of posts, Volume 7"
      },
      {
        "id": "8",
        "title": "A series of posts, Volume 8"
      },
      {
        "id": "9",
        "title": "A series of posts, Volume 9"
      }
    }
  }
}
```
Setting up the "allPostsByAuthor" resolver (DynamoDB Query)

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

- Go to the "Schema" tab
- Modify the Query type in the "Schema" pane to add a new allPostsByAuthor query:

```golang
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 we used with the allPost query.

- Click the Save button
- Find the newly created allPostsByAuthor field on the Query type in the "Data types" pane on the right.
- Click on its Attach button.
- Select PostDynamoDBTable in the "Data source name" dropdown.
- Paste the following into the "Configure the request mapping template" section:

```json
{
  "version" : "2017-02-28",
  "operation" : "Query",
  "index" : "author-index",
  "query" : {
    "expression": "author = :author",
    "expressionValues" : {
      ":author" : { "S" : "${context.arguments.author}" }
    }
  }

  #if( ${context.arguments.count} )
  ,"limit": ${context.arguments.count}
  #end

  #if( ${context.arguments.nextToken} )
  ,"nextToken": "${context.arguments.nextToken}" 
  #end
}
```
Note that 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).

- Paste the following into the "Configure the response mapping template" section:

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

**Note**: This is the same response mapping template we used in the allPost resolver.

- Click the Save button.

See the Query (p. 217) reference documentation for more info about Query request mapping.

### 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 DynamoDB Query operation against the author-index index. We can now query the table to retrieve all the posts by a specific author.

Before we do that, however, lets populate the table with some more posts, because every post so far has the same author.

- Go to the "Queries" tab
- Paste the following mutation into the "Queries" pane

```graphql
mutation addPost {
  post3: addPost(id:12 author: "Steve" title: "I like GraphQL" content: "It's great" url: "https://aws.amazon.com/appsync/" ) { author, title }
}
```

- Then hit the "Execute query" button (the orange play button).

Now, lets query the table, returning all posts authored by Nadia.

- Paste the following query in the "Queries" pane

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

- Then hit the "Execute query" button (the orange play button).
Setting up the "allPostsByAuthor" resolver (DynamoDB Query)

• All the posts authored by Nadia should appear in the results pane to the right of the query pane. It should look something like this:

```
{
  "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 5 at a time.

• Paste the following query in the "Queries" pane

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

• Then hit the "Execute query" button (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 something like this:

```
{
  "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",
```
• Update the `nextToken` argument with the value returned from the previous query:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
    nextToken: "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RGoxWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI3Z0lnSEX...XNGJXdDk0VEg3b0laUU5lYmZYKzVOKy9Td25Hb1dyMTlWK0pEb2lIRVFLZ1cwMWVuYjZKUXo5Slh2Tm95ZzF3RnJPVmxGc2xwNlRHa1BlN2Rnd2IrWT0ifQ=="
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

• Then hit the "Execute query" button (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 something like this:

```
{
  "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

So far our Post type has been a flat key/value object. It’s also possible to model complex objects with the AWS AppSyncDynamoDB resolver, such as sets, lists, and maps.

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

- Go to the "Schema" tab
- Modify the Post type in the "Schema" pane to add a new tags field:

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

- Modify the Query type in the "Schema" pane to add a new allPostsByTag query:

```graphql
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
}
```

- Modify the Mutation type in the "Schema" pane to add new addTag and removeTag mutations:

```graphql
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!
}
```

- Click the Save button
- Find the newly created allPostsByTag field on the Query type in the "Data types" pane on the right.
- Click on its Attach button.
• Select PostDynamoDBTable in the "Data source name" dropdown.
• Paste the following into the "Configure the request mapping template" section:

```json
{
  "version" : "2017-02-28",
  "operation" : "Scan",
  "filter": {
    "expression": "contains (tags, :tag)",
    "expressionValues": {
      ":tag": { "S": "${context.arguments.tag}" }
    }
  }
}
#if( ${context.arguments.count} )
  ,"limit": ${context.arguments.count}
#end
#if( ${context.arguments.nextToken} )
  ,"nextToken": "${context.arguments.nextToken}"
#end
```

• Paste the following into the "Configure the response mapping template" section:

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

• Click the Save button.
• Find the newly created addTag field on the Mutation type in the "Data types" pane on the right.
• Click on its Attach button.
• Select PostDynamoDBTable in the "Data source name" dropdown.
• Paste the following into the "Configure the request mapping template" section:

```json
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : { "S" : "${context.arguments.id}" }
  },
  "update" : {
    "expression" : "ADD tags :tags, version :plusOne",
    "expressionValues" : {
      ":tags" : { "SS": [ "${context.arguments.tag}" ] },
      ":plusOne" : { "N" : 1 }
    }
  }
}
```

• Paste the following into the "Configure the response mapping template" section:

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

• Click the Save button.
• Find the newly created removeTag field on the Mutation type in the "Data types" pane on the right.
• Click on its Attach button.
• Select PostDynamoDBTable in the "Data source name" dropdown.
• Paste the following into the "Configure the request mapping template" section:

```json
{
   "version": "2017-02-28",
   "operation": "UpdateItem",
   "key": {
      "id": { "S": "${context.arguments.id}" }
   },
   "update": {
      "expression": "DELETE tags :tags ADD version :plusOne",
      "expressionValues": {
         ":tags" : { "SS": [ "${context.arguments.tag}" ] },
         ":plusOne" : { "N": 1 }
      }
   }
}
```

• Paste the following into the "Configure the response mapping template" section:

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

• Click the Save button.

**Call the API to work with tags**

Now the resolvers have been set up, AWS AppSync knows how to translate incoming addTag, removeTag, and allPostsByTag requests into DynamoDB UpdateItem and Scan operations.

To try it out, let's select one of the posts we created earlier. For example, let's use one of Nadia's posts.

• Go to the "Queries" tab
• Paste the following query into the "Queries" pane.

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

• Then hit the "Execute query" button (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 something like this:

```json
{
   "data": {
      "allPostsByAuthor": {
         "posts": [
            { "id": "10", "title": "The cutest dog in the world" },
            { "id": "11", }...
```
• Let's use the one with the title "The cutest dog in the world". Note down its id because we'll use it later.

Now let's try adding a "dog" tag.

• Paste the following mutation into the "Queries" pane. You'll also need to update the id argument to have the value we noted down earlier.

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

• Then hit the "Execute query" button (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"
      ]
    }
  }
}
```

We can add more tags as well.

• Update the mutation to change the tag argument to "puppy".

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

• Then hit the "Execute query" button (the orange play button).
• The post is updated with the new tag.

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

- Paste the following mutation into the "Queries" pane. You'll also need to update the id argument to have the value we noted down earlier.

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

- Then hit the "Execute query" button (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"
      ]
    }
  }
}
```

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

- Paste the following query into the "Queries" pane.

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

- Then hit the "Execute query" button (the orange play button).
- All posts that have the "dog" tag are returned:

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
      
```
Using Lists and Maps

In addition to using DynamoDB Sets, we 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 our Post object in DynamoDB.

Note: in a real application, we would model comments in their own table, however for the purpose of this tutorial we will just add them in the Post table.

• Go to the "Schema" tab
• Add a new Comment type in the "Schema" pane:

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

• Modify the Post type in the "Schema" pane to add a new comments field:

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

• Modify the Mutation type in the "Schema" pane to add a new addComment mutation:

```graphql
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,
    ...
• Click the `Save` button.
• Find the newly created `addComment` field on the `Mutation` type in the "Data types" pane on the right.
• Click on its `Attach` button.
• Select `PostDynamoDBTable` in the "Data source name" dropdown.
• Paste the following into the "Configure the request mapping template" section:

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : { "S" : "${context.arguments.id}" } 
  },
  "update" : {
    "expression" : "SET comments = list_append(if_not_exists(comments, :emptyList), :newComment) ADD version :plusOne",
    "expressionValues" : {
      ":emptyList": { "L" : [] },
      ":newComment" : { "L" : [
        { "M" : {
          "author": { "S" : "${context.arguments.author}" },
          "comment": { "S" : "${context.arguments.comment}" } 
        }
      ] },
      ":plusOne" : { "N" : 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.

• Paste the following into the "Configure the response mapping template" section:

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

• Click the `Save` button.

### Call the API to add a comment

Now the resolvers have been set up, AWS AppSync knows how to translate incoming `addComment` requests into DynamoDB `UpdateItem` operations.

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

• Go to the "Queries" tab
• Paste the following query into the "Queries" pane.

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

• Then hit the "Execute query" button (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 something like this:

```json
{
  "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 we've built an API that lets us manipulate Post objects in DynamoDB using AWS AppSync and GraphQL. For further information check out the [Resolver Mapping Template Reference](p. 182).

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

To delete the DynamoDB table and IAM role we created, 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

AWS AppSync allows you to use AWS Lambda to resolve any GraphQL field. For example, a GraphQL query might call out to an Amazon RDS instance, and a GraphQL mutation might write to a Amazon Kinesis stream. This section outlines how you can 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 example.

```javascript
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 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5", "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 "addPostWithErrorWithData":
      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);
  }
};
```
This Lambda function handles retrieving a post by ID, adding a post, retrieving a list of posts, and fetching related posts for a given post.

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

Now let's create this Lambda function using the AWS console or with AWS CloudFormation by clicking here:

```
aws cloudformation create-stack --stack-name AppSyncLambdaExample \
  --template-url https://s3-us-west-2.amazonaws.com/awsappsync/resources/lambda/\n  LambdaCFTemplate.yaml \
  --capabilities CAPABILITY_NAMED_IAM
```

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

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

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

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

From the schema editor in the AWS AppSync console, make sure you schema matches the schema below.

```
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]
}
```

Configuring resolvers

Now that we have registered an AWS Lambda data source and a valid GraphQL schema, we can connect our GraphQL fields to our Lambda data source using resolvers.

To create a resolver, we’ll need mapping templates. To learn more about mapping templates, read AppSync mapping templates overview Resolver Mapping Template Overview (p. 182).

For more information about Lambda mapping templates, see Resolver Mapping Template Reference for Lambda (p. 241).

We are going to attach a resolver to our 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-hand side for `getPost(id:ID!)`: `Post` click **Attach Resolver**.

Select your AWS Lambda data source and under the **request mapping template** section select the dropdown menu for **Invoke And Forward Arguments**.

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

```javascript
{
  "Action": "sts:AssumeRole"
}
"}
}
Now under the **response mapping template** section, select the drop-down menu for **Return Lambda Result**.

We will use the base template as-is. It should look like the following:

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

Click **Save**. You have now attached your first resolver! Repeat this operation for the remaining fields as follows:

**addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!**

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

**addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!**

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

**allPosts: [Post]**

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

**allPosts: [Post]**

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

**Post.relatedPosts: [Post]**

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

In the AppSync console, on the left-hand side, choose the Queries tab. Populate it with the following code:

### addPost mutation

```graphql
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

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

### allPosts query

```graphql
query {
  allPosts {
    id
    author
    title
    content
    url
    ups
  }
}
```
Returning Errors

Any given field resolution can result in an error. AppSync lets you raise errors:

- From the request or response mapping template
- From the Lambda function

From the mapping template

The $utils.error helper method can be used from the VTL template to raise intentional errors. It takes as argument an errorMessage, an errorType, and an optional data value. The data comes handy for returning extra data back to the client, when an error has been raised. The data object will be added to the errors in the GraphQL final response.

For example using it in the Post.relatedPosts: [Post] response mapping template

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

would yield 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"
      ]
    }
  ]
}
```
Returning Errors

```
{
  "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

AppSync also understands errors thrown from the Lambda function. The Lambda programming model lets you raise **Handled** errors. If an error is thrown from the Lambda function, AppSync will fail the resolution of the current field. Only the error message returned from Lambda will be set in the response. Also, it is currently impossible to 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, AppSync will use the error message set by AWS Lambda.

The following Lambda function raises an error:

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

Which would return a GraphQL response similar to below:

```
{
  "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

You may have noticed that the Lambda function in our example had a relatedPosts field which returned a list of related posts for a given post. In our example queries, the allPosts field invocation from our Lambda function returns 5 posts. Because we have specified that we also want to resolve relatedPosts for each returned post, the relatedPosts field operation will, in turn, be invoked 5 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 doesn't sound substantial for this specific use case, our application can get quickly undermined by this compounded over-fetching.

If, say, we 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 our Lambda function $1 + 5 + 25 = 31$ times.

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

One approach to solving this issue is to batch similar field resolver requests together. So in our example, instead of our Lambda function resolving a list of related posts for a single given post, it would instead resolve a list of related posts for a given batch of posts.

To see it in action, let's switch our Post.relatedPosts: [Post] resolver to a batch-enabled resolver.
In the AWS AppSync console, on the right-hand side, choose the existing Post.relatedPosts: [Post] resolver. Change the request mapping template to the following:

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

Note that only the `operation` field has changed from `Invoke` to `BatchInvoke`. The payload field now becomes an array of whatever has been specified in the template, so in our example, our Lambda function will receive as input:

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

When `BatchInvoke` is specified in the request mapping template, the Lambda function is now given a list of requests and is also expected to return a list of results.

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

So in our example, because of batching, our Lambda function needs to return a batch of results:

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

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

```javascript
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 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 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 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5", "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**

We saw previously that it is 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 will flag an entire batch as failed. This might be fine for specific scenarios where an irrecoverable error happened, such as, the connection to a data store going down. However, in cases where some items in the batch succeed, and some others fail, let see how it is possible to return both errors and valid data. AppSync only imposes the batch response to be a list of elements matching the original size of the batch, it is up to you to define a data structure that can differentiate valid data from an error.

For instance, if our Lambda function is expected to return a batch of related posts, we could instead return a list of *Response* object where each object has optional *data*, *errorMessage* and *errorType* fields. If the *errorMessage* field is present, it means there was an error.

See below the updated Lambda function.

```javascript
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 SAMPLE TEXT AUTHOR 1", "ups": "100", "downs": "10"},
```
And we could write a response mapping template to parse each item of our Lambda function, and raise an error if needed:

```
#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 would return a GraphQL response similar to below:

```
{
  "data": {
    "allPosts": [
      {
        "id": "1",
        "relatedPostsPartialErrors": [
          {
            "id": "4",
            "relatedPosts": {
              "1": [posts['4']],
              "2": [posts['3'], posts['5']],
              "3": [posts['2'], posts['1']],
              "4": [posts['2'], posts['1']],
              "5": []
            }
          }
```
"title": "Fourth book"
],
},
{
"id": "3",
"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": 3,
"column": 9
}
]
Tutorial: Amazon Elasticsearch Service Resolvers

AWS AppSync supports using Amazon Elasticsearch Service from domains that you have provisioned in your own AWS account. 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 Elasticsearch (p. 238).

One-Click Setup

If you wish to automatically setup a GraphQL endpoint in AWS AppSync with Amazon Elasticsearch Service configured you can use the following AWS CloudFormation template:

```
aws cloudformation create-stack --stack-name AppSyncElasticsearch \
 --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 this AWS CloudFormation stack in the US West 2 (Oregon) region in your AWS account:

Configure Data Source for Amazon ES

After the Amazon ES 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 "Elasticsearch". Then choose Amazon Elasticsearch domain for Data source type, choose the appropriate region, and
you should see your Amazon ES 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:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1234234",
            "Effect": "Allow",
            "Action": [
                "es:ESHttpDelete",
                "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:

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

Additionally, the Amazon ES domain has its own Access Policy which you can modify through the
Amazon Elasticsearch Service console. You will need to add a policy similar to the below, with the
appropriate actions and resource for the Amazon ES domain. Note that the Principal will be the
AppSync data source role, which if you let the console create this would start with the name of appsync-
datasource-es- and can be found in the AWS IAM console.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::ACCOUNTNUMBER:role/service-role/APPSYNC_DATASOURCE_ROLE"
            },
            "Action": [
                "es:ESHttpDelete",
                "es:ESHttpGet",
                "es:ESHttpPost",
                "es:ESHttpPut"
            ],
        }
    ]
}
```
Connecting a Resolver

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

```graphql
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 Amazon ES 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:

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

Save the schema. On the right side, for `searchPosts`, choose **Attach resolver**. Choose your Amazon ES 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:

```json
{
  "version":"2017-02-28",
  "operation":"GET",
  "path":"/id/post/_search",
}
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:

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

Now choose Attach resolver and select the Amazon ES data source, but use the following example in the response mapping template:

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

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

## Adding Data to Amazon ES

You may want to add data to your Amazon ES 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, it serves as a mechanism for notifying clients of updates to data in your Amazon ES domain.

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

```json
{
    "version":"2017-02-28",
    "operation":"PUT",
    "path": "/id/post/$context.arguments.id",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "id": "$context.arguments.id",
            "author": "$context.arguments.author",
            "ups": "$context.arguments.ups",
            "downs": "$context.arguments.downs",
            "url": "$context.arguments.url",
            "content": "$context.arguments.content",
            "title": "$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:

```javascript
$utils.toJson($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 Amazon ES data source again and use the following mapping template:

```json
{
    "version":"2017-02-28",
    "operation":"GET",
    "path": "/id/post/$context.arguments.id",
    "params":{
        "headers":{}
    }
}
```
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:

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

## Perform Queries and Mutations

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

```graphql
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 Amazon ES domain:

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

## Best Practices

- Amazon ES should be for querying data, not as your primary database. You may want to use Amazon ES in conjunction with Amazon DynamoDB as outlined in Combining GraphQL Resolvers.
- 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 Elasticsearch 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:

```graphql
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:

```json
{
  "version": "2017-02-28",
  "payload": {
    "body": "${context.arguments.body}"
    "from": "${context.identity.username}"
    "to": "${context.arguments.to}"
    "sentAt": "$util.time.nowISO8601()"
  }
}
```

And for the response mapping template, select the default Forward the result. Save your resolver. You 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:

```graphql
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:

```graphql
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 show how you might mix and match data sources in your schema, but before doing so you should have familiarity with setting up data sources and resolvers for AWS Lambda, Amazon DynamoDB and Amazon Elasticsearch Service as outlined in the previous sections.

Example Schema

The below schema has a type of *Post* with 3 Query operations and 3 Mutation operations defined:

```graphql
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. 209). However, there are alternatives to meet your business needs, such as having these GraphQL queries resolve from AWS Lambda or Amazon ES.

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 compatability issues. For illustrative purposes in the below example, we show an AWS Lambda function that manipulates the up-votes and down-votes for blog posts by assigning them random numbers each time the GraphQL resolver is invoked:

```javascript
'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 Amazon ES

For some applications, you might perform mutations or simple lookup queries against DynamoDB, and have a background process transfer documents to Amazon ES. You can then simply attach the `searchPosts` Resolver to the Amazon ES 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 as keyword, fuzzy word matches or even geospatial lookups. Transfering 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 below AWS CloudFormation stack in the US West 2 (Oregon) region in your AWS account:

The schema in this example will let you add posts using a DynamoDB resolver like so:

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

This will write data to DynamoDB which then streams data via Lambda to Amazon Elasticsearch Service which you could search for all posts by different fields. For example, since the data is in Amazon Elasticsearch Service you can search either the author or content fields with freeform text, even with spaces, like so:

```graphql
query searchName{
  searchAuthor(name:"Nadia "){
    id
    title
    content
  }
}
query searchContent{
  searchContent(text:"test"){
    id
    title
    content
  }
}
```

Since the data is written direct 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.

```javascript
var AWS = require('aws-sdk');
```
```javascript
var path = require('path');
var stream = require('stream');

var esDomain = {
    endpoint: 'https://elasticsearch-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;
            default:
                break;
        }
    }
```

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('"
') + '"
);
}
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 Amazon ES domain. For more information on 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:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:BatchGetItem",
        "dynamodb:BatchWriteItem"
      ],
      "Effect": "Allow",
      "Resource": [
        "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:

```json
{
  "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/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:

```graphql
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:

```graphql
#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)
  }
}
```

The Response Mapping Template in this case is a simple passthrough, but note that that the table name is appended as ..data.Posts to the context object:

```graphql
#util.toJson($ctx.result.data.Posts)
```

Now navigate to the Queries page of the AWS AppSync console and run the following batchAdd mutation:

```graphql
mutation add {
  batchAdd(posts:[{
```
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:

```java
#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:

```java
$util.toJson($ctx.result.data.Posts)
```

Now go back to the Queries page of the AWS AppSync console, and run the following batchGet Query:

```graphql
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 BatchGetItem operation:

```java
#set($ids = [])
```
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 temperature readings and temperatureReadings will store sensor location 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:

```graphql
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

```bash
# 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
```
As you can see, the BatchPutItem operation allows us to specify multiple tables.

Let's use the following response mapping template.

**Response Mapping Template**

```cpp
## If there was an error with the invocation
## there might have been partial results
#(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:

```graphql
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
  }
}
```
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**

```java
## 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**

```java
## 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:

```graphql
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

```json
## 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**

```plaintext
#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:

```graphql
query getReadingsForSensorAndTime {
  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.

```json
{
  "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:

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

GraphQL response:

```json
{
  "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:

```javascript
## there was an error let's mark the entire field
```

##
## Error Handling

```graphql
# if ($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type, null, $ctx.result.data.unprocessedKeys)
#end

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

**GraphQL response:**

```json
{
  "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:**

```graphql
# 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
```

162
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:

```json
{
  "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. (...)"
      }
    ]
  }
}
```
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. Subscription connection management is handled automatically by the AWS AppSync client SDK using MQTT over Websockets as the network protocol between the client and service.

Note: To control authorization at connection time to a subscription, you can leverage controls such as AWS IAM, Amazon Cognito Identity, 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. Please see Security (p. 167) for more information.

Subscriptions are triggered from mutations and the mutation selection set is sent to subscribers.

The following example shows how to work with GraphQL subscriptions. Notice that it doesn’t specify a data source, because the data source could be AWS Lambda, Amazon DynamoDB, or Amazon Elasticsearch Service.

To get started to with subscriptions, you must add a subscription entry point to your schema:

```graphql
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. You would add the following `Subscription` definition to your schema:

```graphql
type Subscription {
    addedPost: Post
    updatedPost: Post
    deletedPost: Post
}
```

Suppose further that you have the following mutations:

```graphql
type Mutation {
    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:

```graphql
type Subscription {
```
Using Subscription Arguments

An important part of using GraphQL subscriptions is understanding when and how to use arguments, as subtle changes will allow you to modify how and when clients are notified of mutations that have occurred. To do this, refer to the sample schema from the Quickstart section, which creates "Events" and "Comments". For the sample schema, you will see the following mutation:

```graphql
mutation addPost
{
  addPost
  @aws_subscribe(mutations: ["addPost"], arguments: {
    id: "XYZ",
    author: "ABC"
  })
  {
    title
    content
  }
}
```

To set up subscriptions on the client, see Building a Client App (p. 32).
Using Subscription Arguments

```graphql
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:

```graphql
type Subscription {
  subscribeToEventComments(eventId: String!): Comment
  @aws_subscribe(mutations: ["commentOnEvent"])
}
```

However, if you want to allow 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:

```graphql
subscribeToEventComments(eventId: String): Comment
```

With this change, clients that omitted this argument would get comments for all events. Additionally if you wanted clients to explicitly subscribe to all comments for all events, you would remove the argument:

```graphql
subscribeToEventComments(): Comment
```

These are for comments on one or more events. If you just wanted to know about all events that get created, you might do something like this:

```graphql
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:

```graphql
type Subscription {
  subscribePlaceDate(where: String! when: String!): Event
  @aws_subscribe(mutations: ["createEvent"])
}
```

The client application could now do this:

```graphql
subscription myplaces {
  subscribePlaceDate(where: "Seattle" when: "Saturday"){
    id
    name
    description
  }
}
```
Security

Topics

- API_KEY Authorization (p. 77)
- AWS_IAM Authorization (p. 168)
- OPENID_CONNECT Authorization (p. 169)
- AMAZON_COGNITO_USER_POOLS Authorization (p. 169)
- Fine-Grained Access Control (p. 170)
- Filtering Information (p. 172)
- Authorization Use Cases (p. 173)

This section describes options for configuring security and data protection for your applications.

There are four 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_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.

**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 } }" }' http://YOURAPPSYNCENDPOINT/graphql
```
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:

```json
{
  "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.

```json
{
  "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:

```graphql
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:

```json
{
    "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 e.g. 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 supports RS256, RS384, RS512, PS256, PS384, PS512, HS256, HS384, HS512, ES256, ES384, ES512 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.

**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:

```graphql
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:

```graphql
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"}'
```

## 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 itself based on certain conditions - such as who the user is that is making a call and whether they own the data - you can do use mapping templates in your resolvers. You can also perform more complex business logic, which we describe in Filtering Information (p. 172).

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. 182) and the Resolver Mapping Template Reference for DynamoDB (p. 209).
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:

```graphql
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:

```json
{
    "accountId" : "12321434323",
    "cognitoIdentityPoolId" : "",
    "cognitoIdentityId" : "",
    "sourceIP" : "",
    "caller" : "ThisistheprincipalARN",
    "username" : "username",
    "userArn" : "Sameasabove"
}
```

To use this object in a DynamoDB `UpdateItem` 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" : { "S" : "${context.arguments.id}" }
  },
  "attributeValues" : {
    "Author" : { "S" : "${context.identity.username}" }
  }
}
```

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" : "UpdateItem",
  "key" : {
    "postId" : { "S" : "${context.arguments.id}" }
  },
  "attributeValues" : {
    "Author" : { "S" : "${context.identity.username}" },
    "$entry.key" : { "S" : "$entry.value" }
  }
}
```

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" : { "S" : "${context.arguments.id}" }
  },
  "attributeValues" : {
    "Author" : { "S" : "${context.identity.username}" }
}
```

"operation" : "GetItem",
"key" : {
   "postId" : { "S" : "${context.arguments.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.

Authorization Use Cases

In the Security (p. 167) 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. 182), 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. 184).

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. 195), 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:

<table>
<thead>
<tr>
<th>ID</th>
<th>Data</th>
<th>PeopleCanAccess</th>
<th>GroupsCanAccess</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>{my: data,...}</td>
<td>[Mary, Joe]</td>
<td>[Admins, Editors]</td>
<td>Nadia</td>
</tr>
</tbody>
</table>

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. 209).

In the resolver mapping template overview (p. 182) 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 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:

```plaintext
#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 AWS 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. 195).

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" : { "S" : "${context.argument.id}" }
  },
  "attributeValues" : {
    "owner" : {"S" : "${context.identity.username}"}
  }
}
#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.
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:

```java
#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}) ${context.arguments.count} #else 20 #end, 
  "nextToken": #if(${context.arguments.nextToken}) ${context.arguments.nextToken} #else null #end, 
  "filter":{
    "expression": "$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 the Security section (p. 167) an example was given 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

```java
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : { "S" : "${context.arguments.id}" }
  },
  "update" : {
    "expression" : "SET meta = :meta",
    "expressionValues": {
```

176
Use Case: Group Can Create New Record

```java
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
#set( $expression = "contains(groupsCanAccess, :var$foreach.count )" )
#set( $val = {} )
#set( $test = $val.put("S", $group) )
#set( $values = $expressionValues.put("var$foreach.count", $val) )
#if ( $foreach.hasNext )
#set( $expression = "OR" )
#end
#end

{ "version" : "2017-02-28",
 "operation" : "PutItem",
 "key" : {
 "id" : { "S" : "$context.arguments.id" }
 },
 "attributeValues" : {
 "title" : { "S" : "$context.arguments.title" },
 "content" : { "S" : "$context.arguments.content" },
 "owner" : { "S" : "$context.identity.username" }
 },
 "condition" : {
 "expression" : "attribute_not_exists(id) OR $expression",
 "expressionValues" : $utils.toJson($expressionValues)
 }
}
```

Use Case: Group Can Update Existing Record

```java
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
#set( $expression = "contains(groupsCanAccess, :var$foreach.count )" )
#set( $val = {} )
#set( $test = $val.put("S", $group) )
#set( $values = $expressionValues.put("var$foreach.count", $val) )
#if ( $foreach.hasNext )
#set( $expression = "OR" )
#end
#end

{ "version" : "2017-02-28",
 "operation" : "UpdateItem",
 "key" : {
 "id" : { "S" : "$context.arguments.id" }
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 whith 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:

```plaintext
#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:

```plaintext
#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
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:

```graphql
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:

```json
{
    "version": "2017-02-28",
    "operation": "PutItem",
    "key": {
        "toUser": { "S": "${ctx.args.toUser}" },
        "id": { "S": "${util.autoId()}" }
    },
    "attributeValues": {
        "fromUser": { "S": "${context.identity.username}" },
        "content": { "S": "${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:

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

```bash
#if(${context.identity.username} != ${context.arguments.toUser})
    $utils.unauthorized()
#elif(! ${context.result.isAuthorizedForSubscriptions})
    $utils.unauthorized()
#else
    ##User is authorized, but we return null to continue
    null
#end
```

In this case, you're doing two authorization checks. The first ensures that the user isn't subscribing to messages that are meant for another person. The second 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.
Resolver Mapping Template Reference

Topics
- Resolver Mapping Template Overview (p. 182)
- Resolver Mapping Template Programming Guide (p. 184)
- Resolver Mapping Template Context Reference (p. 195)
- Resolver Mapping Template Reference for DynamoDB (p. 209)
- Resolver Mapping Template Reference for Elasticsearch (p. 238)
- Resolver Mapping Template Reference for Lambda (p. 241)
- Resolver Mapping Template Reference for None Data Source (p. 244)

Resolver Mapping Template Overview

AWS AppSync lets you respond to GraphQL operations by enabling you to perform operations on your AWS resources. For each data source, a GraphQL resolver must run and be able to communicate with that data source appropriately.

Usually, the communication is through parameters or operations that are unique to the data source. For an AWS Lambda resolver, you need to specify the payload. For an Amazon DynamoDB resolver, you need to specify a key. For an Amazon Elasticsearch Service resolver, you need to specify an index and the query operation.

Mapping templates are a way of indicating to 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 main types of mapping templates:
- Request templates: Take the incoming request after a GraphQL operation is parsed and convert it into instructions for the resolver so that the resolver can call your data source.
- Response templates: Interpret responses from your data source and translate into a GraphQL response, optionally performing some logic or formatting first.
Example Template

For example, suppose you have a DynamoDB data source and a resolver on a field named `getPost(id:ID!)` that returns a Post type with the following GraphQL query:

```graphql
getPost(id:1){
  id
  title
  content
}
```

Your resolver template might look like the following:

```json
{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "id" : { "S" : "${context.arguments.id}" }
  }
}
```

This would substitute the `id` input parameter value of `1` for `context.arguments.id` and generate the following JSON:

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

```json
{
  "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:

```json
{
  "id" : "${context.data.id},
  "title" : "${context.data.theTitle},
  "content" : "${context.data.theContent-part1} ${context.data.theContent-part2}"
}
```

Here's how the data is shaped after the template is applied to the data:

```json
{
  "id" : 1,
  "title" : "AWS AppSync works offline!",
  "content" : "It also has realtime functionality using GraphQL"
}
```
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 logicful 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. All of these things are 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 always write Lambda functions in your 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
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:

```vql
#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 would be supRESSED:
You saw one example earlier of creating a variable and simultaneously setting values. You could also do this in two steps by adding data to your map:

```
#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 WILL NOT 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 will add 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 = [])

util.quiet($util.put("id", "first value")
```

**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:

```java
#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:

```java
#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:

```java
#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, $i.toString() + "foo"))
##Reference a variable in a string with "${varname}"
$util.qr($myMap.put($i, "${i}foo")
#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:

```vql
#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:

```vql
#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 `<math>arr2[idx]</math>`. You can look up by name from a Map/dictionary in a similar way:

```vql
#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:

```vql
#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:
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:
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:

```vtl
#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:

```vtl
#set($myMap.owner = "Mary") and default ownership
#set($myMap.defaultOwners = ["Admins", "Editors"])
```

**Context**

Now that you are more familiar with performing logical checks in AWS AWS AppSync resolvers with VTL, take a look at the context object:

```vtl
$util.qr($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.

**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:

```vue
$util.toJson($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:
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 `elsifCheck` value when learning about conditionals:

```
#set($data = {
   "id" : "456",
   "meta" : "Valid Response"
})
$util.toJson($data)

#foreach($item in $context.result.entrySet())
   #if($item.key == "elsifCheck")
      $util.qr($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)
   $utils.toJson($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 section.

**Appendix - Template Sample**

If you followed along with the tutorial, you may have built out this template step by step. However, but we also include it below to copy/paste for your 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
Filtering

```java
#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", 
    
#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($range = [0..5])
#foreach($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))
#end

##Operators doesn'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($hashmap = {
   "DynamoDB" : "https://aws.amazon.com/dynamodb/",
   "DynamoDB2" : "https://aws.amazon.com/dynamodb/",
   "Amplify2" : "https://github.com/aws/aws-amplify"
})
#foreach ($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"
})
Resolver Mapping Template Context Reference

AWS AppSync defines a set of variables and functions for working with resolver mapping templates to make logicfull operations on data easier with GraphQL. This document describes those functions and provides examples for working with templates.

Accessing the $context

The $context variable holds all of the contextual information for your resolver invocation. It has the following structure:

```json
{
  "arguments" : { ... },
  "source" : { ... },
  "result" : { ... },
  "identity" : { ... }
}
```

Each field is defined as follows:

**arguments**

A map containing all GraphQL arguments for this field.

**identity**

An object containing information about the caller. See Identity (p. 196) for more information on the structure of this field.

**source**

A map containing the resolution of the parent field.

**result**

A map containing the results of this resolver. This map is only available to response mapping templates.

For example, if you are resolving the author field of the following query:

```/graphql
query {
  getPost(id: 1234) {
    postId
    title
    } ...
```
Then the full `$context` variable that is available when processing a response mapping template might be:

```json
{
  "arguments" : {},
  "source" : {
    "createdAt" : "2017-02-28T18:12:37Z",
    "title" : "A new post",
    "content" : "A long time ago, in a thread far far away",
    "postId" : "1234",
    "authorId" : "34521"
  },
  "result" : {
    "name" : "Steve",
    "joinDate" : "2017-02-28T18:12:37Z",
    "id" : "34521"
  },
  "identity" : {
    "sourceIp" : ["x.x.x.x"],
    "userArn" : "arn:aws:iam::123456789012:user/appsync",
    "accountId" : "123456789012",
    "user" : "AIDAAAAAAAAAAAAAAAAAA"
  }
}
```

**Identity**

The `identity` section contains information about the caller. The shape of this section depends on the authorization type of your AWS AppSync API.

See Security (p. 167) for more information about this section and how it can be used.

**API_KEY authorization**

The `identity` field is not populated.

**AWS_IAM authorization**

The `identity` has the following shape:

```json
{
  "accountId" = "string",
  "cognitoIdentityPoolId" = "string",
  "cognitoIdentityId" = "string",
  "sourceIp" = ["string"],
  "username" = "string", // IAM user principal
  "userArn" = "string"
}
```

**AMAZON_COGNITO_USER_POOLS authorization**

The `identity` has the following shape:

```json
{
```
"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 the user has.

**cognitoIdentityId**

The Amazon Cognito identity ID of the caller.

**cognitoIdentityPoolId**

The Amazon Cognito identity pool ID associated with the caller.

**defaultAuthStrategy**

The default auth strategy for this caller (ALLOW or DENY).

**issuer**

The token issuer.

**sourceIp**

The source IP address of the caller received by AppSync. If the request does not include the x-forwarded-for header then the source ip value contains only a single IP Address from the TCP Connection. If the request includes a x-forwarded-for header then the source IP will be 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 IAM user ARN.

**username**

The username of the authenticated user. In case of AMAZON_COGNITO_USER_POOLS authorization, the value of username is the value of attribute cognito:username. In case of AWS_IAM authorization, the value of the username is the value of the AWS User Principal. We recommend that you use cognitoIdentityId if you are using AWS IAM authorization with credentials vended from Amazon Cognito Federeated Identities.

**Access Request Headers**

AWS AppSync supports passing custom headers from clients and accessing them in your GraphQL resolvers using $context.request.headers. You can then use the header values for actions like
inserting data to a data source or even authorization checks. Single or multiple request headers can use
used as shown in the following examples using $curl with an API key from the command line:

**Single Header Example**

Suppose you set a header of `custom` with a value of `nadia` like so:

```
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": { "S": "$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 was set with two values:

```
```

You could then access these as an array, such as `$context.request.headers.custom[1]`.

**Utility Helpers in $util**

The `$util` variable contains general utility methods that make it easier to work with data.

Unless otherwise specified, all utilities use the UTF-8 character set.

- `$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.unauthorized()

Throws Unauthorized for the field being resolved. This can be used in request or response mapping templates to decide if the caller should be allowed to resolve the field.

$util.error(String)

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.

$util.error(String, String)

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 can be specified.

$util.error(String, String, 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 and a data 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.

$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.appendError(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.appendError(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: errorInfo will NOT be filtered based on the query selection set.
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.

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

```java
#set( $nowEpochMillis = 1517943695758 )
$util.time.epochMillisToSeconds($nowEpochMillis) : 1517943695
$util.time.epochMillisToISO8601($nowEpochMillis) : 2018-02-06T19:01:35.758Z
$util.time.epochMillisToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ") : 2018-02-06 19:01:35+0000
$util.time.epochMillisToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ", "+08:00") : 2018-02-07 03:01:35+0800
$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.nowEpochMillis() : long
    Returns the number of milliseconds from the epoch of 1970-01-01T00:00:00Z to now.
```

Conversion Examples

```java
#set( $nowEpochMillis = 1517943695758 )
$util.time.epochMillisToSeconds($nowEpochMillis) : 1517943695
$util.time.epochMillisToISO8601($nowEpochMillis) : 2018-02-06T19:01:35.758Z
$util.time.epochMillisToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ") : 2018-02-06 19:01:35+0000
$util.time.epochMillisToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ", "+08:00") : 2018-02-07 03:01:35+0800
```

Parsing Examples

```java
$util.time.parseISO8601ToEpochMillis("2018-02-01T17:21:05.180+08:00") : 1517476865180
$util.time.parseFormattedToEpochMillis("2018-02-02 01:19:22+0800", "yyyy-MM-dd HH:mm:ssZ") : 1517505562000
$util.time.parseFormattedToEpochMillis("2018-02-02 01:19:22", "yyyy-MM-dd HH:mm:ss", "+08:00") : 1517505562000
```
$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.parseFormattedToEpochMillis(String, String) : Long

 Parses a timestamp passed as a String, along with a format, and return the timestamp as milliseconds since epoch.

$util.time.parseFormattedToEpochMillis(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.parseISO8601ToEpochMillis(String) : Long

 Parses an ISO8601 timestamp, passed as a String, and return the timestamp as milliseconds since epoch.

$util.time.epochMillisToSeconds(long) : long

Converts an epoch milliseconds timestamp to an epoch seconds timestamp.

$util.time.epochMillisToISO8601(long) : String

Converts a epoch milliseconds timestamp to an ISO8601 timestamp.

$util.time.epochMillisToFormatted(long, String) : String

Converts a epoch milliseconds timestamp, passed as long, to a timestamp formatted according to the supplied format in UTC.

$util.time.epochMillisToFormatted(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.

Map Helpers in $util.map

$util.map contains methods to help with common List operations, such as removing or retaining items from a list for filtering use cases.
$util.map.copyAndRetainAllKeys(Map, Map) : Map

Makes a shallow copy of the first map, retaining only the keys specified in the second map, if they are present. All other keys will be removed from the copy.

$util.map.copyAndRemoveAllKeys(Map, Map) : Map

Makes a shallow copy of the first map, removing any entries where the key is specified in the second map, 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:

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

```json
{
  "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 version: a version that returns an object (eg., $util.dynamodb.toString(...)), and a version that returns the object as a JSON string (eg., $util.dynamodb.toStringJson(...)). In the previous example, we used the version that returns the data as a JSON string. If you wanted to manipulate the object before being used in the template, you can choose to return an object instead:

```json
{}
```
"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.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 list 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()  

206
DynamoDB helpers in $util.dynamodb

$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").
#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.
Resolver Mapping Template Reference for DynamoDB

The AWS AppSyncDynamoDB 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. 209)
- PutItem (p. 210)
- UpdateItem (p. 212)
- DeleteItem (p. 216)
- Query (p. 217)
- Scan (p. 220)
- Type System (Request Mapping) (p. 223)
- Type System (Response Mapping) (p. 226)
- Filters (p. 229)
- Condition Expressions (p. 230)

GetItem

The GetItem request mapping document lets you tell the AWS AppSyncDynamoDB resolver to make a GetItem request to DynamoDB, and allows 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:

```json
{
    "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. Only 2017-02-28 is supported. This value is required.
operation

The DynamoDB operation to perform. To perform the GetItemDynamoDB 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 223). 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. 226).

For more information about response mapping templates, see Resolver Mapping Template Overview (p. 182).

Example

Following is a mapping template for a GraphQL query `getThing(foo: String!, bar: String!)`:

```json
{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "foo" : { "S" : "${context.arguments.foo}" },
    "bar" : { "S" : "${context.arguments.bar}" }
  },
  "consistentRead" : true
}
```

See the DynamoDB API documentation for more information about the DynamoDB GetItem API.

PutItem

The PutItem request mapping document lets you tell the AWS AppSyncDynamoDB resolver to make a PutItem request to DynamoDB, and allows 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:

```json
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "attributeValues" : {
```
The fields are defined as follows:

**version**

The template definition version. Only 2017-02-28 is supported. This value is required.

**operation**

The DynamoDB operation to perform. To perform the PutItemDynamoDB 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 223). This value is required.

**attributeValues**

The rest of the attributes of the item to be put into DynamoDB. For more information on how to specify a "typed value", see Type System (Request Mapping) (p. 223). 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 will overwrite any existing entry for that item. For more information on conditions, see Condition Expressions (p. 230). This value is optional.

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. 226).

For more information about response mapping templates, see Resolver Mapping Template Overview (p. 182).

**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 will be created. If an item already exists with the specified key, it will be overwritten.

```json
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "foo" : { "S" : "${context.arguments.foo}" },
        "bar" : { "S" : "${context.arguments.bar}" } },
    "attributeValues" : {
        "name" : { "S" : "${context.arguments.name}" },
        "version" : { "N" : ${context.arguments.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`.

```json
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "foo" : { "S" : "${context.arguments.foo}" },
    "bar" : { "S" : "${context.arguments.bar}" }
  },
  "attributeValues" : {
    "name" : { "S" : "${context.arguments.name}" },
    #set( $newVersion = $context.arguments.expectedVersion + 1 )
    "version" : { "N" : ${newVersion} }
  },
  "condition" : {
    "expression" : "version = :expectedVersion",
    "expressionValues" : {
      ":expectedVersion" : { "N" : ${context.arguments.expectedVersion} }
    }
  }
}
```

See the DynamoDB API documentation for more information about the DynamoDB `PutItem` API.

**UpdateItem**

The `UpdateItem` request mapping document lets you tell the AWS AppSyncDynamoDB resolver to make an `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:

```json
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key": {
    "foo" : "... typed value,  
    "bar" : "... typed value 
  },
  "update" : {
    "expression" : "someExpression"
    "expressionNames" : {
      "#foo" : "foo"
    },
    "expressionValues" : {
      ":bar" : "... typed value 
    }
  }
}
```
"condition" : {
    ...
    
}

The fields are defined as follows:

**version**

The template definition version. Only 2017-02-28 is 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 223). This value is required.

**update**

The update section lets you specify an update expression that describes how to update the item in DynamoDB. See the DynamoDB UpdateExpressions documentation for more information on how to write update expressions. 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 223). 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 will update any existing entry regardless of its current state. For more information on conditions, see Condition Expressions (p. 230). This value is optional.

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. 226).

For more information about response mapping templates, see Resolver Mapping Template Overview (p. 182).
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.

```json
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : { "S" : "${context.arguments.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 checks to be sure the item currently in DynamoDB has the `version` field set to `expectedVersion`:

```json
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : { "S" : "${context.arguments.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}' == "") )
        #set( $discard = ${expRemove.add("#$!{entry.key}"}) )
      #end
    #end
  #end
```
$\{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\}\})\}
#endif
#endif
#endif
#endif
#endif
## 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,\" )
#endif
#endif
## 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,\" )
#endif
#endif
## 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,\" )
#endif
#endif
## Finally, write the update expression into the document, along with any expressionNames and expressionValues **
"update" : {
  "expression" : "$\{expression\"
  #if( !${expNames.isEmpty()} )
  "expressionNames" : $\{expNames\}
  #endif
  #if( !${expValues.isEmpty()} )
  "expressionValues" : $\{expValues\}
  #endif
},
"condition" : {
  "expression" : "version = :expectedVersion",
  "expressionValues" : {

The DeleteItem request mapping document lets you tell the AWS AppSyncDynamoDB resolver to make a DeleteItem request to DynamoDB, and allows 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": "2017-02-28",
   "operation": "DeleteItem",
   "key": {
      "foo": ... typed value,
      "bar": ... typed value
   },
   "condition": {
      ...
    }
}
```

The fields are defined as follows:

- **version**
  - The template definition version. Only 2017-02-28 is supported. This value is required.

- **operation**
  - The DynamoDB operation to perform. To perform the DeleteItemDynamoDB 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 223). 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 will delete an item regardless of its current state. For more information on conditions, see Condition Expressions (p. 230). This value is optional.

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. 226).

For more information about response mapping templates, see Resolver Mapping Template Overview (p. 182).
Example 1

Following is a mapping template for a GraphQL mutation `deleteItem(id: ID!)`. If an item exists with this ID, it will be deleted.

```
{
  "version" : "2017-02-28",
  "operation" : "DeleteItem",
  "key" : {
    "id" : { "S" : "${context.arguments.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 will be deleted, but only if its `version` field set to `expectedVersion`:

```
{
  "version" : "2017-02-28",
  "operation" : "DeleteItem",
  "key" : {
    "id" : { "S" : "${context.arguments.id}" }
  },
  "condition" : {
    "expression"       : "attribute_not_exists(id) OR version = :expectedVersion",
    "expressionValues" : {
      ":expectedVersion" : { "N" : ${context.arguments.expectedVersion} }
    }
  }
}
```

See the [DynamoDB API documentation](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/API_DeleteItem.html) for more information about the DynamoDB `DeleteItem` API.

Query

The Query request mapping document lets you tell the AWS AppSync DynamoDB resolver to make a Query request to DynamoDB, and allows 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",
  }
}
```
The fields are defined as follows:

version

The template definition version. Only 2017-02-28 is 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. See the DynamoDB KeyConditions documentation for more information on how to write key condition expressions. This section must be specified.

type expression

The query expression. This field must be specified.

type 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.

type 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 223). 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 on filters, see Filters (p. 229). 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 will tell DynamoDB to query the specified index. If omitted, the primary key index will be 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 results to fetch at a single time. 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 AppSyncDynamoDB resolver will only return 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, then for each matching item in the index DynamoDB will fetch the entire item from the parent table. 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. 226).

For more information about response mapping templates, see Resolver Mapping Template Overview (p. 182).

The results have the following structure:

```json
{
  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 will contain a pagination token that can be used in another request. Note that AWS AppSync will encrypt and obfuscate the pagination token returned
from DynamoDB. This is so data from your tables are not 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.

```json
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "query" { 
        "expression" : "ownerId = :ownerId",
        "expressionValues" : {
            "ownerId" : { "S" : "${context.arguments.owner}" } 
        }
    },
    "index" : "owner-index"
}
```

See the [DynamoDB API documentation](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/introduction.html) for more information about the DynamoDB Query API.

**Scan**

The **Scan** request mapping document lets you tell the AWS AppSyncDynamoDB resolver to make a **Scan** request to DynamoDB, and allows 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:

```json
{
    "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. Only 2017-02-28 is supported. This value is required.

**operation**

The DynamoDB operation to perform. To perform the ScanDynamoDB operation, this must be set to Scan. This value is required.

**filter**

An filter that can be used to filter the results from DynamoDB before they are returned. For more information on filters, see Filters (p. 229). 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, then the primary key index will be queried.

**limit**

The maximum number of results to fetch at a single time. This field is optional.

**consistentRead**

A boolean indicating 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 AppSyncDynamoDB resolver will only return 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, then for each matching item in the index DynamoDB will fetch the entire item from the parent table. If the index is configured to project all item attributes, then 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. 226).

For more information about response mapping templates, see Resolver Mapping Template Overview (p. 182).

The results have the following structure:

```json
{
    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 will contain a pagination token that can be used in another request. Note that AWS AppSync will encrypt and obfuscate the pagination token returned from DynamoDB. This is so data from your tables are not inadvertently leaked to the caller. Also note that these pagination tokens cannot 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.

```json
{
    "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.

```json
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "filter" : {
        "expression" : "begins_with(title, :title)",
        "expressionValues" : {
            ":title" : { "S" : "${context.arguments.title}" }
        }
    }
}
```

See the DynamoDB API documentation for more information about the DynamoDB Scan API.
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 \texttt{S} denotes that the value is a string, and the value \texttt{identifier} is the string value itself.

\begin{verbatim}
{ "S" : "identifier" }
\end{verbatim}

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 will not be parsed.

A DynamoDB value is used anywhere in a request mapping document where you need to specify a value. Some places where you will need to do this include: key and attributeValue sections, and the expressionValues section of expression sections. In the following example, the DynamoDB String value \texttt{identifier} is being assigned to the \texttt{id} field in a key section (perhaps in a GetItem request mapping document).

\begin{verbatim}
"key" : {
  "id" : { "S" : "identifier" }
}
\end{verbatim}

**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:

\begin{verbatim}
{ "S" : "some string" }
\end{verbatim}

An example usage is:

\begin{verbatim}
"key" : {
  "id" : { "S" : "some string" }
}
\end{verbatim}

**String set type SS**

A set of string values. A DynamoDB String Set value is denoted by:

\begin{verbatim}
{ "SS" : [ "first value", "second value", ... ] }
\end{verbatim}

An example usage is:

\begin{verbatim}
"attributeValues" : {
  "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
\end{verbatim}
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 will decode 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" : {
    "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

Binary set type BS
A set of binary values. A DynamoDB Binary Set value is denoted by:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=" , "SG93IGFyZSBb3U/Cg==" ... ] }
```

Note that the value is actually a string, where the string is the Base64 encoded representation of the binary data. AWS AppSync will decode 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:

```json
{ "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:

```json
{ "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:

```json
{ "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:

```json
{ "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:

```json
{ "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:
An example usage is:

```json
"attributeValues" : {
    "phoneNumbers" : { "NULL" : null }
}
```

See the DynamoDB documentation for more information on each type.

## 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:

```json
{
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

Then the AWS AppSyncDynamoDB resolver converts it into GraphQL and JSON types as:

```json
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

This section explains how AWS AppSync will convert the following DynamoDB scalar, document and set types:

### String type S

A single string value. A DynamoDB String value will be returned simply as a string.

For example, if DynamoDB returned the following DynamoDB String value:

```json
{ "S" : "some string" }
```

AWS AppSync will convert it to a string:

"some string"

### String set type SS

A set of string values. A DynamoDB String Set value will be returned as a list of strings.

For example, if DynamoDB returned the following DynamoDB String Set value:

```json
{ "SS" : [ "first value", "second value", ... ] }
```
AWS AppSync will convert it to a list of strings:

```
[ "+1 555 123 4567", "+1 555 234 5678"
```

Number type \texttt{N}

A single numeric value. A DynamoDB Number value will be returned as a number.

For example, if DynamoDB returned the following DynamoDB Number value:

```
{ "N" : 1234 }
```

AWS AppSync will convert it to a number:

```
1234
```

Number set type \texttt{NS}

A set of number values. A DynamoDB Number Set value will be 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 will convert it to a list of numbers:

```
[ 67.8, 12.2, 70 ]
```

Binary type \texttt{B}

A binary value. A DynamoDB Binary value will be 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 will convert 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 \texttt{BS}

A set of binary values. A DynamoDB Binary Set value will be 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="", "SG93IGFyZSBsb3I/" ] }
```

AWS AppSync will convert it to a list of strings containing the Base64 representation of the values:

```
[ "SGVsbG8sIFdvcmxkIQo="", "SG93IGFyZSBsb3I/" ]
```
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 will be returned as a boolean.

For example, if DynamoDB returned the following DynamoDB Boolean value:

```json
{ "BOOL" : true }
```

AWS AppSync will convert it to a boolean:

```json
true
```

**List type `L`**

A list of any other supported DynamoDB value. A DynamoDB List value will be returned as a list of values, where each inner value is also converted.

For example, if DynamoDB returned the following DynamoDB List value:

```json
{ "L" : [
    { "S" : "A string value" },
    { "N" : 1 },
    { "SS" : [ "Another string value", "Even more string values!" ] }
  ]
}
```

AWS AppSync will convert it to a list of converted values:

```json
[ "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 will be returned as a JSON object, where each key/value is also converted.

For example, if DynamoDB returned the following DynamoDB Map value:

```json
{ "M" : {
    "someString" : { "S" : "A string value" },
    "someNumber" : { "N" : 1 },
    "stringSet" : { "SS" : [ "Another string value", "Even more string values!" ] }
  }
}
```

AWS AppSync will convert it to a JSON object:

```json
{
    "someString" : "A string value",
    "someNumber" : 1,
    "stringSet" : [ "Another string value", "Even more string values!" ]
}
```

**Null type `NULL`**

A null value.
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. See the DynamoDB QueryFilter and DynamoDB ScanFilter documentation for more information on how to write filter expressions. 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 on how to specify a "typed value", see Type System (Request Mapping) (p. 223). 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" : {
      ":value" : "title"
   },
}
```
Condition Expressions

When you mutate objects in DynamoDB by using the PutItem, UpdateItem, and DeleteItem 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 AppSyncDynamoDB 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 does not have a condition expression, so it will put an item in DynamoDB even if an item with the same key already exists, overwriting the existing item.

```json
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : { "S" : "1" }
    }
}
```

Example 2

The following PutItem mapping document does have a condition expression that will only let the operation succeed if an item with the same key does not exist in DynamoDB.

```json
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : { "S" : "1" }
    },
    "condition" : {
        "expression" : "attribute_not_exists(id)"
    }
}
```

By default, if the condition check fails, then the AWS AppSyncDynamoDB resolver will return 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 AppSyncDynamoDB resolver offers some additional features to help developers handle some common edge cases:

- If AWS AppSyncDynamoDB resolver can determine that the current value in DynamoDB matches the desired result, then it will treat 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 AppSyncDynamoDB resolver should handle the failure.

These will be described in greater detail in the Handling a Condition Check Failure (p. 232) section.
See the DynamoDB ConditionExpressions documentation for more information about DynamoDB conditions expressions.

**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. See the DynamoDB ConditionExpressions documentation for more information about how to write condition expressions. 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 on 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 AppSyncDynamoDB resolver how to handle a condition check failure:

**equalsIgnore**

When a condition check fails when using the PutItem operation, the AWS AppSyncDynamoDB resolver will compare the item currently in DynamoDB against the item it tried to write. If they are the same, then it will treat 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, then treat the operation as it if succeeded. This field is optional.
**consistentRead**

When a condition check fails, AWS AppSync will get the current value of the item from DynamoDB using a strongly consistent read. You can use this field to tell the AWS AppSyncDynamoDB 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 AppSyncDynamoDB resolver will treat 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 AppSyncDynamoDB resolver will take after it has compared the current value in DynamoDB against the expected result. This field is required, and has two possible values:

- **Reject**
  
  The mutation will fail, 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 AppSyncDynamoDB resolver will invoke 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 to decide how the AWS AppSyncDynamoDB resolver should handle the condition check failure. This field must only be specified when `strategy` is set to `Custom`. See Handling a Condition Check Failure (p. 232) for more information about how to use this feature.

**Handling a Condition Check Failure**

By default, when a condition check fails, the AWS AppSyncDynamoDB resolver will return 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 AppSyncDynamoDB resolver offers some additional features to help developers handle some common edge cases:

- If AWS AppSyncDynamoDB resolver can determine that the current value in DynamoDB matches the desired result, then it will treat 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 AppSyncDynamoDB resolver should handle the failure.

The flowchart for this process is:

**Checking for the Desired Result**

When the condition check fails, the AWS AppSyncDynamoDB resolver will perform a `GetItemDynamoDB` request to get the current value of the item from DynamoDB. By default, it will use 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 AppSyncDynamoDB resolver will compare 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, then it will treat the operation as successful and return the item that was retrieved from DynamoDB. Otherwise, it will then follow the configured strategy.

For example, if the PutItem request mapping document looked like this:

```json
{
  "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 this:

```json
{
  "id" : { "S" : "1" },
  "name" : { "S" : "Steve" },
  "version" : { "N" : 8 }
}
```

Then the AWS AppSyncDynamoDB 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 AppSyncDynamoDB resolver will see if an item was returned from DynamoDB. If no item was returned, it will treat the operation as successful. Otherwise, it will follow the configured strategy.
- For the UpdateItem operation, the AWS AppSyncDynamoDB 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, then the AWS AppSyncDynamoDB resolver will follow the configured strategy, to either reject the mutation or invoke a Lambda function to decide what to do next.

**Following the "Reject" Strategy**

When following the Reject strategy, the AWS AppSyncDynamoDB resolver will return an error for the mutation, and the current value of the object in DynamoDB will also be returned in a `data` field in the `error` section of the GraphQL response. The item returned from DynamoDB will be put through the response mapping template to translate it into a format the client expects, and will also be filtered by the selection set.

For example, given the following mutation request:

```graphql
mutation {

```
updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
    Name
    theVersion
}

If the item returned from DynamoDB looks like:

```json
{
    "id": { "S": "1" },
    "name": { "S": "Steve" },
    "version": { "N": 8 }
}
```

and the response mapping template looks like:

```json
{
    "id": "${context.result.id}",
    "Name": "${context.result.name}",
    "theVersion": ${context.result.version}
}
```

then the GraphQL response will look like:

```json
{
    "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 note that if any fields in the returned object would have been filled by other resolvers if the mutation had succeeded, they will not be resolved when the object is returned in the error section.

### Following the "Custom" Strategy

When following the Custom strategy, the AWS AppSyncDynamoDB resolver will invoke a Lambda function to decide what to do next. The Lambda function has three options to choose from:

- **reject** the mutation. This will tell the AWS AppSyncDynamoDB 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. This will tell the AWS AppSyncDynamoDB resolver to silently ignore the condition check failure, and just return the value in DynamoDB.
- **retry** the mutation. This will tell the AWS AppSyncDynamoDB resolver to retry the mutation with a new request mapping document.

### The Lambda invocation request
The AWS AppSyncDynamoDB resolver will invoke the Lambda function specified in the `lambdaArn`. It will use the same `service-role-arn` configured on the data source. The payload of the invocation has the following structure:

```json
{
    "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:

```json
{
    "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": {
                "version": { "N": 1 }
            },
            "equalsIgnore": [ "version" ]
        }
    },
    "currentValue": {
        "id": { "S": "1" }
    }
}
```
The Lambda Invocation Response

The Lambda function can inspect the invocation payload and apply any business logic to decide how the AWS AppSyncDynamoDB 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:

  ```json
  {
    "action": "reject"
  }
  ```

  This will tell the AWS AppSyncDynamoDB 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:

  ```json
  {
    "action": "discard"
  }
  ```

  This will tell the AWS AppSyncDynamoDB resolver to silently ignore the condition check failure, and just return the value in DynamoDB.

- **retry** the mutation. The response payload for this option must have this structure:

  ```json
  {
    "action": "retry",
    "retryMapping": { "..." }
  }
  ```

  This will tell the AWS AppSyncDynamoDB 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. See PutItem (p. 210) for a description of the attributeValues field.

  ```json
  {
    "attributeValues": { "..." },
    "condition": {
  ```
For **UpdateItem**, the retryMapping section has the following structure. See **UpdateItem** (p. 212) for a description of the update section.

```json
{
  "update": {
    "expression": "someExpression",
    "expressionNames": {
      "#foo": "foo"
    },
    "expressionValues": {
      "#:bar": ... typed value
    },
    "condition": {
      "consistentRead": true
    }
  }
}
```

For **DeleteItem**, the retryMapping section has the following structure.

```json
{
  "condition": {
    "consistentRead": true
  }
}
```

Note that there is no way to specify a different operation or key to work on: the AWS AppSyncDynamoDB resolver will only allow retries of the same operation on the same object. Also note the condition section doesn't allow a conditionalCheckFailedHandler to be specified. If the retry fails, then the AWS AppSyncDynamoDB resolver will follow 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** then it will retry the request, updating the version and expectedVersion from the item currently in DynamoDB; otherwise it will reject the mutation.

```javascript
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 }
  }
}
```
Resolver Mapping Template Reference for Elasticsearch

The AWS AppSync resolver for Amazon Elasticsearch Service enables you to use GraphQL to store and retrieve data in existing Amazon ES domains in your account. This resolver works by allowing you to map an incoming GraphQL request into an Amazon ES request, and then map the Amazon ES response back to GraphQL. This section describes the mapping templates for the supported Amazon ES operations.

Request Mapping Template

Most Amazon ES request mapping templates have a common structure where just a few pieces change. The following example runs a search against an Amazon ES 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:

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

For more information on query options, see the Elasticsearch Query DSL Reference.

Response Mapping Template

As with other data sources, Amazon ES sends a response to AWS AppSync that needs to be converted to GraphQL. The shape of an Amazon ES response can be seen in the Elasticsearch Request Body Search DSL Reference.
Most GraphQL queries are looking for the _source field from an Amazon ES 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 Amazon ES:

**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 Amazon ES domain. Both the key and the value must be a string.

```
"operation" : "PUT"
```

**path field**

(Request Mapping Template only)

The search path for an Amazon ES 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 Amazon ES domain. For example, the previous example might translate to:

```
GET https://elasticsearch-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" : "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 Amazon ES 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. 197).

**Resolver Mapping Template Reference for Lambda**

The AWS AppSync Lambda resolver mapping templates enable you to shape requests from AWS AppSync to AWS Lambda functions located in your account, and responses from your Lambda functions back to AWS AppSync. Mapping templates also enable you 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 AWS 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.

```json
{
    "version": string,
    "operation": Invoke|BatchInvoke,
    "payload": any type
}
```

Here is the JSON schema representation of the Lambda request mapping template, once resolved.

```json
{
    "definitions": {},
    "$schema": "http://json-schema.org/draft-06/schema#",
    "$id": "http://aws.amazon.com/appsync/request-mapping-template.json",
    "type": "object",
    "properties": {
        "version": {
            "$id": "/properties/version",
            "type": "string",
            "enum": [
                "2017-02-28"
            ],
            "title": "The Mapping template version.",
            "default": "2017-02-28"
        }
    }
}
```
Here is an example where we chose to pass the `field` value, and the GraphQL field arguments from the context.

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

The entire mapping document will be passed as input to your Lambda function, so that the previous example would now look like the following:

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

**operation**

The Lambda data source allows you to define two operations, `Invoke` and `BatchInvoke`. The `Invoke` lets AWS AppSync know to call your Lambda function for every GraphQL field resolver, while `BatchInvoke` instructs AWS AppSync to batch requests for the current GraphQL field.
For **Invoke**, the resolved request mapping template exactly matches the input payload of the Lambda function. So the following sample template:

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

is resolved and passed to the Lambda function, as follows:

```json
{
  "version": "2017-02-28",
  "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 of the resolved mapping template `payload` values into a list under a single object matching the mapping template.

The following example template shows the merge:

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

This template is resolved into the following mapping document:

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

```json
[
  { "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 will wrap 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 needs to be converted to a GraphQL type.

The result of the Lambda function will be set on the context object that is available via the 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:

```$utils.toJson($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 will be set to null accordingly.

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 a 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:
Here is an example where we chose to pass the field arguments via the VTL context property $context.arguments:

```
{
    "version": "2017-02-28",
    "payload": $utils.toJson($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, `version` defines the version used by the template. `version` is required.

Example:

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

**payload**

The `payload` field is a container that can be used to pass any well-formed JSON to the response mapping template. `payload` 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:

```
$utils.toJson($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.
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" : { "S" : "${context.arguments.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:

```json
{
  "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:

```json
"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:

```typescript
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:

```javascript
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.
Monitoring and Logging

Monitoring with Amazon CloudWatch

You monitor and debug requests in AWS AppSync by using Amazon CloudWatch. Using AWS AppSync, you can use GraphQL to request data from the cloud. There are often times when you want to get more information about the execution of your API. To help debug issues related to request execution of your GraphQL API, you can enable Amazon CloudWatch Logs to monitor API calls. Once enabled, AWS AppSync logs API calls in CloudWatch.

Setup and Configuration

You can enable logging on a GraphQL API automatically through the AWS AppSync console.

1. Sign in to the AWS AppSync console.
2. Choose Settings from the navigation panel.
3. Under Logging, click the toggle to Enable Logs.
4. When the console prompts you, provide or create a CloudWatch ARN role.
5. (Optional) Choose to configure the Field resolver log level from the list.
6. Choose Save. The logging is automatically configured for your API.

Manual Role Configuration

To enable CloudWatch Logs, you must grant AWS AppSync correct permissions to write logs to CloudWatch for your account. To do this, you need to provide a service role ARN so that AWS AppSync can assume this role when writing the logs.

First, navigate to the IAM console. Then create a new policy with the name AWSAppSyncPushToCloudWatchLogsPolicy that has the following definition:

```json
{
  "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 above policy to this role. Edit the trust relationship for this role to have the following:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["appsync:*"],
      "Resource": "*"
    }
  ]
}
```
Copy the role ARN and register this with AWS AppSync to enable writing into CloudWatch.

CloudWatch Metrics

You can use CloudWatch metrics to monitor and provide alerts about specific events that can be triggered by HTTP status codes or by latency, such as the overall GraphQL request and response latency. The following are the metrics that are emitted.

4XX The number of errors captured as a result of invalid requests due to incorrect client configuration. Typically, these errors happen anywhere outside of the GraphQL execution. For example, this could be an incorrect JSON payload or an incorrect query in the request, when the service is throttled, or even a potential misconfiguration on the Auth settings.

**Unit**: Count. Use the Sum statistic to get the total occurrences of these errors.

5XX Errors encountered during the execution of a GraphQL query. For example, this could occur when a query request is initiated for an empty or incorrect schema, if the Amazon Cognito user pool ID or AWS Region is invalid. Alternatively, this could also happen if AWS AppSync encounters an issue during an execution 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.

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 enabled, the following information is logged:

- The request and response HTTP headers
- The GraphQL query that is being executed in the request
- The overall execution summary
- New and existing GraphQL subscriptions that are registered
Field-Level Logs

When enabled, 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 enable 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 pertaining to 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** - The following information is logged for all fields in the query:
  - Field-level tracing information
  - The generated request/response functions that got resolved for each field

Benefits of Enabling 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 execution in CloudWatch Logs might look similar to the following:

```json
{
    "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:
In the log results above, **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). **startOffset** gives an offset from the start of the GraphQL query execution. **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 execution to perform poorly.

You can also choose to enable metric filters in CloudWatch. You can then use these metric filters to turn log data into numerical CloudWatch metrics, so you can graph or set an alarm on them.
Auditing

AWS AppSync integrates with AWS CloudTrail, a service that captures API calls made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. CloudTrail captures API calls from the AWS AppSync console or from the AWS AppSync API. Using the information collected by CloudTrail, you can determine what request was made to AWS AppSync, the source IP address from which the request was made, who made the request, when it was made, and so on. To learn more about CloudTrail, including how to configure and enable it, see What Is Amazon CloudTrail in the Amazon CloudTrail User Guide.

AWS AppSync Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, API calls made to AWS AppSync actions are tracked in CloudTrail log files. AWS AppSync records are written together with other AWS service records in a log file. CloudTrail determines when to create and write to a new file based on a time period and file size.

All AWS AppSync actions are logged by CloudTrail and are documented in the AWS AppSync API Reference. For example, calls to the CreateGraphqlApi, CreateDataSource, and ListResolvers actions generate entries in the CloudTrail log files.

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 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.

You can also create a trail and store your log files in your Amazon S3 bucket for as long as you want, and define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted with Amazon S3 server-side encryption (SSE).

To be notified of log file delivery, configure CloudTrail to publish Amazon SNS notifications when new log files are delivered. For more information, see Configuring Amazon SNS Notifications for CloudTrail.

You can also aggregate AWS AppSync log files from multiple AWS regions and multiple AWS accounts into a single Amazon S3 bucket.

For more information, see Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts.

Understanding AWS AppSync Log File Entries

CloudTrail log files can contain one or more log entries. Each entry lists multiple JSON-formatted events. A log entry 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. Log entries are not an ordered stack trace of the public API calls, so they do not appear in any specific order.
Because of potential confidentiality issues, log entries do not contain the synthesized text. Instead, this text is redacted in the log entry.

The following example shows a CloudTrail log entry that demonstrates the `CreateApiKey` action.

```json
{
    "Records": [
        {
            "eventVersion": "1.05",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "A1B2C3D4E5F6G7EXAMPLE",
                "arn": "arn:aws:iam::123456789012:user/Alice",
                "accountId": "123456789012",
                "accessKeyId": "AKIAIOSFODN7EXAMPLE",
                "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": "123456789012"
        }
    ]
}
```

The following example shows a CloudTrail log entry that demonstrates the `ListApiKeys` action.

```json
{
    "Records": [
        {
            "eventVersion": "1.05",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "A1B2C3D4E5F6G7EXAMPLE",
                "arn": "arn:aws:iam::123456789012:user/Alice",
                "accountId": "123456789012",
                "accessKeyId": "AKIAIOSFODN7EXAMPLE",
                "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": 1518037200000
                ]
            },
            "requestID": "99999999-9999-9999-9999-999999999999",
            "eventID": "99999999-9999-9999-9999-999999999999",
            "readOnly": false,
            "eventType": "AwsApiCall",
            "recipientAccountId": "123456789012"
        }
    ]
}
```
The following example shows a CloudTrail log entry that demonstrates the `DeleteApiKey` action.

```
{
    "Records": [{
        "eventVersion": "1.05",
        "userIdentity": {
            "type": "IAMUser",
            "principalId": "A1B2C3D4E5F6G7EXAMPLE",
            "arn": "arn:aws:iam::123456789012:user/Alice",
            "accountId": "123456789012",
            "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": "123456789012"
    }
}]
```