Lab 6: Authentication with Apollo and React

This lab covers the following topics:

- Implementing authentication in Node.js and Apollo
- Signing up and logging in users
- Authenticating GraphQL queries and mutations
- Accessing the user from the request context

Lab Solution

Complete solution for this lab is available in the following directory:

`cd ~/Desktop/react-graphql-course/labs/Lab06`

Authentication with GraphQL

The basics of authentication should now be clear to you. Our task is now to implement a secure way for users to authenticate. If we have a look at our current database, we are missing the required fields. Let's prepare and add a [password] and an [email] field. As we learned in Lab 3, we create a migration to edit our user table. You can look up the commands in the third lab if you have forgotten them:

```
sequelize migration:create --migrations-path src/server/migrations --name add-email-password-to-post
```

The preceding command generates the new file for us. You can replace the content of it and try writing the migration on your own, or check for the right commands in the following code snippet:

```
'use strict';

module.exports = {
  up: (queryInterface, Sequelize) => {
    return Promise.all([
      queryInterface.addColumn('Users',
        'email',
        {
          type: Sequelize.STRING,
          unique : true,
        }
      ),
      queryInterface.addColumn('Users',
        'password',
        {
          type: Sequelize.STRING,
        }
      ),
    ]);
  },

  down: (queryInterface, Sequelize) => {
    return Promise.all([
      queryInterface.removeColumn('Users', 'email'),
      queryInterface.removeColumn('Users', 'password'),
    ]);
  }
};
```

All fields are simple strings. You can execute the migration, as stated in the Lab 3. The email address needs to be unique. Our old seed file for the users needs to be updated now to represent the new fields that we have just added. Copy the following fields:

```
password: '$2a$10$bE3ovf9/Tiy/d68bwNUQ0.zCjwtNFq9ukg9h4rhKiHCb6x5ncKife',
email: 'test1@example.com',
```

Do this for all three users and change the email address for each of them. Otherwise, it will not work. The password is in hashed format and represents the plain password 123456789. As we have added new fields in a separate migration, we have to add these to the model.

Open and add the new lines as fields to the [user.js] file in the [model] folder:

```
email: DataTypes.STRING,
password: DataTypes.STRING,
```

The first thing to do now is get the login running. At the moment, we are just faking being logged in as the first user in our database.

Apollo login mutation

We are now going to edit our GraphQL schema and implement the matching resolver function. Let's start with the schema and a new mutation to the [RootMutation] object of our [schema.js] file:

```
login (
  email: String!
  password: String!
): Auth
```

The preceding schema gives us a login mutation that accepts an email address and a password. Both are required to identify and authenticate the user. We then need to respond with something to the client. For now, the [Auth] type returns a token, which is a JWT in our case. You might want to add a different option according to your requirements:

```
type Auth {
  token: String
}
```

The schema is now ready. Head over to the [resolvers] file and add the login function inside the mutation object. Before doing this, we have to install and import two new packages:

```
npm install --save jsonwebtoken bcrypt
```

The [jsonwebtoken] package handles everything required to sign, verify, and decode JWTs.

The important part is that all passwords for our users are not saved as plain text but are first encrypted using hashing, including a random salt. This generated hash cannot be decoded or decrypted to a plain password, but the package can verify if the password that was sent with the login attempt matches with the password hash saved on the user. Import these packages at the top of the resolvers file:

```
import bcrypt from 'bcrypt';
import JWT from 'jsonwebtoken';
```

The [login] function receives [email] and [password] as parameters. It should look like the following code:

```
login(root, { email, password }, context) {
  return User.findAll({
    where: {
      email
    },
    raw: true
  }).then(async (users) => {
    if(users.length = 1) {
      const user = users[0];
      const passwordValid = await bcrypt.compare(password,
      user.password);
      if (!passwordValid) {
        throw new Error('Password does not match');
      }
      const token = JWT.sign({ email, id: user.id }, JWT_SECRET, {
        expiresIn: '1d'
      });

      return {
        token
      };
    } else {
      throw new Error("User not found");
    }
  });
},
```

The preceding code goes through the following steps:

1. We query all users where the email address matches.
2. If a user is found, we can go on. It is not possible to have multiple users with the same address, as the MySQL unique constraint forbids this.
3. Next, we use the user password and compare it with the submitted password, using the [bcrypt] package, as explained previously.
4. If the password was correct, we generate a JWT token to the [jwt] variable using the [jwt.sign] function. It takes three arguments: the payload, which is the user id and their email address; the key with which we sign the JWT; and the amount of time in which the JWT is going to expire.
5. In the end, we return an object containing our JWT.

ProTip

Something that you might need to rethink is how much detail you give in an error message. For example, we might not want to distinguish between an incorrect password and a non-existent user. It gives possible attackers or data collectors the opportunity to know which email address is in use.

The [login] function is not working yet, because we are missing [JWT_SECRET], which is used to sign the JWT. In production, we use the environment variables to pass the JWT secret key into our back end code so that we use this approach in development too.

For Linux or Mac, you can use the following command directly in the Terminal:

```
export
JWT_SECRET=awv4BcIzsRysXkhoSAb8t8lNENgXSqBruVlLwd45kGdYjeJHLap9LUJ1t9DTdw36DvLcWs3qEkPy
```

The [export] function sets the [JWT_SECRET] environment variable for you. Replace the JWT provided with a random one. You can use any password generator by setting the character count to 128 and excluding any special characters. Setting the environment variable allows us to read the secret in our application. You have to replace it when going to production.

Insert the following code at the top of the file:

```
const { JWT_SECRET } = process.env;
```

This code reads the environment variable from the global Node.js [process] object. Be sure to replace the JWT once you publish your application, and be sure to always store the secret securely. After letting the server reload, we can send the first login request. We are going to take a look how to do this in React later, but the following code shows an example using Postman:

```
{
  "operationName":null,
  "query": "mutation login($email : String!, $password : String!) {
   login(email: $email, password : $password) { token }}",
  "variables":{
    "email": "test1@example.com",
    "password": "123456789"
  }
}
```

This request should return a token:

```
{
  "data": {
    "login": {
      "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InRlc3QxQGV4YW1wbGUuY29tIiwiaWQiOjEsI
Uj0MPHvlY"
    }
  }
}
```

As you can see, we have generated a signed JWT and returned it within the mutation's response. We can go on and send the token with every request inside the HTTP authorization header. We can then get the authentication running for all the other GraphQL queries or mutations that we have implemented so far.

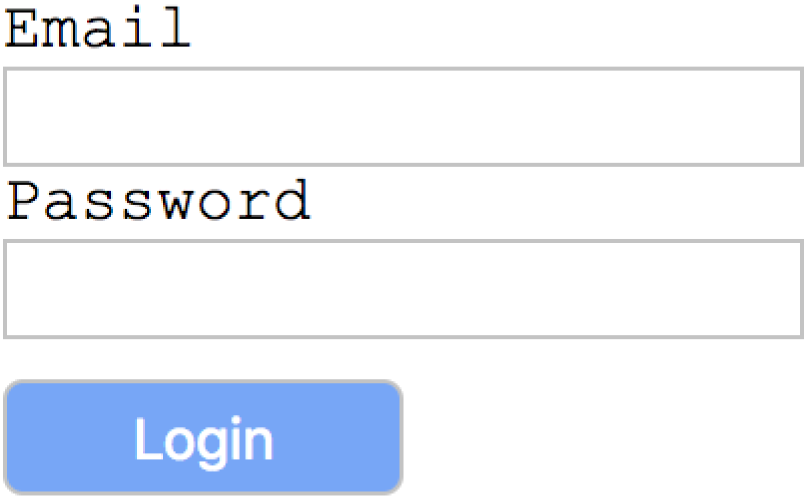
Let's continue and learn how to set up React to work with our authentication on the back end.

The React login form

We need to handle the different authentication states of our application:

- The first scenario is that the user is not logged in and cannot see any posts or chats. In this case, we need to show a login form to allow the user to authenticate themselves.
- The second scenario is that an email and password are sent through the login form. The response needs to be interpreted, and if the result is correct, we need to save the JWT inside the [localStorage] of the browser for now.
- When changing the [localStorage], we also need to rerender our React application to show the logged-in state.
- Furthermore, the user should be able to log out again.
- We must also handle if the JWT expires and the user is unable to access any functionalities.

The result for our form looks as follows:



To get started with the login form, observe the following steps:

1. Set up the login [Mutation] component. It is likely that we only need this component at one place in our code, but it is a good idea to save Apollo requests in separate files.
2. Build the login form component, which uses the login mutation to send the form data.
3. Create the [CurrentUser] query to retrieve the logged-in user object.
4. Conditionally render the login form if the user is not authenticated or the real application like the news feed if the user is logged in.

Begin by creating a new [login.js] file inside the [mutations] folder for the client components:

```
import React, { Component } from 'react';
import { Mutation } from 'react-apollo';
import gql from 'graphql-tag';

const LOGIN = gql`
  mutation login($email : String!, $password : String!) {
    login(email : $email, password : $password) {
      token
    }
  }`;
```

```
export default class LoginMutation extends Component {
  render() {
    const { children } = this.props;
    return (
      <Mutation
        update = {(store, { data: { login } }) => {
          if(login.token) {
            localStorage.setItem('jwt', login.token);
          }
        }}
        mutation={LOGIN}>
          {(login, { loading, error}) =>
            React.Children.map(children, function(child){
              return React.cloneElement(child, { login, loading, error });
            })
          }
      </Mutation>
    )
  }
}
```

Like in the previous mutations, we parse the query string and hand the resulting [login] function over to all the children of the component. We now give the [loading] and [error] states to those children, in case we want to show an error or loading message. The [update] function is a bit different than before. We don't write the return value in the Apollo cache, but we do need to store the JWT inside the [localStorage]. The syntax is pretty simple. You can directly use [localStorage.get] and [localStorage.set] to interact with the web storage.

Now, we implement the underlying children, which makes up the login form. To do this, we create a [loginregister.js] file directly in the [components] folder. As you may expect, we handle the login and registration of users in one component. Import the dependencies:

```
import React, { Component } from 'react';
import Error from './error';
import LoginMutation from './mutations/login';
```

The [LoginForm] class will store the form state, display an error message if something goes wrong, and send the login mutation including the form data:

```
class LoginForm extends Component {
  state = {
    email: '',
    password: '',
  }
  login = (event) => {
    event.preventDefault();
    this.props.login({ variables: { email: this.state.email, password:
    this.state.password }});
  }
  render() {
    const { error } = this.props;
    return (
      <div className="login">
```

```
        <form onSubmit={this.login}>
          <label>Email</label>
          <input type="text" onChange={(event) => this.setState({email:
           event.target.value})} />
          <label>Password</label>
          <input type="password" onChange={(event) =>
           this.setState({password: event.target.value})} />
          <input type="submit" value="Login" />
        </form>
        {error && (
          <Error><p>There was an error logging in!</p></Error>
        )}
      </div>
    )
  }
}
```

We render the login form inside the wrapping component, which is called [LoginRegisterForm]. It is important to surround the form with the login mutation so that we can send the Apollo request:

```
export default class LoginRegisterForm extends Component {
  render() {
    return (
      <div className="authModal">
        <LoginMutation><LoginForm/></LoginMutation>
      </div>
    )
  }
}
```

All the basics for authenticating the user are now ready, but they have not been imported yet or displayed anywhere. Open the [App.js] file. There, we directly display the feed, chats, and the top bar. The user should not be allowed to see everything if he is not logged in. Continue reading to change this.

Import the new form that we have just created:

```
import LoginRegisterForm from './components/loginregister';
```

We then have to store whether the user is logged in or not. We save it in the component state, as follows:

```
state = {
  loggedIn: false
}
```

When loading our page, this variable needs to be set to [true] if we have a token in our [localStorage]. We handle this inside the [componentWillMount] function provided by React:

```
componentWillMount() {
  const token = localStorage.getItem('jwt');
  if(token) {
      this.setState({loggedIn: true});
  }
}
```

Then, in the [render] method, we can use conditional rendering to show the login form when the [loggedIn] state variable is set to [false], which means that there is no JWT inside our [localStorage]:

```
{this.state.loggedIn ?
  <div>
    <Bar />
    <Feed />
    <Chats />
  </div>
  : <LoginRegisterForm/>
}
```

If you try the login page, you will see that nothing happens, even though no error message is shown. That happens because we save the JWT, but we do not tell React to rerender our [App] class. We only check for the JWT when the page loads initially. To test your implementation, you can reload the window, and you should be logged in.

We have to pass a function down the React tree to the components, who are then able to trigger a logged-in state so that React can rerender and show the logged in area. We call this function [changeLoginState] and implement it inside the [App.js] file as follows:

```
changeLoginState = (loggedIn) => {
  this.setState({ loggedIn });
}
```

The function can change the current application state as specified through the [loggedIn] parameter. We then integrate this method into the [LoginMutation] component. To do this, we edit the [render] method of the [App] class to pass the right property:

```
<LoginRegisterForm changeLoginState={this.changeLoginState}/>
```

Then, inside the [LoginRegisterForm] class, we replace the [render] method with the following code:

```
render() {
  const { changeLoginState } = this.props;
  return (
    <div className="authModal">
      <LoginMutation changeLoginState={changeLoginState}><LoginForm/></LoginMutation>
    </div>
  )
}
```

Edit the [LoginMutation] component and extract the new function from the properties:

```
const { children, changeLoginState } = this.props;
```

We can then execute the [changeLoginState] function within the [update] method:

```
if(login.token) {
  localStorage.setItem('jwt', login.token);
  changeLoginState(true);
}
```

When logging in, our application presents us with the common posts feed as before. The authentication flow is now working, but there is one more open task. In the next step, we allow new users to register at Graphbook.

Apollo sign up mutation

You should now be familiar with creating new mutations. First, edit the schema to accept the new mutation:

```
signup (
  username: String!
  email: String!
  password: String!
): Auth
```

We only need the [username], [email], and [password] properties that were mentioned in the preceding code to accept new users. If your application requires a gender or something else, you can add it here. When trying to sign up, we need to ensure that neither the email address nor the username is already taken. You can copy over the code to implement the resolver for signing up new users:

```
signup(root, { email, password, username }, context) {
  return User.findAll({
    where: {
      [Op.or]: [{email}, {username}]
    },
    raw: true,
  }).then(async (users) => {
    if(users.length) {
      throw new Error('User already exists');
    } else {
      return bcrypt.hash(password, 10).then((hash) => {
        return User.create({
          email,
          password: hash,
          username,
          activated: 1,
        }).then((newUser) => {
          const token = JWT.sign({ email, id: newUser.id }, JWT_SECRET,
          {
            expiresIn: '1d'
          });
          return {
            token
          };
        });
      });
    }
  });
},
```

Let's go through the code step by step:

1. As mentioned previously, we first check if a user with the same email or username exists. If this is the case, we throw an error. We use the [Op.or] Sequelize operator to implement the MySQL OR condition.
2. If the user does not exist, we can hash the password using [bcrypt]. You cannot save the plain password for security reasons. When running the [bcrypt.hash] function, a random salt is used to make sure nobody ever

gets access to the original password. This command takes quite some computing time, so the [bcrypt.hash] function is asynchronous, and the promise must be resolved before continuing.

3. The encrypted password, including the other data the user has sent, is then inserted in our database as a new user.
4. After creating the user, we generate a JWT and return it to the client. The JWT allows us to log in the user directly after signing up. If you do not want this behavior, you can of course just return a message to indicate that the user has signed up successfully.

You can now test the [signup] mutation again with Postman if you want while starting the back end using [npm run server]. We have now finished the back end implementation, so we start working on the front end.

React sign up form

The registration form is nothing special. We follow the same steps as we took with the login form. You can clone the [LoginMutation] component, replace the request at the top with the [signup] mutation, and hand over the [signup] method to the underlying children. At the top, import all the dependencies and then parse the new query:

```
import React, { Component } from 'react';
import { Mutation } from 'react-apollo';
import gql from 'graphql-tag';

const SIGNUP = gql`
  mutation signup($email : String!, $password : String!, $username :
   String!) {
    signup(email : $email, password : $password, username : $username) {
    token
  }
}`;
```

As you can see, the [username] field is new here, which we send with every [signup] request. The component itself has not changed, so we have to extract the JWT from the [signup] field when logging the user in after a successful request.

We use the [changeLoginState] method to do so. We also changed the name of the mutation function we pass from [login] to [signup], of course:

```
export default class SignupMutation extends Component {
  render() {
    const { children, changeLoginState } = this.props;
    return (
      <Mutation
        update = {(store, { data: { signup } }) => {
          if(signup.token) {
            localStorage.setItem('jwt', signup.token);
            changeLoginState(true);
          }
        }}
        mutation={SIGNUP}>
          {(signup, { loading, error}) =>
            React.Children.map(children, function(child){
              return React.cloneElement(child, { signup, loading, error
              });
            })
          }
      </Mutation>
    )
  }
}
```

It is a good thing for the developer to see, that the [login] and [signup] mutations are quite similar. The biggest change is that we conditionally render the login form or the registration form. In the [loginregister.js] file, you first import the new mutation. Then, you replace the complete [LoginRegisterForm] class with the following new one:

```
export default class LoginRegisterForm extends Component {
  state = {
    showLogin: true
  }
  render() {
    const { changeLoginState } = this.props;
    const { showLogin } = this.state;
    return (
      <div className="authModal">
        {showLogin && (
          <div>
            <LoginMutation changeLoginState=
              {changeLoginState}><LoginForm/></LoginMutation>
            <a onClick={() => this.setState({ showLogin: false })}>Want
             to sign up? Click here</a>
          </div>
        )}
        {!showLogin && (
          <div>
            <RegisterMutation changeLoginState=
              {changeLoginState}><RegisterForm/></RegisterMutation>
            <a onClick={() => this.setState({ showLogin: true })}>Want to
             login? Click here</a>
          </div>
        )}
      </div>
    )
  }
}
```

You should notice that we are storing a [showLogin] variable in the component state, which decides if the login or register form is shown. The matching mutation wraps each form, and they handle everything from there. The last thing to do is insert the new register form in the [login.js] file:

```
class RegisterForm extends Component {
  state = {
    email: '',
    password: '',
    username: '',
  }
  login = (event) => {
    event.preventDefault();
    this.props.signup({ variables: { email: this.state.email, password:
```

```
    this.state.password, username: this.state.username }});
  }
  render() {
    const { error } = this.props;
    return (
      <div className="login">
        <form onSubmit={this.login}>
          <label>Email</label>
          <input type="text" onChange={(event) => this.setState({email:
           event.target.value})} />
          <label>Username</label>
          <input type="text" onChange={(event) =>
           this.setState({username: event.target.value})} />
          <label>Password</label>
          <input type="password" onChange={(event) =>
           this.setState({password: event.target.value})} />
          <input type="submit" value="Sign up" />
        </form>
        {error && (
          <Error><p>There was an error logging in!</p></Error>
        )}
      </div>
    )
  }
}
```

In the preceding code, I have added the [username] field, which has to be given to the mutation. Everything is now set to invite new users to join our social network and log in as often as they want.

In the next section, we will see how to use authentication with our existing GraphQL requests.

Authenticating GraphQL requests

The problem is that we are not using the authentication everywhere at the moment. We verify that the user is who they say they are, but we do not recheck this when the requests for chats or messages come in. To accomplish this, we have to send the JWT token, which we generated specifically for this case, with every Apollo request. On the back end, we have to specify which request requires authentication, read the JWT from the HTTP authorization header, and verify it.

Open the [index.js] file from the [apollo] folder for the client-side code. Our [ApolloClient] is currently configured as explained in Lab 4. Before sending any request, we have to read the JWT from the [localStorage] and add it as an HTTP authorization header. Inside the [link] property, we have specified the links for our [ApolloClient] processes. Before the configuration of the HTTP link, we insert a third preprocessing hook as follows:

```
const AuthLink = (operation, next) => {
  const token = localStorage.getItem('jwt');
  if(token) {
    operation.setContext(context => ({
      ...context,
      headers: {
        ...context.headers,
        Authorization: `Bearer ${token}`,
      },
    }));
  }
  return next(operation);
};
```

Here, we have called the new link [AuthLink], because it allows us to authenticate the client on the server. You can copy the [AuthLink] approach to other situations in which you need to customize the header of your Apollo requests. Here, we just read the JWT from the [localStorage] and, if it is found, we construct the header using the spread operator and adding our token to the Authorization field as a Bearer token. It is everything that needs to be done on the client-side.

To clarify things, take a look at the following [link] property to see how to use this new preprocessor. There is no initialization required; it is merely a function that is called every time a request is made. Copy the [link] configuration to our Apollo Client setup:

```
link: ApolloLink.from([
  onError(({ graphQLErrors, networkError }) => {
    if (graphQLErrors) {
      graphQLErrors.map(({ message, locations, path }) =>
      console.log(`[GraphQL error]: Message: ${message}, Location:
      ${locations}, Path: ${path}`));
      if (networkError) {
        console.log(`[Network error]: ${networkError}`);
      }
    }
  }),
  AuthLink,
  new HttpLink({
    uri: 'http://localhost:8000/graphql',
    credentials: 'same-origin',
  }),
]),
```

For our back end, we need a pretty complex solution. Create a new file called [auth.js] inside the [graphql] [services] folder. We want to be able to mark specific GraphQL requests in our schema with a so-called directive. If we add this directive to our GraphQL schema, we execute a function whenever the marked GraphQL action is requested. In this function, we can verify whether the user is logged in or not. Have a look at the following function and save it right to the [auth.js] file:

```
import { SchemaDirectiveVisitor, AuthenticationError } from 'apollo-server-express';

class AuthDirective extends SchemaDirectiveVisitor {
  visitFieldDefinition(field) {
    const { resolve = defaultFieldResolver } = field;
    field.resolve = async function(...args) {
      const ctx = args[2];
      if (ctx.user) {
        return await resolve.apply(this, args);
      } else {
        throw new AuthenticationError("You need to be logged in.");
      }
    };
  }
}

export default AuthDirective;
```

Starting from the top, we import the [SchemaDirectiveVisitor] class from the [apollo-server-express] package. This class allows us to handle all requests that have the [AuthDirective] attached. We extend the [SchemaDirectiveVisitor] class and override the [visitFieldDefinition] method. Within the method, we resolve the current context of the request with the [field.resolve] function. If the context has a user attached, we can be sure that the authorization header has been checked before and the identified user has been added to the request context. If not, we throw an error. The [AuthError] we are throwing gives us the opportunity to implement certain behaviors when an [UNAUTHENTICATED] error is sent to the client.

We have to load the new [AuthDirective] class in the [graphql] [index.js] file, which sets up the whole Apollo Server:

```
import auth from './auth';
```

While using the [makeExecutableSchema] function to combine the schema and the resolvers, we can add a further property to handle all schema directives, as follows:

```
const executableSchema = makeExecutableSchema({
  typeDefs: Schema,
  resolvers: Resolvers.call(utils),
  schemaDirectives: {
    auth: auth
  },
});
```

Here, we combine the schema, the resolvers, and the directives into one big object. It is important to note that the [auth] index inside the [schemaDirectives] object is the mark that we have to set at every GraphQL request in our schema that requires authentication. To verify what we have just done, go to the GraphQL schema and edit [postsFeed] [RootQuery] by adding [@auth] at the end of the line like this:

```
postsFeed(page: Int, limit: Int): PostFeed @auth
```

Because we are using a new directive, we also must define it in our GraphQL schema so that our server knows about it. Copy the following code directly to the top of the schema:

```
directive @auth on QUERY | FIELD_DEFINITION | FIELD
```

This tiny snippet tells the Apollo Server that the [@auth] directive is usable with queries, fields, and field definitions so that we can use it everywhere.

As we have implemented the error component earlier, we are now correctly receiving an unauthenticated error for the [postsFeed] query if the user is not logged in. How can we use the JWT to identify the user and add it into the request context?

In Lab 2, we set up the Apollo Server by providing the executable schema and the context, which has been the request object until now. We have to check if the JWT is inside the request. If this is the case, we need to verify it and query the user to see if the token is valid. Let's start by verifying the authorization header. Before doing so, import the new dependencies in the Graphql [index.js] file:

```
import JWT from 'jsonwebtoken';
const { JWT_SECRET } = process.env;
```

The [context] field of the [ApolloServer] initialization has to look as follows:

```
const server = new ApolloServer({
  schema: executableSchema,
  context: async ({ req }) => {
    const authorization = req.headers.authorization;
    if(typeof authorization !== typeof undefined) {
      var search = "Bearer";
      var regEx = new RegExp(search, "ig");
      const token = authorization.replace(regEx, '').trim();
      return jwt.verify(token, JWT_SECRET, function(err, result) {
        return req;
      });
    } else {
      return req;
    }
  },
});
```

We have extended the [context] property of the [ApolloServer] class to a full-featured function. We read the [auth] token from the headers of the requests. If the [auth] token exists, we need to strip out the bearer string, because it is not part of the original token that was created by our back end. The bearer token is the best method of JWT authentication.

ProTip

There are other authentication methods like basic authentication, but the bearer method is the best to follow. You can find a detailed explanation under RFC6750 by the IETF at this link: https://tools.ietf.org/html/rfc6750.

Afterwards, we use the [jwt.verify] function to check if the token matches the signature generated by the secret from the environment variables. The next step is to retrieve the user after successful verification. Replace the content of the [verify] callback with the following code:

```
if(err) {
  return req;
} else {
  return utils.db.models.User.findById(result.id).then((user) => {
    return Object.assign({}, req, { user });
  });
}
```

If the `err` object in the previous code has been filled, we can only return the ordinary request object, which triggers an error when it reaches the `AuthDirective` class, since there is no user attached. If there are no errors, we can use the `utils` object we are already passing to the Apollo Server setup to access the database.

After querying the user, we add them to the request object and return the merged user and request object as the context. This leads to a successful response from our authorizing directive.

You can now test this behavior. Start the front end with `npm run client` and the back end using `npm run server`. Don't forget that all Postman requests now have to include a valid JWT if the `auth` directive is used in the GraphQL query. You can run the login mutation and copy it over to the authorization header to run any query. We are now able to mark any query or mutation with the authorization flag and, as a result, require the user to be logged in.

Accessing the user context from resolver functions

At the moment, all the API functions of our GraphQL server allow us to simulate the user by selecting the first available one from the database. As we have just introduced a full-fledged authentication, we can now access the user from the request context. This section quickly explains how to do this for the chat and message entities. We also implement a new query called [currentUser], where we retrieve the logged-in user in our client.

Chats and messages

First of all, you have to add the [@auth] directive to the chats inside GraphQL's [RootQuery] to ensure that users need to be logged in to access any chats or messages.

Take a look at the resolver function for the chats. Currently, we use the [findAll] method to get all users, take the first one, and query for all chats of that user. Replace the code with the new resolver function:

```
chats(root, args, context) {
  return Chat.findAll({
    include: [{
      model: User,
      required: true,
      through: { where: { userId: context.user.id } },
    },
    {
      model: Message,
    }],
  });
},
```

We skip the retrieval of the user and directly insert the user ID from the context, as you can see in the preceding code. That's all we have to do: all chats and messages belonging to the logged-in user are queried directly from the chats table.

CurrentUser GraphQL query

The JWT gives us the opportunity to query for the currently logged-in user. Then, we can display the correct authenticated user in the top bar. To request the logged-in user, we require a new query called [currentUser] on our back end. In the schema, you simply have to add the following line to the [RootQuery] queries:

```
currentUser: User @auth
```

Like the [postsFeed] and [chats] queries, we also need the [@auth] directive to extract the user from the request context.

Similarly, in the resolver functions, you only need to insert the following three lines:

```
currentUser(root, args, context) {
  return context.user;
},
```

We return the user from the context right away, because it is already a user model instance with all the appropriate data returned by Sequelize. On the client side, we create this query in a separate component and file. Bear in mind that you don't need to pass the result on to all the children because this is done automatically by [ApolloConsumer] later on. You can follow the previous query component examples. Just use the following query, and you are good to continue:

```
const GET_CURRENT_USER = gql`
  query currentUser {
    currentUser {
      id
      username
      avatar
    }
  }
`;
```

You can now import the new query component inside the [App.js] file. Replace the old [div] tag within the logged-in state with the following code:

```
<CurrentUserQuery>
  <Bar />
  <Feed />
  <Chats />
</CurrentUserQuery>
```

Now, every time the [loggedIn] state variable is true, the [CurrentUserQuery] component is mounted and the query is executed. To get access to the response, we use the [ApolloConsumer] in the bar component that we implemented in the previous lab. I have surrounded the feed and the chats with the [currentUser] query to be sure that the user already exists in the local cache of the Apollo Client before the other components are rendered, including the bar component.

We have to adjust the [user.js] context file. First, we parse the [currentUser] query. The query is only needed to extract the user from the cache. It is not used to trigger a separate request. Insert the following code at the top of the [user.js] file:

```
import gql from 'graphql-tag';
const GET_CURRENT_USER = gql`
  query currentUser {
    currentUser {
      id
      username
      avatar
    }
  }
`;
```

Instead of having a hardcoded fake user inside [ApolloConsumer], we use the [client.readQuery] function to extract the data stored in the [ApolloClient] cache to give it to the underlying child component:

```
{client => {
  const {currentUser} = client.readQuery({ query: GET_CURRENT_USER});
  return React.Children.map(children, function(child){
    return React.cloneElement(child, { user: currentUser });
  });
}}
```

We pass the extracted [currentUser] result from the [client.readQuery] method directly to all the wrapped children of the current component.

The chats that are created from now on and the user in the top bar are no longer faked but instead are filled with the user who is currently logged in.

The mutations to create new posts or messages still use a static user id. We can switch over to the real logged-in user in the same way as we did previously in this section by using the user id from the [context.user] object. You should now be able to do this on your own.

Logging out using React

To complete the circle, we still have to implement the functionality to log out. There are two cases when the user can be logged out:

- The user wants to log out and hits the logout button
- The JWT has expired after one day as specified; the user is no longer authenticated, and we have to set the state to logged out

We will begin by adding a new logout button to the top bar of our application's front end. To do this, we need to create a new [logout.js] component inside the [bar] folder. It should appear as follows:

```
import React, { Component } from 'react';
import { withApollo } from 'react-apollo';

class Logout extends Component {
  logout = () => {
    localStorage.removeItem('jwt');
    this.props.changeLoginState(false);
    this.props.client.resetStore();
  }
  render() {
    return (
      <button className="logout" onClick={this.logout}>Logout</button>
    );
  }
}

export default withApollo(Logout);
```

As you can see from the preceding code, the logout button triggers the component's logout method when it is clicked. Inside the [logout] method, we remove the JWT from [localStorage] and execute the [changeLoginState] function that we receive from the parent component. Be aware that we do not send a request to our server to log out, but instead we remove the token from the client. That is because there is no black or white list that we are using to disallow or allow a certain JWT to authenticate on our server. The easiest way to log out a user is to remove the token on the client side so that neither the server nor the client has it.

We also reset the client cache. When a user logs out, we must remove all data. Otherwise, other users of the same browser will be able to extract all the data, which we have to prevent. To get access to the underlying Apollo Client, we import the [withApollo] HoC and export the [Logout] component wrapped inside it. When logging out, we execute the [client.resetStore] function and all data is deleted. To use our new [Logout] component, open the [index.js] file from the [bar] folder and import it at the top. We can render it within the [div] top bar, below the other inner [div] tag:

```
<Logout changeLoginState={this.props.changeLoginState}/>
```

We pass the [changeLoginState] function to the [Logout] component. In the [App.js] main file, you have to ensure that you hand over this function not only to the [LoginRegisterForm] but also to the bar component, as follows:

```
<Bar changeLoginState={this.changeLoginState}/>
```

If you copy the complete CSS from the official GitHub repository, you should see a new button at the top-right corner of the screen when you are logged in. Hitting it logs you out and requires you to sign in again since the JWT has been deleted.

The other situation in which we implement a logout functionality is when the JWT we are using expires. In this case, we log the user out automatically and require them to log in again. Go to the [App] class and add the following lines:

```
constructor(props) {
  super(props);
  this.unsubscribe = props.client.onResetStore(
    () => this.changeLoginState(false)
  );
}
componentWillUnmount() {
  this.unsubscribe();
}
```

We need a [constructor] because we are using the [client.onResetStore] event, which is caught through the [client.onResetStore] function.

To get the preceding code working, we have to access the Apollo Client in our [App] component. The easiest way is to use the [withApollo] HoC. Just import it from the [react-apollo] package in the [App.js] file:

```
import { withApollo } from 'react-apollo';
```

Then, export the [App] class---not directly, but through the HoC. The following code must go directly beneath the [App] class:

```
export default withApollo(App);
```

Now, the component can access the client through its properties. The [resetStore] event is thrown whenever the client restore is reset, as the name suggests. You are going to see why we need this shortly. When listening to events in React, we have to stop listening when the component is unmounted. We handle this inside the [componentWillUnmount] function in the preceding code. Now, we have to reset the client store to initiate the logout state. When the event is caught, we execute the [changeLoginState] function automatically. Consequently, we could remove the section in which we passed the [changeLoginState] to the logout button initially because it is no longer needed, but this not what we want to do here.

Instead, go to the [index.js] file in the [apollo] folder. There, we already catch and loop over all errors returned from our GraphQL API. What we do now is loop over all errors but check each of them for an [UNAUTHENTICATED] error. Then, we execute the [client.resetStore] function. Insert the following code into the Apollo Client setup:

```
onError(({ graphQLErrors, networkError }) => {
  if (graphQLErrors) {
    graphQLErrors.map(({ message, locations, path, extensions }) => {
      if(extensions.code === 'UNAUTHENTICATED') {
        localStorage.removeItem('jwt');
        client.resetStore()
      }
      console.log(`[GraphQL error]: Message: ${message}, Location:
      ${locations}, Path: ${path}`);
    });
    if (networkError) {
      console.log(`[Network error]: ${networkError}`);
    }
  }
}),
```

As you can see, we access the [extensions] property of the error. The [extensions.code] field holds the specific error type that's returned. If we are not logged in, we remove the JWT and then reset the store. By doing this, we trigger the event in our [App] class, which sends the user back to the login form.

A further extension would be to offer a refresh token API function. The feature could be run every time we successfully use the API. The problem with this is that the user would stay logged in forever, as long as they are using the application. Usually, this is no problem, but if someone else is accessing the same computer, they will be authenticated as the original user. There are different ways to implement these kinds of functionalities to make the user experience more comfortable, but I am not a big fan of these for security reasons.

Summary

Until now, one of the main issues we had with our application is that we didn't have any authentication. We can now tell who is logged in every time a user accesses our application. This allows us to secure the GraphQL API and insert new posts or messages in the name of the correct user. In this lab, we discussed the fundamental aspects of JSON Web Tokens, [localStorage], and cookies. We also looked at how the verification of hashed passwords or signed tokens works. This lab then covered how to implement JWTs inside React and how to trigger the correct events to log in and log out.

In the next lab, we are going to implement image uploads with a reusable component that allows the user to upload new avatar images.