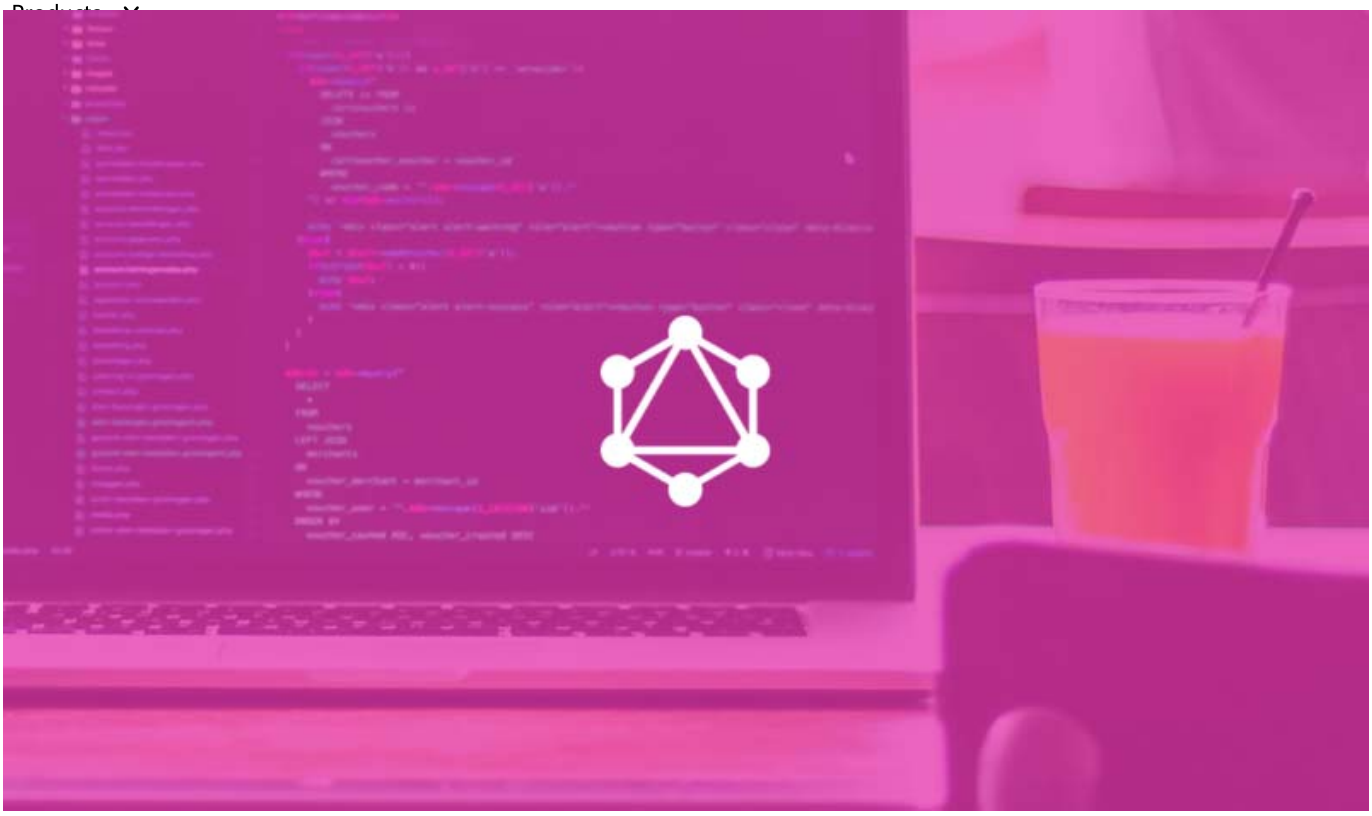
Handling authentication in GraphQL – Part 2: JWT



This is part 2 of a 3 part tutorial. You can find part 1 here and part 3 here.

> In the first part of this series, we looked at an overview of authentication, how it is done in REST and how it can be done in GraphQL. Today, we'll be taking a more practical approach by building a GraphQL server and then add authentication to it.

What we'll be building

To demonstrate things, we'll be building a GraphQL server that has a kind of authentication system. This authentication system will include the ability for users to signup, login and view their profile. The authentication system will make use of JSON Web Tokens (JWT). The GraphQL server will be built using the Express framework.

This tutorial assumes you already have Node.js (at least version 8.0) installed on your computer.

Create a new project

We'll start by creating a new Node.js project. Create a new folder called `graphql-jwt-auth`. Open the newly created folder in the terminal and run the command below:

```
npm init -y
```

? The `-y` flag indicates we are selecting yes to all the `npm init` options and using the defaults.

Then we'll update the `dependencies` section of `package.json` as below:

```
// package.json
```

```
  "dependencies": {
    "apollo-server-express": "^1.3.2",
    "bcrypt": "^1.0.3",
    "body-parser": "^1.18.2",
    "dotenv": "^4.0.0",
    "express": "^4.16.2",
    "express-jwt": "^5.3.0",
    "graphql": "^0.12.3",
    "graphql-tools": "^2.19.0",
    "jsonwebtoken": "^8.1.1",
    "mysql2": "^1.5.1",
    "sequelize": "^4.32.2"
  }
```

These are all the dependencies for our GraphQL server and the authentication system. We'll go over each of them as we begin to use them. Enter the command below to install them:

```
npm install
```

Database setup

For the purpose of this tutorial we'll be using MySQL as our database and Sequelize as our ORM. To make using Sequelize seamless, let's install the Sequelize CLI on our computer. We'll install it globally:

```
npm install -g sequelize-cli
```

?You can also install the CLI locally to the `node_modules` folder with: `npm install sequelize-cli --save`. The rest of the tutorial assumes you installed the CLI globally. If you installed it locally, you will need to run CLI with `./node_modules``/.bin/sequelize`.

Once that's installed, we can use the CLI to initialize Sequelize in our project. In the project's root directory, run the command below:

```
sequelize init
```

The command will create some folders within our project root directory. The `config`, `models` and `migrations` folders are the ones we are concerned with in this tutorial.

The `config` folder contains a `config.json` file. We'll rename this file to `config.js`. Now, open `config/config.js` and paste the snippet below into it:

```
// config/config.js

require('dotenv').config()

const dbDetails = {
  username: process.env.DB_USERNAME,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME,
  host: process.env.DB_HOST,
  dialect: 'mysql'
}

module.exports = {
  development: dbDetails,
  production: dbDetails
}
```

We are using the `dotenv` package to read our database details from a `.env` file. Let's create the `.env` file within the project's root directory and paste the snippet below into it:

```
//.env

NODE_ENV=development
DB_HOST=localhost
DB_USERNAME=root
DB_PASSWORD=
DB_NAME=graphql_jwt_auth
```

Update accordingly with your own database details.

Since we have changed the config file from JSON to JavaScript file, we need to make the Sequelize CLI aware of this. We can do that by creating a `.sequelizerc` file in the project's root directory and pasting the snippet below into it:

```
// .sequelizerc

const path = require('path')

module.exports = {
  config: path.resolve('config', 'config.js')
}
```

Now the CLI will be aware of our changes.

One last thing we need to do is update `models/index.js` to also reference `config/config.js`. Replace the line where the config file is imported with the line below:

```
// models/index.js

var config = require(__dirname + '/../config/config.js')[env]
```

With the setup out of the way, let's create our model and migration. For the purpose of this tutorial, we need only one model which will be the `User` model. We'll also create the corresponding migration file. For this, we'll be using the Sequelize CLI. Run the command below:

```
sequelize model:generate --name User --attributes username:string,email:string,password:string
```

This creates the User model and its attributes/fields: `username`, `email` and `password`. You can learn more on how to use the CLI to create models and migrations in the docs.

A `user.js` file will be created in the `models` folder and a migration file with name like `TIMESTAMP-create-user.js` in the `migrations` folder.

Now, let's run our migration:

```
sequelize db:migrate
```

Creating the GraphQL server

With the dependencies installed and database setup, let's begin to flesh out the GraphQL server. Create a new `server.js` file and paste the code below into it:

```
// server.js

const express = require('express')
const bodyParser = require('body-parser')
const { graphqlExpress } = require('apollo-server-express')
const schema = require('./data/schema')

// create our express app
const app = express()

const PORT = 3000

// graphql endpoint
app.use('/api', bodyParser.json(), graphqlExpress({ schema }))

app.listen(PORT, () => {
  console.log(`The server is running on http://localhost:${PORT}/api`)
})
```

We import some of the dependencies we installed earlier: `express` is the Node.js framework, `body-parser` is used to parse incoming request body and `graphqlExpress` is the express implementation of Apollo server which will be used to power our GraphQL server. Then, we import our GraphQL `schema` which we'll created shortly.

Next, we define the route (in this case `/api`) for our GraphQL server. Then we add `body-parser` middleware to the route. Also, we add `graphqlExpress` passing along our GraphQL schema.

Finally, we start the server and listen on a specified port.

Defining GraphQL schema

Now, let's define our GraphQL schema. We'll keep the schema simple. Create a folder name `data` and, within this folder, create a new `schema.js` file then paste the code below into it:

```
// data/schema.js

const { makeExecutableSchema } = require('graphql-tools')
const resolvers = require('./resolvers')

// Define our schema using the GraphQL schema language
const typeDefs = `
  type User {
    id: Int!
    username: String!
    email: String!
  }

  type Query {
    me: User
  }

  type Mutation {
    signup (username: String!, email: String!, password: String!): String
    login (email: String!, password: String!): String
  }
`

module.exports = makeExecutableSchema({ typeDefs, resolvers })
```

We import `apollo-tools` which allows us to define our schema using the GraphQL schema language. We also import our resolvers which we'll create shortly. The schema contains just one type which is the `User` type. Then a `me` query which will be used to fetch the profile of the currently authenticated user. Then, we define two mutations; for users to signup and login respectively.

Lastly we use `makeExecutableSchema` to build the schema, passing to it our schema and the resolvers.

Writing resolver functions

Remember we referenced a `resolvers.js` file which doesn't exist yet. Now, let's create it and define our resolvers. Within the `data` folder, create a new `resolvers.js` file and paste the following code into it:

```
// data/resolvers.js

const { User } = require('../models')
const bcrypt = require('bcrypt')
const jsonwebtoken = require('jsonwebtoken')
```

```
require('dotenv').config()

const resolvers = {
  Query: {
    // fetch the profile of currently authenticated user
    async me (_, args, { user }) {
      // make sure user is logged in
      if (!user) {
        throw new Error('You are not authenticated!')
      }

      // user is authenticated
      return await User.findById(user.id)
    }
  },

  Mutation: {
    // Handle user signup
    async signup (_, { username, email, password }) {
      const user = await User.create({
        username,
        email,
        password: await bcrypt.hash(password, 10)
      })

      // return json web token
      return jsonwebtoken.sign(
        { id: user.id, email: user.email },
        process.env.JWT_SECRET,
        { expiresIn: '1y' }
      )
    },

    // Handles user login
    async login (_, { email, password }) {
      const user = await User.findOne({ where: { email } })

      if (!user) {
        throw new Error('No user with that email')
      }

      const valid = await bcrypt.compare(password, user.password)

      if (!valid) {
        throw new Error('Incorrect password')
      }

      // return json web token
      return jsonwebtoken.sign(
        { id: user.id, email: user.email },
        process.env.JWT_SECRET,
        { expiresIn: '1d' }
      )
    }
  }
}

module.exports = resolvers
```

First, we import the User model and other dependencies. `bcrypt` will be used for hashing users passwords, `jsonwebtoken` will be used to generate a JSON Web Token (JWT) which will be used to authenticate users and `dotenv` will be used to read from our `.env` file.

The `me` query will be used to fetch the profile of the currently authenticated user. This query accepts a `user` object as the context argument. `user` will either be an object or `null` depending on whether a user is logged in or not. If the user is not logged in, we simply throw an error. Otherwise, we fetch the details of the user by ID from the database.

The `signup` mutation accepts the user's username, email and password as arguments then create a new record with these details in the users database. We use the `bcrypt` package to hash the users password. The `login` mutation checks if a user with the email and password supplied exists in the database. Again, we use the `bcrypt` package to compare the password supplied with the password hash generated while creating the user. If the user exists, we generate a JWT that contains the user's ID and email. This JWT will be used to authenticate the user.

That's all for our resolvers. Noticed we use `JWT_SECRET` from the environment variable which we are yet to define. Add the line below to `.env`:

```
// .env

JWT_SECRET=somereallylongsecretkey
```

Adding authentication middleware

We'll use an auth middleware to validate incoming requests made to our GraphQL server. Open `server.js` and add the code below to it:

```
// server.js

const jwt = require('express-jwt')
require('dotenv').config()

// auth middleware
const auth = jwt({
  secret: process.env.JWT_SECRET,
  credentialsRequired: false
})
```

First, we import the `express-jwt` and the `dotenv` packages. Then we create an `auth` middleware which uses the `express-jwt` package to validate the JWT from the incoming requests `Authorization` header and set the `req.user` with the attributes (`id` and `email`) encoded in the JWT. It is

worth mentioning that `req.user` is not the Sequelize User model object. We set `credentialsRequired` to `false` because we want users to be able to at least signup and login first.

Then update the route as below:

```
// server.js

// graphql endpoint
app.use('/api', bodyParser.json(), auth, graphqlExpress(req => ({
  schema,
  context: {
    user: req.user
  }
})))
)
```

We add the `auth` middleware created above to the route. This makes the route secured as it will check to see if there is an `Authorization` header with a JWT on incoming requests. `express-jwt` adds the details of the authenticated user to the request body so we simply pass `req.user` as context to GraphQL. This way, `user` will be available as the context argument across our GraphQL server.

Testing it out

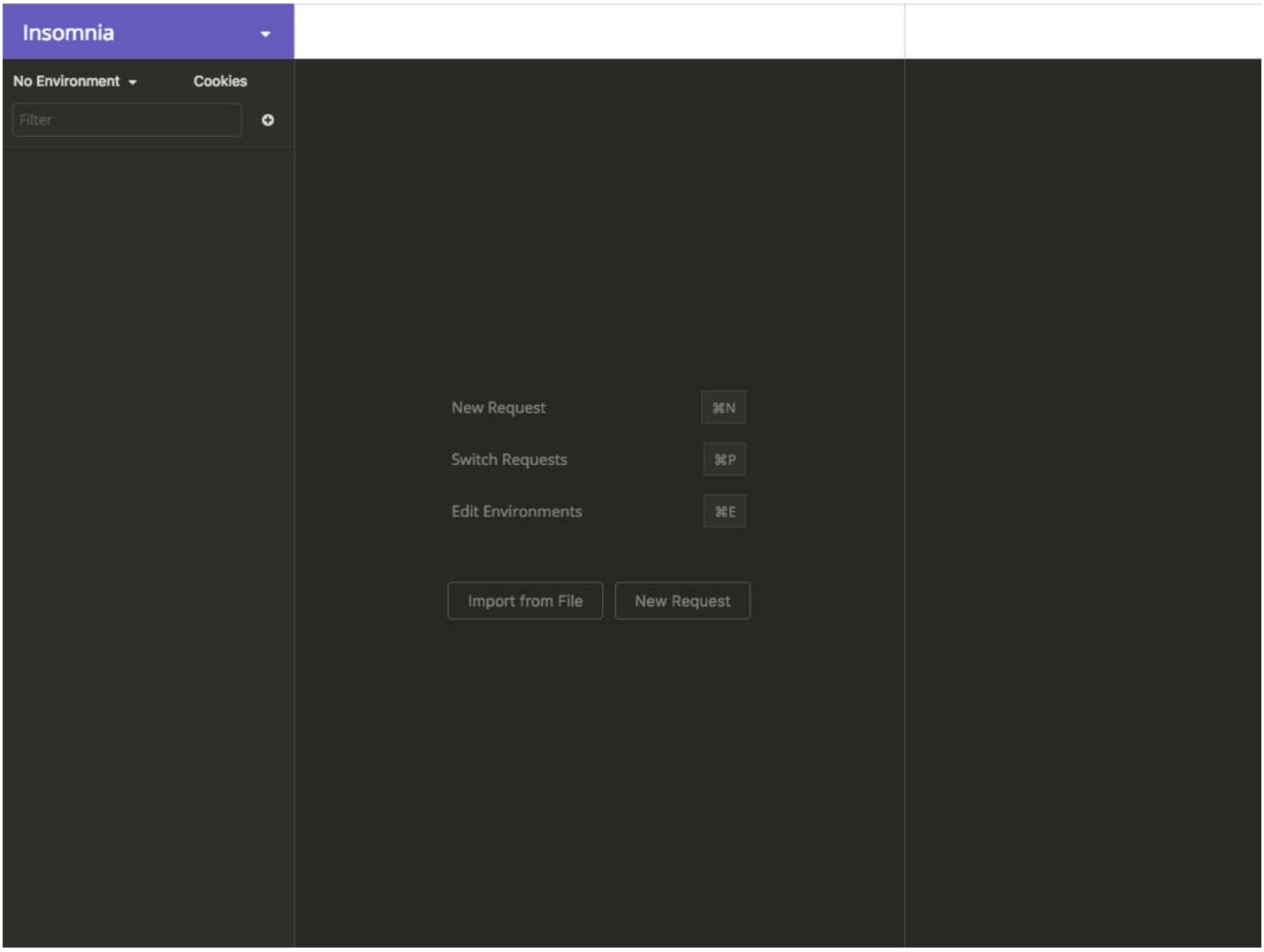
For the purpose of testing out our GraphQL, we' ll be making use of Insomnia. Of course you can make use of other HTTP clients that supports GraphQL or even GraphiQL.

Start the GraphQL server:

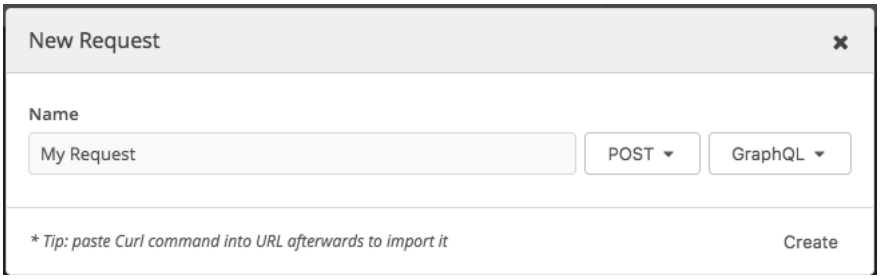
```
node server.js
```

It should be running on `http://localhost:3000/api`.

Then start Insomnia:



Click on create **New Request**. Give the request a name if you want, then select **POST** as the request method, then select **GraphQL Query**. Finally, click **Create**.

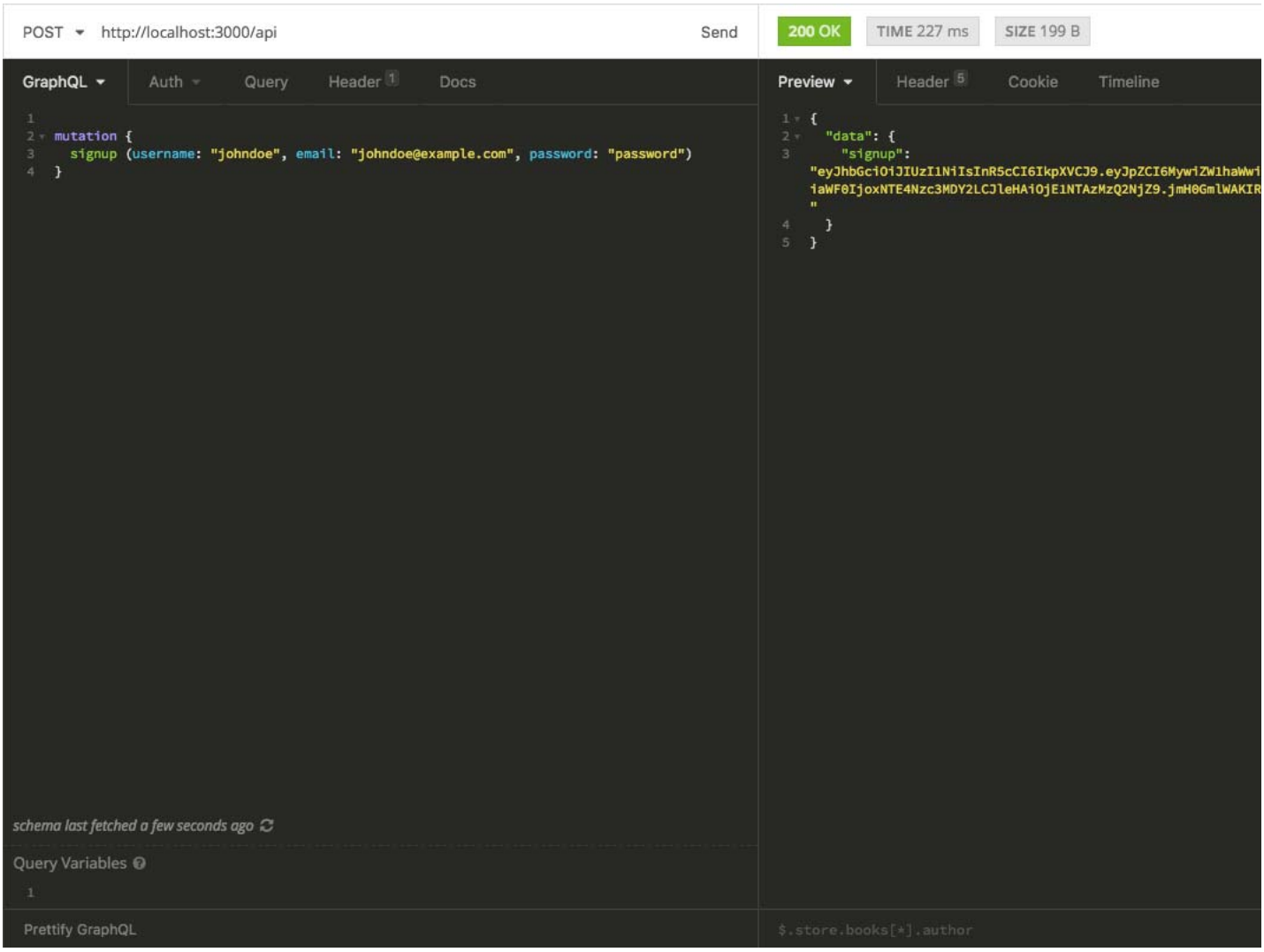


Next, enter `http://localhost:3000/api`. in the address bar.

Let' s try signing up:

```
mutation {
  signup (username: "johndoe", email: "johndoe@example.com", password: "password")
}
```

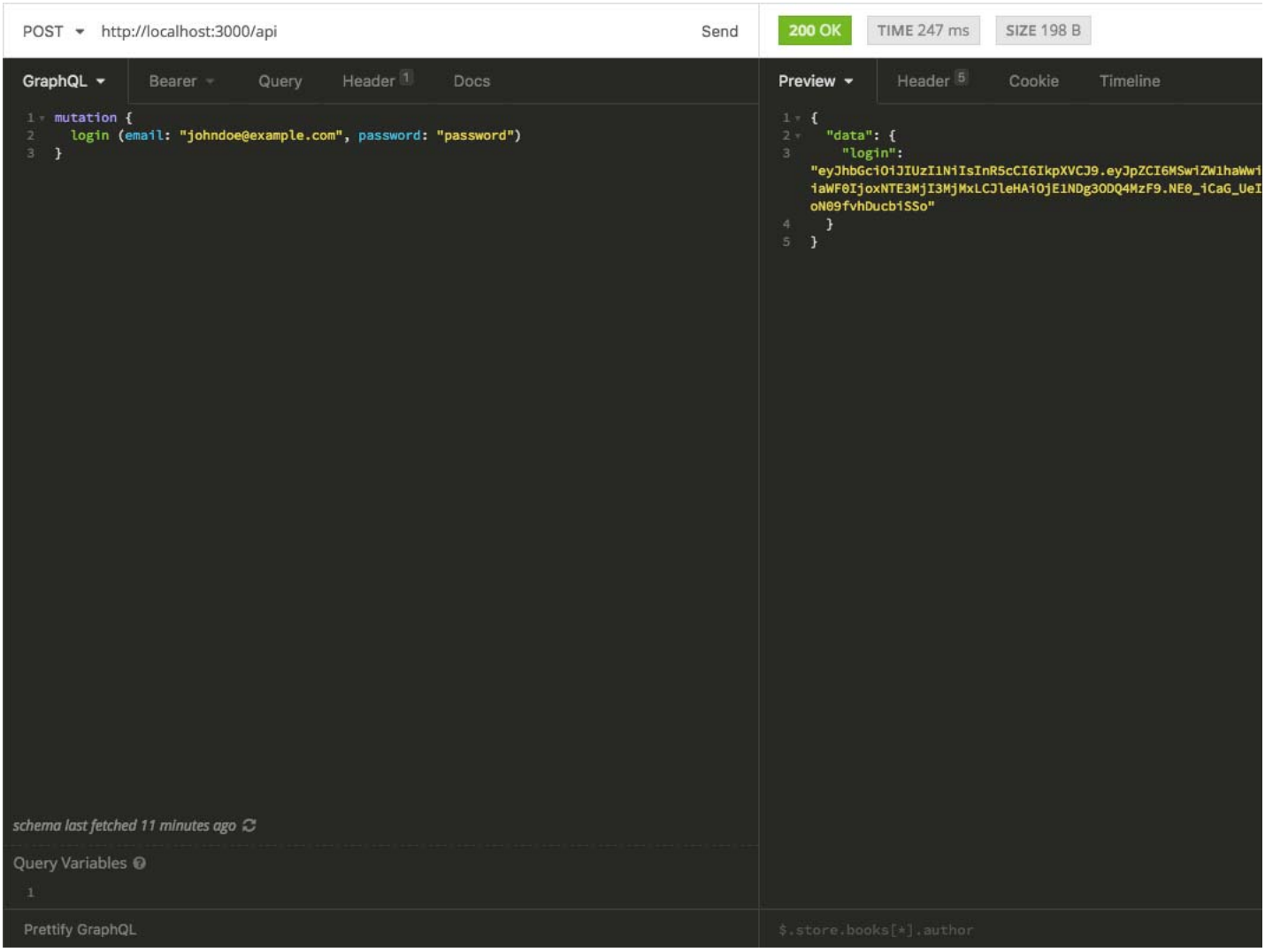
We should get a response as in the image below:



Next, we can login:

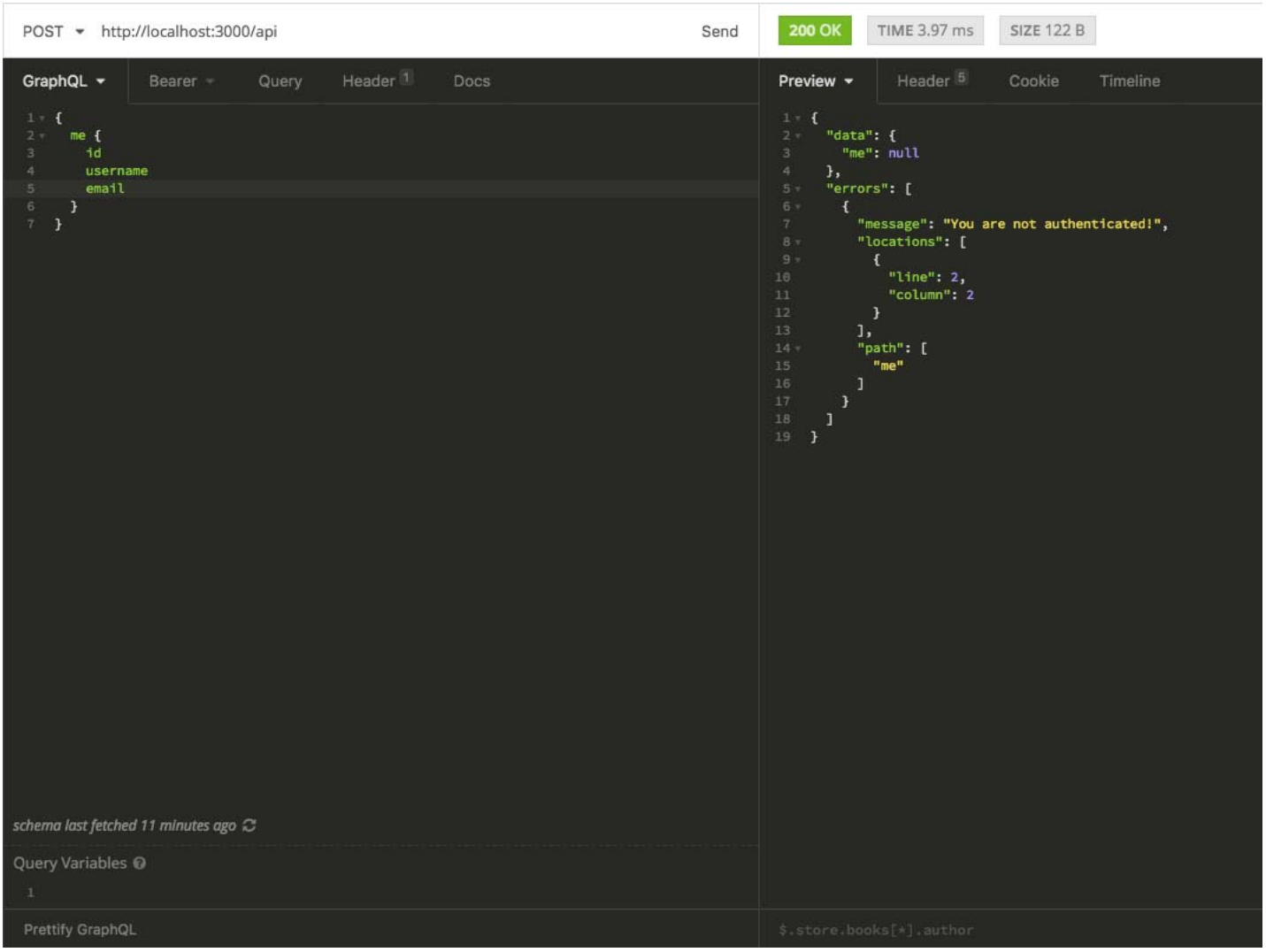
```
mutation {
  login(email: "johndoe@example.com", password: "password")
}
```

We should get a response as in the image below:



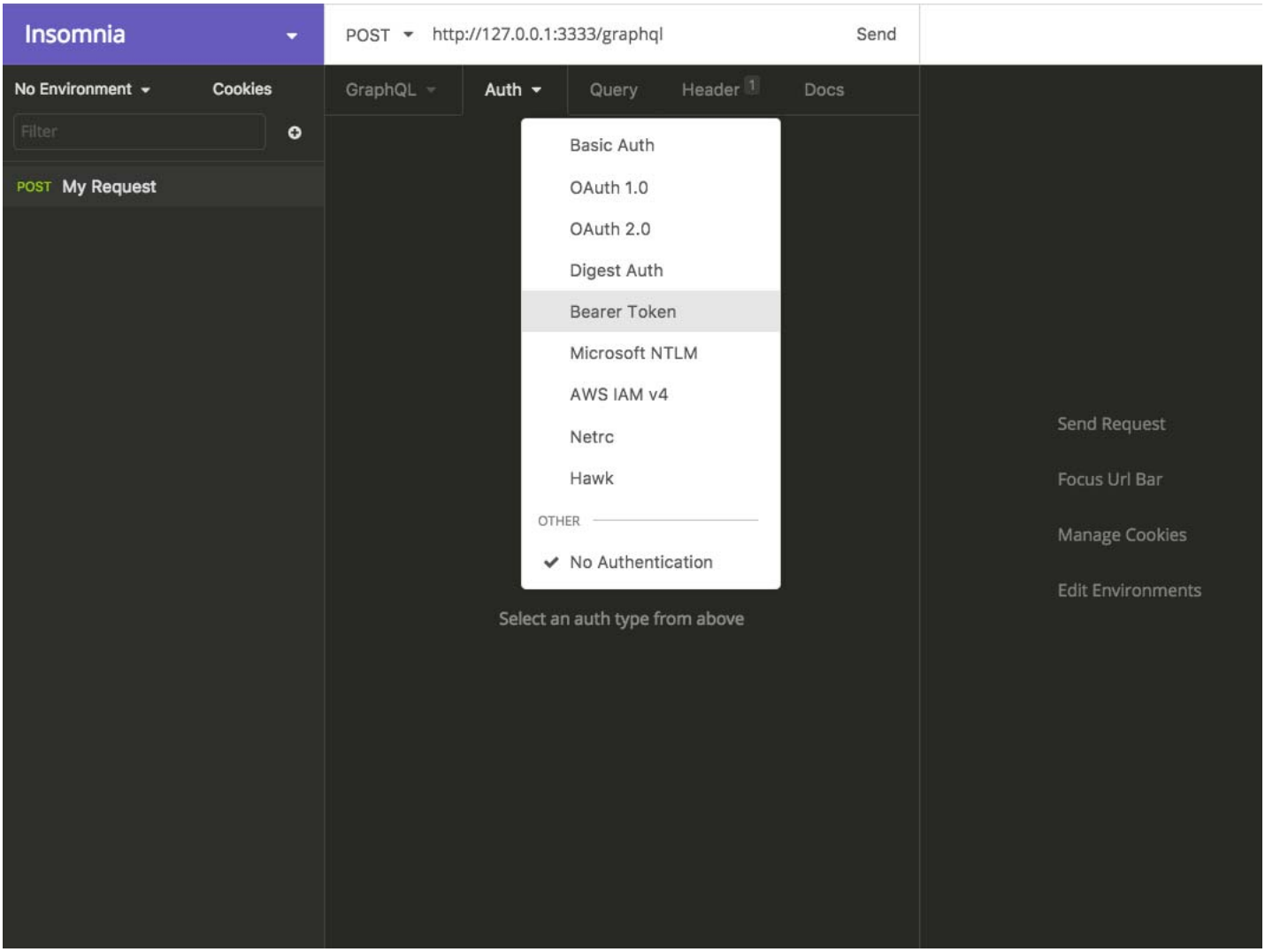
A JWT is returned on successful login.

Now, if we try fetching the user's profile using the `me` query, we'll get the **You are not authenticated!** error message as in the image below:



Remember we said the auth middleware will check the incoming request for an `Authorization` header. So, we need to set the `Authorization: Bearer <token>` header to authenticate the request.

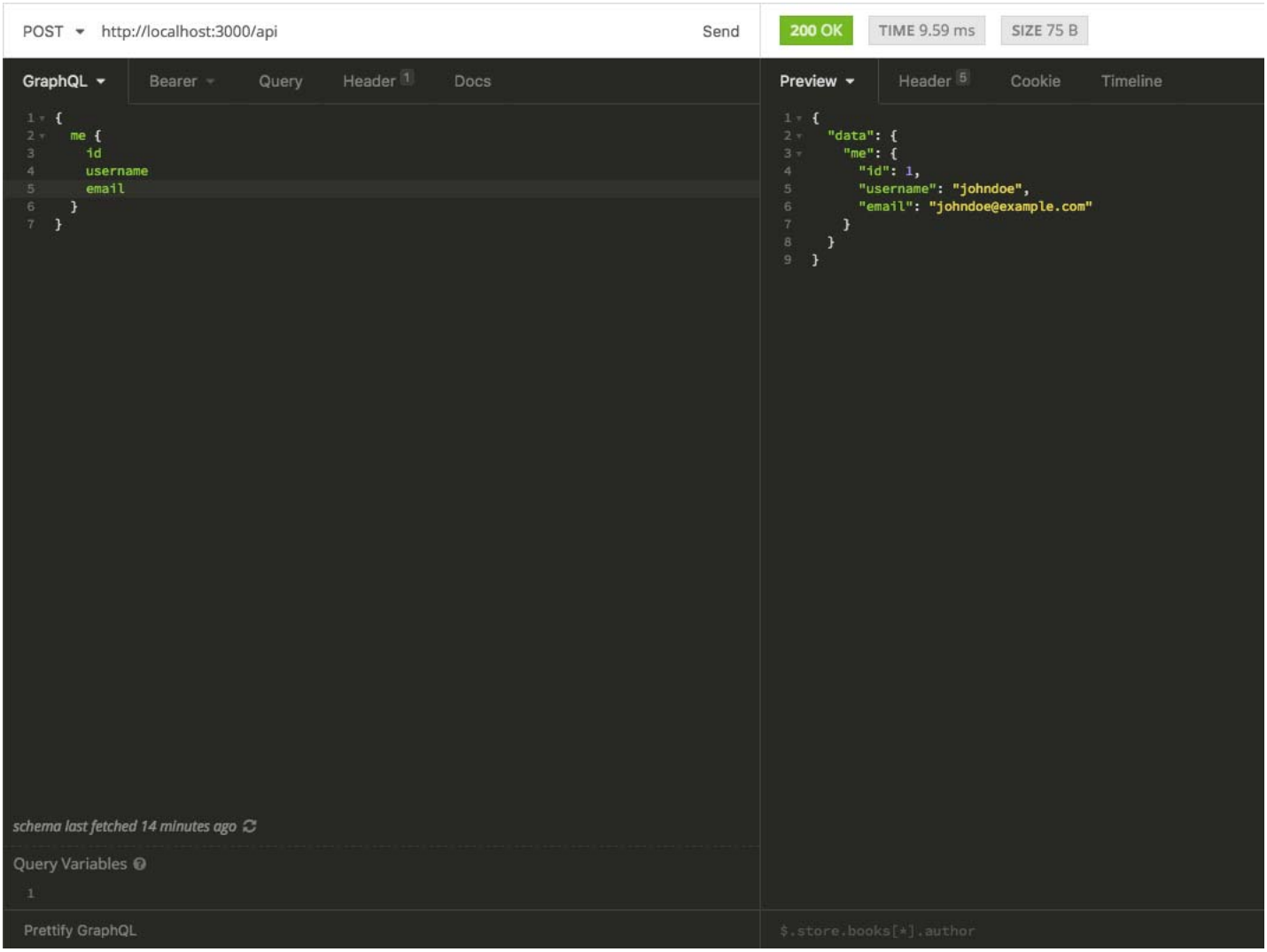
From **Auth** dropdown, select **Bearer Token** and paste the token (JWT) above in the field provided.



Now we can fetch the profile of the currently authenticated user:

```
{
  me {
    id
    username
    email
  }
}
```

We should get a response as in the image below:



The complete code is available on GitHub.

Conclusion

That's it! In this tutorial, we have seen how to add authentication using JWT to a GraphQL server. We also saw how to test our GraphQL server using Insomnia.

In the next part in this series, we'll cover how to add authentication to GraphQL using a third-party authentication service like Auth0.

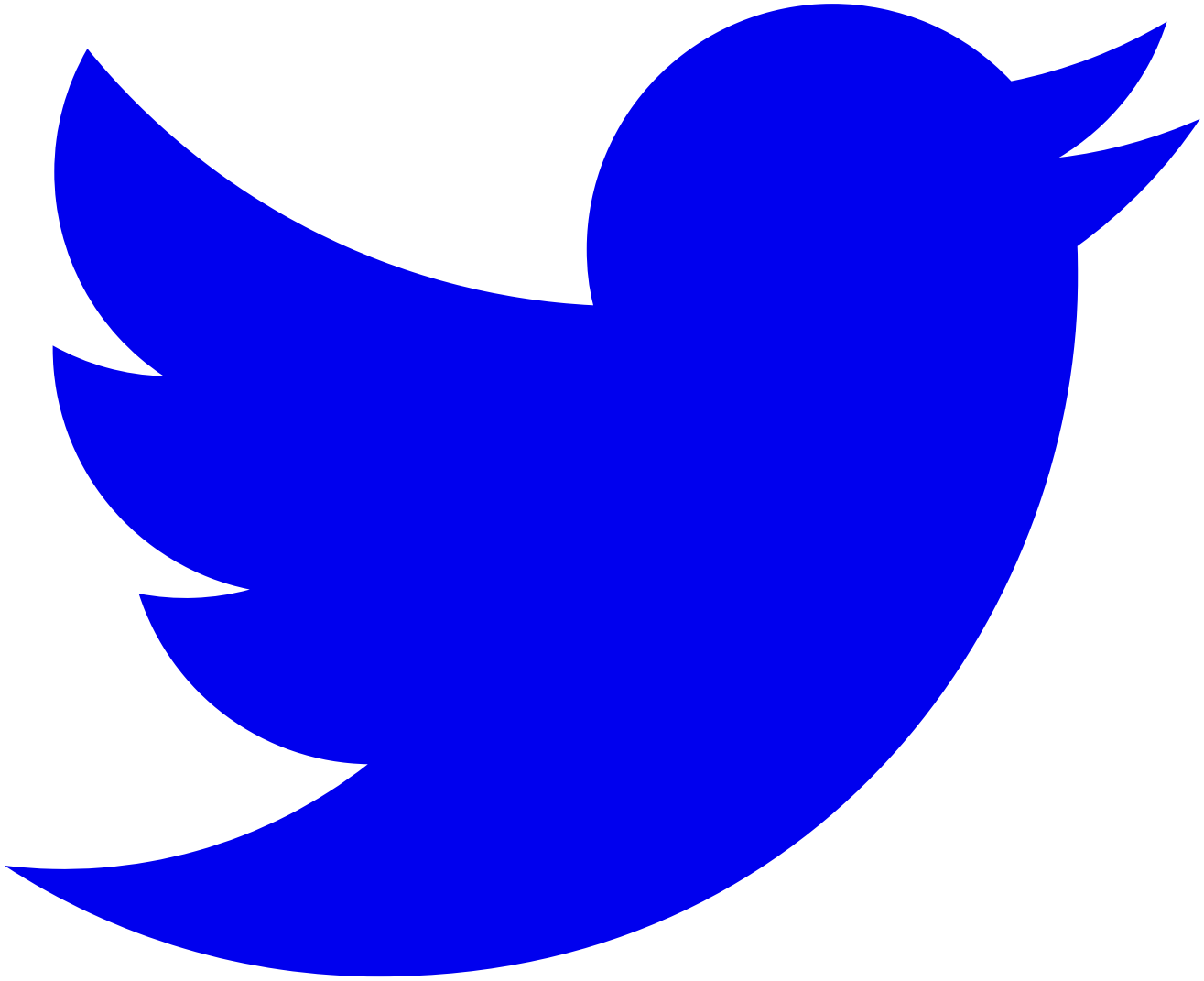
This is part 2 of a 3 part tutorial. You can find part 1 here and part 3 here.

authentication GraphQL JWT

April 10, 2018

Chimezie Enyinnaya

Share article



Multi-channel event Publishing

Pusher at Startup Weekend Edinburgh

PUSHER

Products

- Channels
- Beams

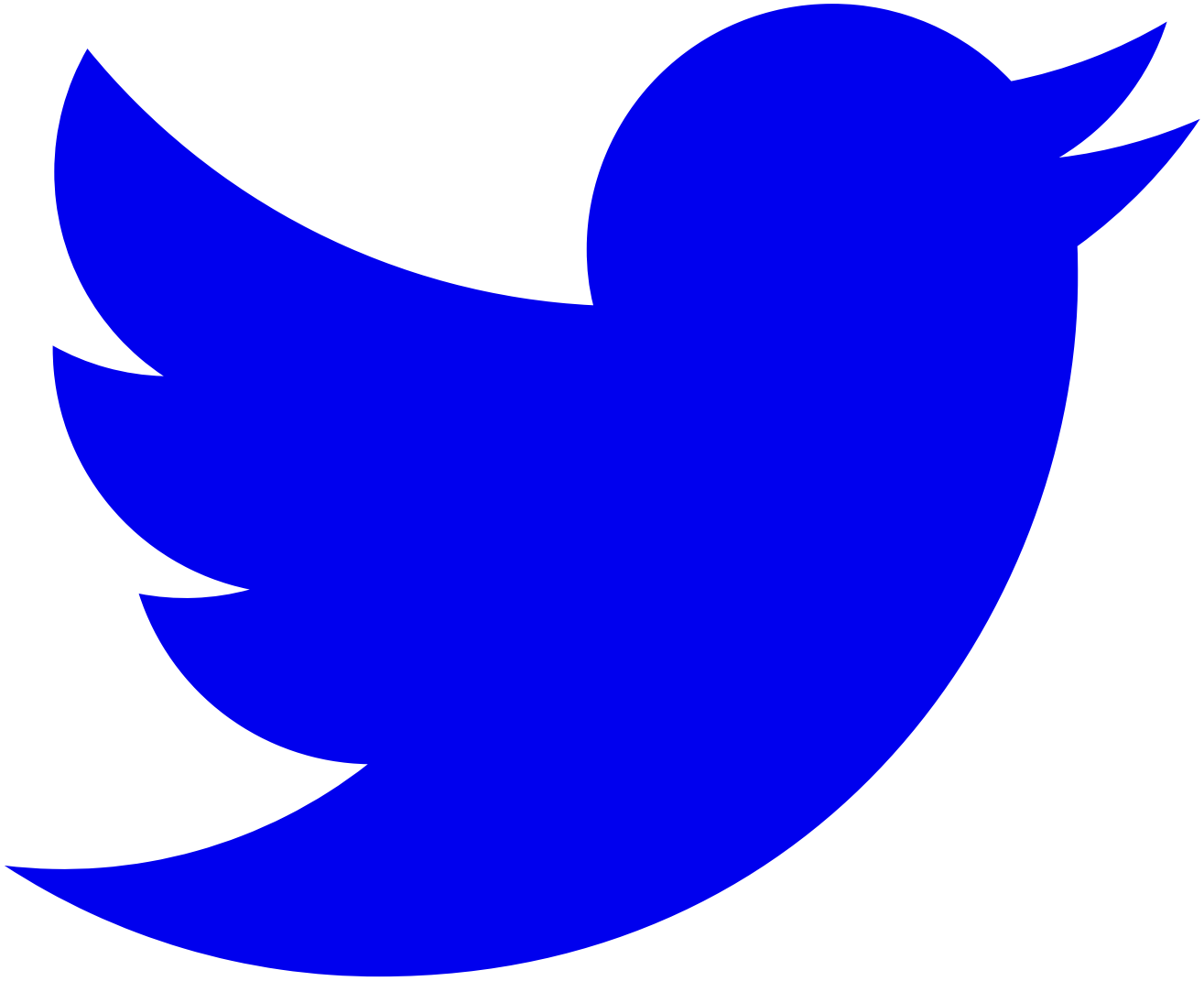
Developers

- Docs
- Tutorials
- Status
- Support
- Sessions

Company

- Contact Sales
- Customer stories
- Terms of Service
- Security
- Careers
- Blog
- Legal

Connect



-