

## TITLE: 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:

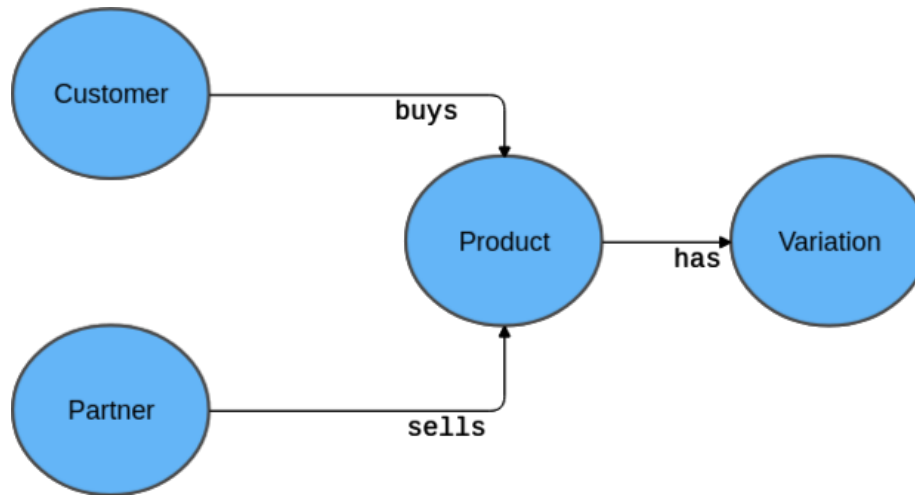


Figure 1: GraphQL Example

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:

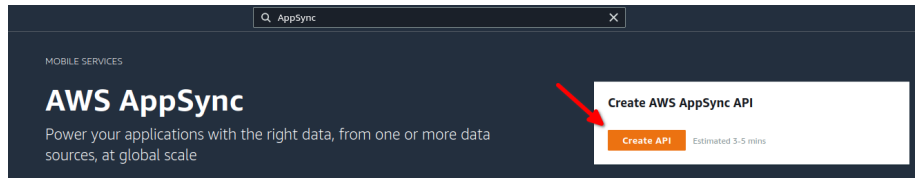


Figure 2: AppSync overview

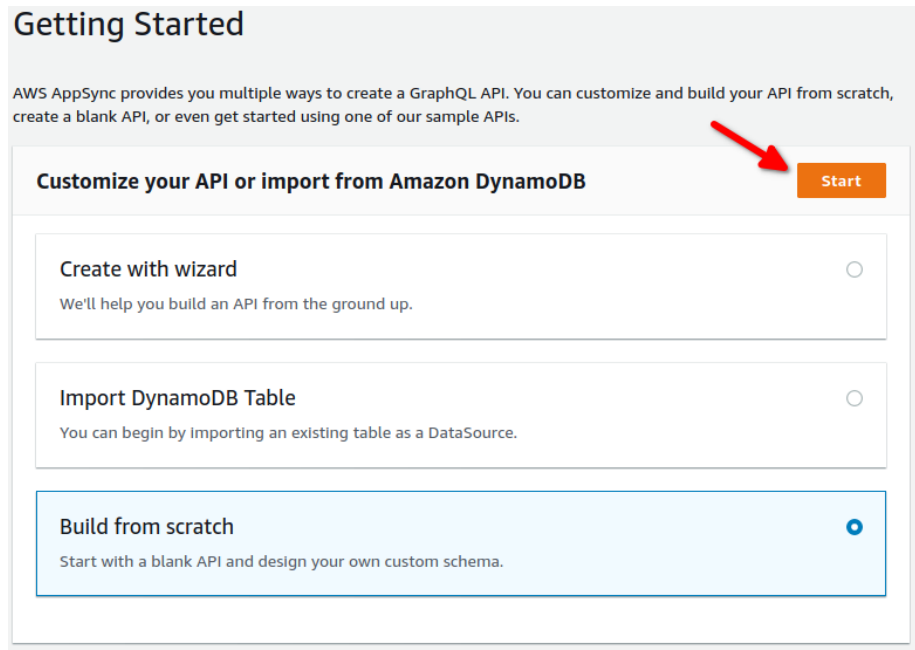


Figure 3: AppSync creation

## Schema

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

Let's start by creating a Customer:

## Create resources

**API configuration**

API name

Type a name for your AWS AppSync API.

Cancel Create




Figure 4: AppSync name

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](#)




Figure 5: Edit schema

```

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 means 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 so called *Mutations*:

```

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:

## 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 will create the `Customers` DynamoDB table:

Then we will add the corresponding data source to AWS App Sync:

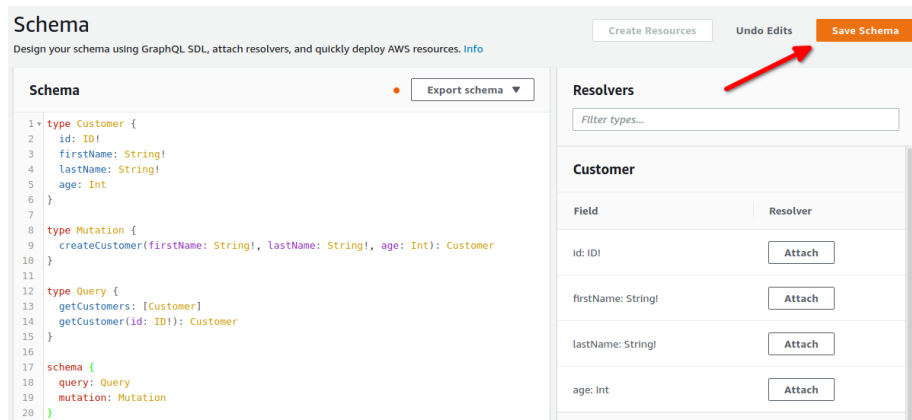


Figure 6: schema

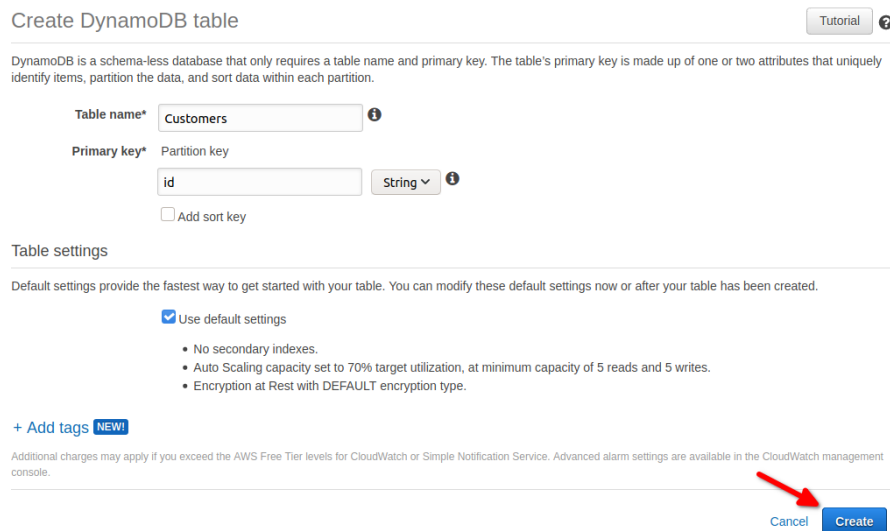


Figure 7: DynamoDB

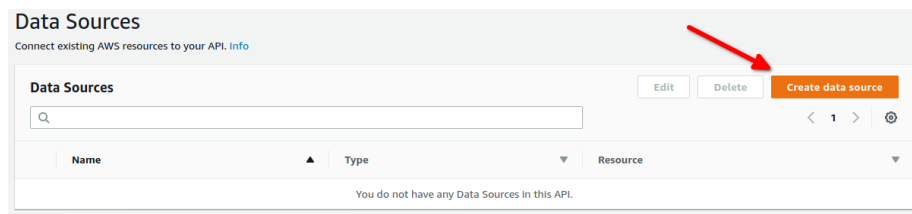


Figure 8: create data source

**New Data Source**

**Create new Data Source**

Data source name  
Customers  
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

Figure 9: data source

## Resolvers

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

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

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)
}
```

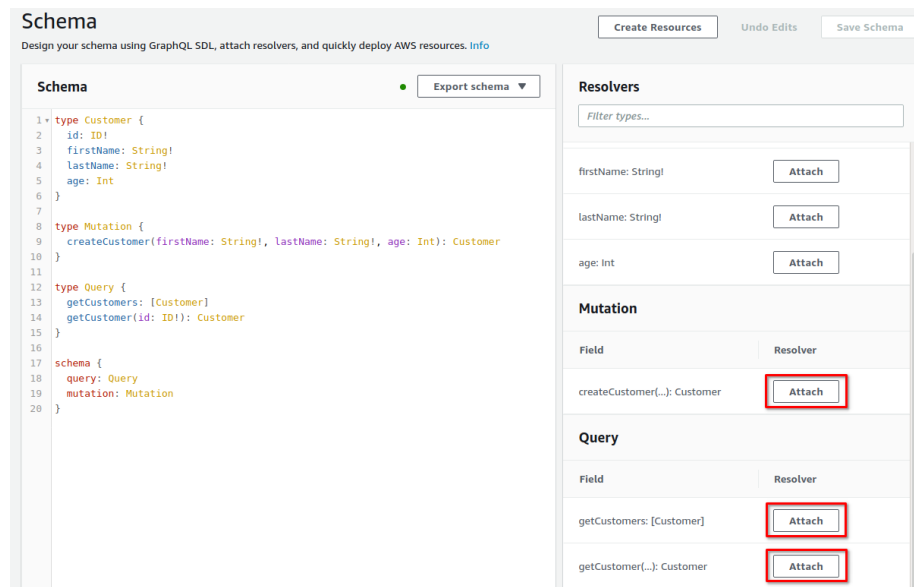


Figure 10: data source

### 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:

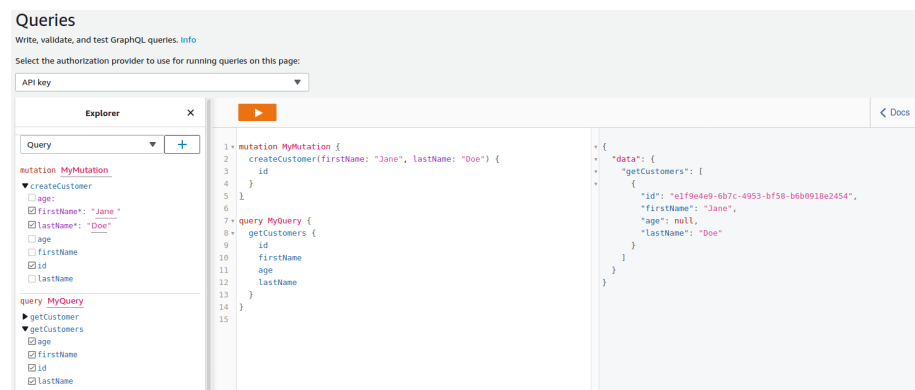


Figure 11: testing

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>.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>.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>.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`.