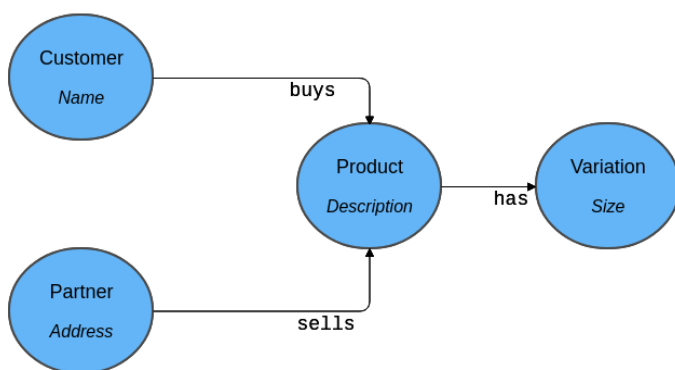


# Implementing a GraphQL API using AWS AppSync

[GraphQL](#) is an exciting API specification that has been around since 2005 and offers a graph based way to build APIs.

Typically a graph defines a group of nodes that are related to each other and have certain attributes:



The graph based approach allows API consumers to navigate through the different nodes and gather only the information they need, as opposed to REST APIs where response bodies are already defined. This implies that GraphQL is helpful when data can be easily represented in a graph, otherwise it might not be the right solution.

## Use case

The aim of this blog post is to demonstrate how to get started with GraphQL using [AWS AppSync](#). Quoting [AWS documentation](#):

*AWS AppSync is a fully managed service that makes it easy to develop GraphQL APIs by handling the heavy lifting of securely connecting to data sources like AWS DynamoDB, Lambda, and more.*

To show you how it works we will implement a small API that creates and queries customer objects stored in a [DynamoDB table](#).

In our use case customers have a first name, last name and an age. We want to be able to create them with an automatically generated ID as well as query the whole collection of customers and single customers.

## Implementation

First of all we need to create an AppSync App:

AppSync

MOBILE SERVICES

# AWS AppSync

Power your applications with the right data, from one or more data sources, at global scale

Create AWS AppSync API

Create APIEstimated 3-5 mins

## Getting Started

AWS AppSync provides you multiple ways to create a GraphQL API. You can customize and build your API from scratch, create a blank API, or even get started using one of our sample APIs.

Customize your API or import from Amazon DynamoDB

Start

Create with wizard

We'll help you build an API from the ground up.

Import DynamoDB Table

You can begin by importing an existing table as a DataSource.

Build from scratch

Start with a blank API and design your own custom schema.

## Create resources

API configuration

API name

Type a name for your AWS AppSync API.

Customers App

CancelCreate

## Schema

Now you're ready to start defining the API's schema:

AWS AppSync > Customers App

## Getting Started

▼ Define the schema

The first step when creating an AppSync API is to design your schema. A schema is made up of object, input, interface, union, enum, and scalar data types that are organized as a graph. While you use the schema to define the shape of your API, you use resolvers to define the functionality. Attach resolvers to any object fields on any object type in your schema, and that resolver will be invoked every time that field is listed in a query or mutation.

Click the button or use the navigation panel on the left to navigate to the schema page. From the schema page, you can click the Create Resources button to quickly add an object type and generate queries, mutations, and subscriptions.

Edit Schema

[Learn more about the schema](#)

Let's start by creating a Customer:

```
type Customer {  
  id: ID!  
  firstName: String!  
  lastName: String!  
  age: Int  
}
```

As you can see, `id`, `firstName` and `lastName` have an exclamation mark (!) following their scalar types - that denotes that they are required fields whereas `age` is not.

Then, we need to specify our queries:

```
type Query {  
  getCustomers: [Customer]  
  getCustomer(id: ID!): Customer  
}
```

This means that `getCustomers` returns a collection of customers and `getCustomer` returns the customer matching the given id.

Creating a customer is done by using a so called *Mutation*:

```

type Mutation {
  createCustomer(firstName: String!, lastName: String!, age: Int): Customer
}

```

Using the `createCustomer` Mutation with the mandatory parameters `firstName` and `lastName` and the optional parameter `age` we can create a new customer.

Finally, both queries and mutations need to be specified in the schema:

```

schema {
  query: Query
  mutation: Mutation
}

```

The complete schema should look like this:

The screenshot shows the AWS AppSync console interface. On the left, the 'Schema' tab is active, displaying a GraphQL schema definition. On the right, the 'Resolvers' tab is active, showing a table of resolvers for the 'Customer' type. A red arrow points to the 'Save Schema' button in the top right corner.

**Schema**

```

1 type Customer {
2   id: ID!
3   firstName: String!
4   lastName: String!
5   age: Int
6 }
7
8 type Mutation {
9   createCustomer(firstName: String!, lastName: String!, age: Int): Customer
10 }
11
12 type Query {
13   getCustomers: [Customer]
14   getCustomer(id: ID!): Customer
15 }
16
17 schema {
18   query: Query
19   mutation: Mutation
20 }

```

**Resolvers**

| Field              | Resolver                |
|--------------------|-------------------------|
| id: ID!            | <button>Attach</button> |
| firstName: String! | <button>Attach</button> |
| lastName: String!  | <button>Attach</button> |
| age: Int           | <button>Attach</button> |

## Data source

GraphQL itself does not provide any storage functionality. Data may come from different sources including traditional databases such as DynamoDB, RDS or Elasticsearch as well as Lambda or HTTP endpoints. We're going to use DynamoDB because it's a flexible easy to use database solution.

First we navigate to the AWS DynamoDB service and create the `Customers` table:

## Create DynamoDB table

Tutorial ?

DynamoDB is a schema-less database that only requires a table name and primary key. The table's primary key is made up of one or two attributes that uniquely identify items, partition the data, and sort data within each partition.

Table name\*  ⓘ

Primary key\* Partition key

String ▼ ⓘ

☐ Add sort key

### Table settings

Default settings provide the fastest way to get started with your table. You can modify these default settings now or after your table has been created.

- ☒ Use default settings
- No secondary indexes.
  - Auto Scaling capacity set to 70% target utilization, at minimum capacity of 5 reads and 5 writes.
  - Encryption at Rest with DEFAULT encryption type.

+ Add tags **NEW!**

Additional charges may apply if you exceed the AWS Free Tier levels for CloudWatch or Simple Notification Service. Advanced alarm settings are available in the CloudWatch management console.

[Cancel](#) [Create](#)

Then we switch back to our AWS AppSync API to add the corresponding data source:

## Data Sources

Connect existing AWS resources to your API. [Info](#)

### Data Sources

[Edit](#) [Delete](#) [Create data source](#)

< 1 > ⚙

Name ▲

Type ▼

Resource ▼

You do not have any Data Sources in this API.

## New Data Source

**Create new Data Source**

Data source name

A name starts with a letter and contains only numbers, letters, and underscores(\_)

Data source type

Select the type of your data source.

Amazon DynamoDB table ▼

Region

Select the region that contains your data source.

EU-CENTRAL-1 ▼

Table name

Select a table from the dropdown.

Customers ▼

[Don't see your table?](#)

Create or use an existing role

Allow AWS AppSync to securely interact with your data source.

☒ New role

☐ Existing role

Versioning [Info](#)

☐ Enable data source versioning

Automatically generate GraphQL

Turning this on will extend your existing schema and automatically configure resolvers.

☐

Cancel

Create

## Resolvers

In order to tell AppSync how to interact with our DynamoDB table we need to define resolvers for each query or mutation in our schema.

Creating a resolver can be done by clicking on *Attach* next to the method name.

## Schema

Design your schema using GraphQL SDL, attach resolvers, and quickly deploy AWS resources. [Info](#)

Create Resources

Undo Edits

Save Schema

Schema

Export schema

```

1 type Customer {
2   id: ID!
3   firstName: String!
4   lastName: String!
5   age: Int
6 }
7
8 type Mutation {
9   createCustomer(firstName: String!, lastName: String!, age: Int): Customer
10 }
11
12 type Query {
13   getCustomers: [Customer]
14   getCustomer(id: ID!): Customer
15 }
16
17 schema {
18   query: Query
19   mutation: Mutation
20 }

```

Resolvers

Filter types...

firstName: String!

Attach

lastName: String!

Attach

age: Int

Attach

Mutation

| Field                         | Resolver |
|-------------------------------|----------|
| createCustomer(...): Customer | Attach   |

Query

| Field                      | Resolver |
|----------------------------|----------|
| getCustomers: [Customer]   | Attach   |
| getCustomer(...): Customer | Attach   |

Afterwards we need to select `Customers` as data source and configure the mapping templates as follows:

## createCustomer

### Request mapping template:

```

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

```

### Response mapping template:

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

This way a new customer will be created using the given parameters and automatically generated Id. As a response we'll receive the created customer.

## getCustomers

Getting all customers can be done by scanning the DynamoDB table:

### Request mapping template:

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

#### Response mapping template:

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

which will return the whole collection of customers.

### getCustomer

Finally, this is how you can get a single customer:

#### Request mapping template:

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

#### Response mapping template:

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

## Testing the API

You can test your API by using the built-in Queries tool:

The screenshot shows the AWS IAM console's 'Queries' tool. The 'Query' dropdown is set to 'API key'. The 'Explorer' sidebar on the left shows a tree view of the API schema. The main editor area displays a GraphQL query. The query is as follows:

```
1 mutation MyMutation {
2   createCustomer(firstName: "Jane", lastName: "Doe") {
3     id
4   }
5 }
6
7 query MyQuery {
8   getCustomers {
9     id
10    firstName
11    age
12    lastName
13  }
14 }
15
```

The JSON response is displayed on the right:

```
{
  "data": {
    "getCustomers": [
      {
        "id": "e1f9e4e9-6b7c-4953-bf58-b6b8918e2454",
        "firstName": "Jane",
        "age": null,
        "lastName": "Doe"
      }
    ]
  }
}
```

In this example we created a new customer called "Jane Doe" and afterwards we queried all customers.

### Authorization



As you saw in the previous screenshot, we're using an API key that is created by AppSync by default to authorize our requests. However, you can choose among multiple authorization options including IAM and OpenID Connect.

## cURL examples

Using the desired authorization method you can now use your API from anywhere you need, e.g. using cURL.

### Creating a customer

```
curl -X POST \
  -H 'Content-Type: application/json' \
  -H 'x-api-key: <api_key>' \
  -d '{"query":"mutation {createCustomer(firstName: \"Jane\", lastName: \"Doe\") {id}}}"' \
  https://<graphql_api_id>.appsync-api.eu-central-1.amazonaws.com/graphql
```

### Retrieving all customers

```
curl -X POST \
  -H 'Content-Type: application/json' \
  -H 'x-api-key: <api_key>' \
  -d '{"query":"{getCustomers {id, firstName, lastName, age}}}"' \
  https://<graphql_api_id>.appsync-api.eu-central-1.amazonaws.com/graphql
```

### Retrieve a customer

```
curl -X POST \
  -H 'Content-Type: application/json' \
  -H 'x-api-key: <api_key>' \
  -d '{"query":"{getCustomer(id: \"<customer_id>\") {id, firstName, lastName, age}}}"' \
  https://<graphql_api_id>.appsync-api.eu-central-1.amazonaws.com/graphql
```

## Conclusion

GraphQL excels when the data you want to expose can be well described as a graph and offers advantages over REST APIs such as more flexibility for consumers to retrieve exactly the information they need.

It's a promising fast growing technology which is already widely used by big tech players such as GitHub, Twitter or Facebook.

As we have seen in this article, there's already good tooling available in the market to deliver and consume GraphQL APIs.

If you want to see a more detailed terraform automated way to implement what is described here you are welcome to check out this GitHub repository: [aws-graphql-demo](#).