Lab 9: Implementing Server-Side Rendering

This lab covers the following topics:

- An introduction to server-side rendering
- Setting up Express.js to render React on the server
- Enabling JWT authentication in connection with server-side rendering
- Running all GraphQL queries in the React tree

Let's get started with writing some code.

Lab Solution

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

```
cd ~/Desktop/react-graphql-course/labs/Lab09
```

SSR in Express.js

The first step is to implement basic server-side rendering on the back end. We are going to extend this functionality later to validate the authentication of the user. An authenticated user allows us to execute Apollo or GraphQL requests, and not only to render the pure React markup. First, we need some new packages. Because we are going to use universal rendered React code, we require an advanced webpack configuration; hence, we will install the following packages:

```
npm install --save-dev webpack-dev-middleware webpack-hot-middleware @babel/cli
```

For development with SSR enabled, the back end uses these packages to distribute the bundled React code to the client, after the server-side rendering has finished. The server itself relies on the plain [src] files, and not on the webpack bundle that the client receives.

We also depend on one further essential package, as follows:

```
npm install --save node-fetch
```

To set up the Apollo Client on the back end, we require a replacement of the standard [window.fetch] method. The Apollo Client uses it to send GraphQL requests, which is why we install [node-fetch] as a polyfill. We are going to set up the Apollo Client for the back end later in this lab.

Before starting with the primary work, ensure that your [NODE_ENV] environment variable is set to [development].

Head over to the server's [index.js] file, where all of the Express magic happens. We didn't cover this file in the previous lab, because we are going to adjust it now to support server-side including the routing directly.

First, we will set up the development environment for server-side rendering, as it is essential for the next tasks. Follow these steps to get your development environment ready for SSR:

1. The first step is to import the two new webpack modules: [webpack-dev-middleware] and [webpack-hot-middleware]. These should only be used in a development environment, so we should require them conditionally, by checking the environment variables. In a production environment, we generate the webpack bundles in advance. Put the following code underneath the setup for the Express.js helmet, in order to only use the new packages in development:

```
if(process.env.NODE_ENV === 'development') {
  const devMiddleware = require('webpack-dev-middleware');
  const hotMiddleware = require('webpack-hot-middleware');
  const webpack = require('webpack');
  const config = require('../../webpack.server.config');
  const compiler = webpack(config);
  app.use(devMiddleware(compiler));
  app.use(hotMiddleware(compiler));
}
```

2. After loading those packages, we will also require webpack, because we will parse a new webpack configuration file. The new configuration file is only used for the server-side rendering.
3. After both the webpack and the configuration file have been loaded, we will use the [webpack(config)] command to parse the configuration and create a new webpack instance.
4. We are going to create the webpack configuration file next. We pass the created webpack instance to our two new modules. When a request reaches the server, the two packages take action according to the configuration file.

The new configuration file has only a few small differences, as compared to the original configuration file, but these have a big impact. Create the new [webpack.server.config.js] file, and enter the following configuration:

```
const path = require('path');
const webpack = require('webpack');
const buildDirectory = 'dist';
module.exports = {
  mode: 'development',
  entry: [
    'webpack-hot-middleware/client',
    './src/client/index.js'
  ],
  output: {
    path: path.join(__dirname, buildDirectory),
    filename: 'bundle.js',
    publicPath: '/'
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
        },
      },
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader'],
      },
      {
        test: /\.(png|woff|woff2|eot|ttf|svg)$/,
        loader: 'url-loader?limit=100000',

      },
```

```
    ],
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin(),
    new webpack.NamedModulesPlugin(),
  ],
};
```

We have made three changes in the preceding configuration, in comparison to the original [webpack.client.config.js], as follows:

- In the [entry] property, we now have multiple entry points. The [index] file for the front end code, like before, is one entry point. The second one is the new [webpack-hot-middleware] module, which initiates the connection between the client and the server. The connection is used to send the client notifications to update the bundle to a newer version.
- I removed the [devServer] field, as this configuration does not require webpack to start its own server. Express.js is the web server, which we are already using when loading the configuration.
- The plugins are entirely different from those of the client's webpack configuration. We do not need the [CleanWebpackPlugin] plugin, as this cleans the [dist] folder, nor the [HtmlWebpackPlugin], which inserts the webpack bundles into the [index.html] file; this is handled by the server differently. These plugins are only useful for client-side development. Now, we have the [HotModuleReplacementPlugin] plugin, which enables **Hot Module Replace** (**HMR**). It allows for JS and CSS to be exchanged on the fly. [NamedModulesPlugin] displays the relative paths for modules injected by HMR. Both plugins are only recommended for developmental use.

The webpack preparation is now finished.

Now, we have to focus on how to render React code, and how to serve the generated HTML. However, we cannot use the complete React code that we have written. There are specific adjustments that we have to make to the main files: [index.js], [App.js], [router.js], and [apollo/index.js]. Many packages that we use, such as React Router or Apollo Client, have default settings or modules that we have to configure differently when executed on the server.

We will begin with the root of our React application, which is the [index.js] file. We are going to implement an individual SSR [index] file, as there are server-specific adjustments to do.

Create a new folder, called [ssr], inside the [server] folder. Insert the following code into an [index.js] file inside the [ssr] folder:

```
import React from 'react';
import { ApolloProvider } from 'react-apollo';
import App from './app';

export default class ServerClient extends React.Component {
  render() {
    const { client, location, context } = this.props;
    return(
      <ApolloProvider client={client}>
        <App location={location} context={context}/>
      </ApolloProvider>
    );
  }
}
```

The preceding code is a modified version of our client [index.js] root file. The changes that the file has gone through are listed as follows:

- Instead of using the [ReactDOM.render] function to insert the HTML into the DOMNode with the id [root], we are now exporting a React component. The returned component is called [ServerClient]. There is no DOM that we can access to let ReactDOM render anything, so we skip this step when rendering on the server.
- Also, the [ApolloProvider] component now receives the Apollo Client directly from the [ServerClient] properties, whereas we previously set up the Apollo Client directly inside this file by importing the [index.js] file from the [apollo] folder and passing it to the provider. You will soon see why we are doing this.
- The last change that we made was to extract a [location] and a [context] property. We pass these properties to the [App] component. In the original version, there were no properties passed to the [App] component. Both properties are required in order to configure React Router to work with SSR. We are going to implement the properties later in the lab.

Before looking at why we made these changes in more detail, let's create the new [App] component for the back end. Create an [app.js] file next to the [index.js] file in the [ssr] folder, and insert the following code:

```
import React, { Component } from 'react';
import { Helmet } from 'react-helmet';
import { withApollo } from 'react-apollo';
import '../../client/components/fontawesome';
import Router from '../../client/router';

class App extends Component {
  state = {
    loggedIn: false
  }
  changeLoginState = (loggedIn) => {
    this.setState({ loggedIn });
  }
  render() {
    return (
      <div>
        <Helmet>
          <title>Graphbook - Feed</title>
          <meta name="description" content="Newsfeed of all your
           friends on Graphbook" />
        </Helmet>
        <Router loggedIn={this.state.loggedIn} changeLoginState=
         {this.changeLoginState} location={this.props.location}
         context={this.props.context}/>
      </div>
    )
  }
}

export default withApollo(App)
```

The following are a few changes that we made:

- The first change, in comparison to the original client-side [App] class, was to adjust the [import] statements to load the router and the [fontawesome] component from the [client] folder, as they do not exist in the [server] folder.
- The second change was to remove the [constructor], the [componentWillMount], and the [componentWillUnmount] methods. We did this because the authentication that we built uses the

[localStorage]. It is fine for client-side authentication. Neither Node.js nor the server support such storage, in general. That is the reason why we remove the authentication when moving our application to server-side rendering. We are going to replace the [localStorage] implementation with cookies in a later step. For the moment, the user stays logged out of the server.

- The last change involves passing the two new properties, [context] and [location], to the [Router] in the preceding code.

React Router provides instant support for SSR. Nevertheless, we need to make some adjustments. The best is that we use the same router for the back end and front end, so that we do not need to define routes twice, which is inefficient and can lead to problems. Open the [router.js] inside the [client] folder and follow these steps:

1. Delete the [import] statement for the [react-router-dom] package.
2. Insert the following code to import the package properly:

```
const ReactRouter = require("react-router-dom");
let Router;
if(typeof window !== typeof undefined) {
  const { BrowserRouter } = ReactRouter;
  Router = BrowserRouter;
}
else {
  const { StaticRouter } = ReactRouter;
  Router = StaticRouter;
}
const { Route, Redirect, Switch } = ReactRouter;
```

We use the [require] statement in the preceding code. The reason is that [import] statements are statically analyzed and do not allow for conditional extracting of the package's modules. Notice that after requiring the React Router package, we check whether the file is executed on the server or the client by looking for the [window] object. Since there is no [window] object in Node.js, this is a sufficient check. An alternative approach would be to set up the [Switch] component, including the routes, in a separate file. This approach would allow us to import the routes directly into the correct router, if we create two separate router files for client-side and server-side rendering.

If we are on the client-side, we use the [BrowserRouter], and if not, we use the [StaticRouter]. The logic is that with the [StaticRouter], we are in a stateless environment, where we render all routes with a fixed location. The [StaticRouter] does not allow for the location to be changed by redirects, since no user interaction can happen when using server-side rendering. The other components, [Route], [Redirect], and [Switch], can be used as before.

No matter which of the routers is extracted, we save them in the [Router] variable. We then use them in the [render] method of the [Routing] class.

3. We prepared the properties [context] and [location], which are passed from the top [ServerClient] component to the [Router]. If we are on the server, these properties should be filled, because the [StaticRouter] requires them. You can replace the [Router] tag in the bottom [Routing] component, as follows:

```
<Router context={this.props.context} location={this.props.location}>
```

The [location] holds the path that the router should render. The [context] variable stores all of the information the [Router] processes, such as redirects. We can inspect this variable after rendering the [Router] to trigger the redirects manually. This behavior is the big difference between the [BrowserRouter] and the [StaticRouter]. The [BrowserRouter] redirects the user automatically, but the [StaticRouter] does not.

The crucial components to render our React code successfully have now been prepared. However, there are still some modules that we have to initialize before rendering anything with React. Open the [index.js] server file again. At

the moment, we are serving the [dist] path statically on the root [/] path for client-side rendering, which can be found at [http://localhost:8000]. When moving to SSR, we have to serve the HTML generated by our React application at the [/] path instead.

Furthermore, any other path, such as [/app], should also use SSR to render those paths on the server. Remove the current [app.get] method at the bottom of the file, right before the [app.listen] method. Insert the following code as a replacement:

```
app.get('*', (req, res) => {
  res.status(200);
  res.send(`<!doctype html>`);
  res.end();
});
```

The asterisk that we are using in the preceding code can overwrite any path that is defined later in the Express routing. Always remember that the [services] routine that we use in Express can implement new paths, such as [/graphql], that we do not want to overwrite. To avoid this, put the code at the bottom of the file, below the [services] setup. The route catches any requests sent to the back end.

You can try out this route by running the [npm run server] command. Just visit [http://localhost:8000].

Currently, the preceding catch-all route only returns an empty site, with a status of [200]. Let's change this. The logical step would be to load and render the [ServerClient] class from the [index.js] file of the [ssr] folder, since it is the starting point of the React SSR code. The [ServerClient] component, however, requires an initialized Apollo Client, as we explained before. We are going to create a special Apollo Client for SSR next.

Create a [ssr/apollo.js] file, as it does not exist yet. We will set up the Apollo Client in this file. The content is nearly the same as the original setup for the client:

```
import { ApolloClient } from 'apollo-client';
import { InMemoryCache } from 'apollo-cache-inmemory';
import { onError } from 'apollo-link-error';
import { ApolloLink } from 'apollo-link';
import { HttpLink } from 'apollo-link-http';
import fetch from 'node-fetch';

export default (req) => {
  const AuthLink = (operation, next) => {
    return next(operation);
  };
  const client = new ApolloClient({
    ssrMode: true,
    link: ApolloLink.from([
      onError(({ graphQLErrors, networkError }) => {
        if (graphQLErrors) {
          graphQLErrors.map(({ message, locations, path, extensions })
          => {
            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',
        fetch
      })
    ]),
    cache: new InMemoryCache(),
  });
  return client;
};
```

There are a few changes that we made to get the client working on the server. These changes were pretty big, so we created a separate file for the server-side Apollo Client setup. Take a look at the changes, as follows, to understand the differences between the front end and the SSR setup for the Apollo Client:

- Instead of using the [createUploadLink] function that we introduced to allow the user to upload images or other files, we are now using the standard [HttpLink] again. You could have used the [UploadClient], but the functionalities that it provides won't be used on the server, as the server won't upload files (of course).
- The [AuthLink] skips to the next link, as we have not implemented server-side authentication yet.
- The [HttpLink] receives the [fetch] property, which is filled by the [node-fetch] package that we installed at the beginning of the lab. It is used instead of the [window.fetch] method, which is not available in Node.js.
- Rather than exporting the [client] directly, we export a wrapping function that accepts a [request] object. We pass it as a parameter in the Express route. As you can see in the preceding code, we haven't used the object yet, but that will change soon.

Import the [ApolloClient] class at the top of the server [index.js], as follows:

```
import ApolloClient from './ssr/apollo';
```

The imported [ApolloClient] function accepts the [request] object of our Express server.

Add the following line to the top of the new Express catch-all route:

```
const client = ApolloClient(req);
```

This way, we set up a new [client] instance that we can hand over to our [ServerClient] component.

We can continue and implement the rendering of our [ServerClient] component. To make the future code work, we have to load React and, of course, the [ServerClient] itself:

```
import React from 'react';
import Graphbook from './ssr/';
```

The [ServerClient] class is imported under the [Graphbook] name. We import React because we use the standard JSX syntax while rendering our React code.

Now that we have access to the Apollo Client and the [ServerClient] component, insert the following two lines below the [ApolloClient] setup in the Express route:

```
const context= {};
const App = (<Graphbook client={client} location={req.url} context=
 {context}/>);
```

We pass the initialized [client] variable to the [Graphbook] component. We use the regular React syntax to pass all properties. Furthermore, we set the [location] property to the request object's [url], to tell the router which path to render. The [context] property is passed as an empty object.

However, why do we pass an empty object as [context] to the Router at the end?

The reason is that after rendering the [Graphbook] component to HTML, we can access the [context] object and see whether a redirect, or something else, would have been triggered regularly. As we mentioned before, redirects have to be implemented by the back end code. The [StaticRouter] component of React Router does not make assumptions about the Node.js web server that you are using. That is why the [StaticRouter] does not execute them automatically. Tracking and post-processing these events is possible with the [context] variable.

The resulting React object is saved to a new variable, called [App]. Now, there should be no errors if you start the server with [npm run server] and visit [http://localhost:8000]. Still, we see an empty page. That happens because we only return an empty HTML page; we haven't rendered the React [App] object to HTML. To render the object to HTML, import the following package at the top of the server [index.js] file:

```
import ReactDOM from 'react-dom/server';
```

The [react-dom] package not only provides bindings for the browser, but also provides a special module for the server, which is why we use the suffix [/server] while importing it. The returned module provides a number of server-only functions.

We can translate the React [App] object into HTML by using the [ReactDOM.rendertoString] function. Insert the following line of code beneath the [App] object:

```
const content = ReactDOM.renderToString(App);
```

This function generates HTML and stores it inside the [content] variable. It can be returned to the client now. If you return pre-rendered HTML from the server, the client goes through it and checks whether its current state would match the returned HTML. The comparison is made by identifying certain points in the HTML, such as the [data-reactroot] property.

If, at any point, the markup between the server-rendered HTML and the one that the client would generate does not match, an error is thrown. The application will still work, but the client will not be able to make use of server-side rendering; the client will replace the complete markup returned from the server by rerendering everything again. The server's HTML response is thrown away in this case. This is, of course, very inefficient and not what we are aiming for.

We have to return the rendered HTML to the client. The HTML that we have rendered begins with the root [div] tag, and not the [html] tag. We must wrap the [content] variable inside a template, which includes the surrounding HTML tags. Create a [template.js] file, inside the [ssr] folder. Enter the following code to implement the template for our rendered HTML:

```
import React from 'react';
import ReactDOM from 'react-dom/server';

export default function htmlTemplate(content) {
  return `
    <html lang="en">
      <head>
        <meta charSet="UTF-8"/>
        <meta name="viewport" content="width=device-width, initial-
         scale=1.0"/>
        <meta httpEquiv="X-UA-Compatible" content="ie=edge"/>
        <link rel="shortcut icon" href="data:image/x-icon;," type="image/x-
```

```
         icon">
        ${(process.env.NODE_ENV === 'development')? "":"<link
           rel='stylesheet' href='/bundle.css'/>"}
      </head>
      <body>
        ${ReactDOM.renderToStaticMarkup(<div id="root"
          dangerouslySetInnerHTML={{ __html: content }}></div>)}
        <script src="/bundle.js"></script>
      </body>
    </html>
  `;
};
```

The preceding code is pretty much the same HTML markup as that in the [index.html] that we usually serve to the client. The difference is that we use React and [ReactDOM] here.

First, we export a function, which accepts the [content] variable with the rendered HTML.

Secondly, we render a [link] tag inside the [head] tag, which downloads the CSS bundle if we are in a production environment. For our current development scenario, there is no bundled CSS.

The important part is that we use a new [ReactDOM] function called [rendertoStaticMarkup] inside the [body] tag. This function inserts the React root tag into the body of our HTML template. Before, we used the [renderToString] method, which included special React tags, such as the [data-reactroot] property. We use the [rendertoStaticMarkup] function to generate standard HTML, without special React tags. The only parameter that we pass to the function is the [div] tag with the id [root] and a new property, [dangerouslySetInnerHTML]. This attribute is a replacement for the regular [innerHTML] attribute, but for use in React. It lets React insert the HTML inside the root [div] tag. As the name suggests, it is dangerous to do this, but only if it is done on the client, as there is no possibility for XSS attacks on the server. We use the [ReactDOM.renderToStaticMarkup] function to make use of the attribute. The inserted HTML was initially rendered with the [renderToString] function, so that it would include all critical React HTML attributes and the wrapping [div] tag with the id [root]. It can then be reused in the browser by the front end code without any problems.

Require this [template.js] file in the server [index] file, at the top of the file:

```
import template from './ssr/template';
```

The template function can now be used directly in the [res.send] method, as follows:

```
res.send(`<!doctype html>\n${template(content)}`);
```

We do not only return a [doctype] anymore; we also respond with the return value of the [template] function. As you should see, the [template] function accepts the rendered [content] variable as a parameter, and composes it to a valid HTML document.

At this point, we have managed to get our first version of a server-side rendered React application working. You can prove this by right-clicking in your browser window and choosing to view the source code. The window shows you the original HTML that is returned by the server. The output equals the HTML from the [template] function, including the login and signup forms.

Nevertheless, there are two problems that we face, as follows:

- There is no description meta [head] tag included in the server response. Something must have gone wrong with React Helmet.

- When logged in on the client side and, for example, viewing the news feed under the [/app] path, the server responds without having rendered the news feed, nor the login form. Normally, React Router would have redirected us to the login form, since we are not logged in on the server side. Since we use the [StaticRouter], however, we have to initiate the redirect separately, as we explained before. We are going to implement the authentication in a separate step.

We will start with the first issue. To fix the problem with React [Helmet], import it at the top of the server [index.js] file, as follows:

```
import { Helmet } from 'react-helmet';
```

Now, before setting the response status with [res.status], you can extract the React [Helmet] status, as follows:

```
const head = Helmet.renderStatic();
```

The [renderStatic] method is specially made for server-side rendering. We can use it after having rendered the React application with the [renderToString] function. It gives us all [head] tags that would have been inserted throughout our code. Pass this [head] variable to the [template] function as a second parameter, as follows:

```
res.send(`<!doctype html>\n${template(content, head)}`);
```

Go back to the [template.js] from the [ssr] folder. Add the [head] parameter to the exported function's signature. Add the following two new lines of code to the HTML's [head] tag:

```
${head.title.toString()}
${head.meta.toString()}
```

The [head] variable extracted from React Helmet holds a property for each [meta] tag. They provide a [toString] function that returns a valid HTML tag, which you can directly enter into the document's [head]. The first problem should be fixed: all [head] tags are now inside the server's HTML response.

Let's focus on the second problem. The server response returns an empty React [root] tag when visiting a [PrivateRoute]. As we explained previously, the reason is that the naturally initiated redirect does not get through to us, since we are using the [StaticRouter]. We are redirected away from the [PrivateRoute], because the authentication is not implemented for the server-rendered code. The first thing to fix is to handle the redirect, and at least respond with the login form, instead of an empty React [root] tag. Later, we need to fix the authentication problem.

You would not notice the problem without viewing the source code of the server's response. The front end downloads the [bundle.js] and triggers the rendering on its own, as it knows about the authentication status of the user. The user would not notice that. Still, it is more efficient if the server sends the correct HTML directly. The HTML will be wrong if the user is logged in, but in the case of an unauthenticated user, the login form is pre-rendered by the server as it initiates the redirects.

To fix this issue, we can access the [context] object that has been filled by React Router after it has used the [renderToString] function. The final Express route should look as follows:

```
app.get('*', (req, res) => {
  const client = ApolloClient(req);
  const context= {};
  const App = (<Graphbook client={client} location={req.url} context=
  {context}/>);
  const content = ReactDOM.renderToString(App);
  if (context.url) {
    res.redirect(301, context.url);
  } else {
```

```
    const head = Helmet.renderStatic();
    res.status(200);
    res.send(`<!doctype html>\n${template(content, head)}`);
    res.end();
  }
});
```

The condition for rendering the correct route on the server is that we inspect the [context.url] property. If it is filled, we can initiate a redirect with Express.js. That will navigate the browser to the correct path. If the property is not filled, we can return the HTML generated by React.

This route renders the React code correctly, up to the point at which authentication is required. The SSR route correctly renders all public routes, but none of the secure routes. That means that we only respond with the login form at the moment, since it is the only route that doesn't require authentication.

The next step is to implement authentication in connection with SSR, in order to fix this huge issue.

Authentication with SSR

You should have noticed that we have removed most of the authentication logic from the server-side React code. The reason is that the [localStorage] cannot be transmitted to the server on the initial loading of a page, which is the only case where SSR can be used at all. This leads to the problem that we cannot render the correct route, because we cannot verify whether a user is logged in. The authentication has to be transitioned to cookies, which are sent with every request.

It is important to understand that cookies also introduce some security issues. We will continue to use the regular HTTP authorization header for the GraphQL API that we have written. If we use cookies for the GraphQL API, we will expose our application to potential CSRF attacks. The front end code continues to send all GraphQL requests with the HTTP authorization header.

We will only use the cookies to verify the authentication status of a user, and to initiate requests to our GraphQL API for server-side rendering of the React code. The SSR GraphQL requests will include the authorization cookie's value in the HTTP authorization header. Our GraphQL API only reads and verifies this header, and does not accept cookies. As long as you do not mutate data when loading a page and only query for data to render, there will be no security issues.

ProTip

As the whole topic of CSRF and XSS is big, I recommend that you read up on it, in order to fully understand how to protect yourself and your users. You can find a great article at https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

The first thing to do is install a new package with [npm], as follows:

```
npm install --save cookies
```

The [cookies] package allows us to easily interact through the Express request object with the cookies sent by the browser. Instead of parsing and reading through the cookie string (which is just a comma-separated list) manually, you can access the cookies with simple [get] and [set] methods. To get this package working, you have to initialize it inside Express.

Import the [cookies] and [jwt] packages, and also extract the [JWT_SECRET] from the environment variables at the top of the server [index.js] file:

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

To use the [cookies] package, we are going to set up a new middleware route. Insert the following code before initializing the webpack modules and the services routine:

```
app.use(
  (req, res, next) => {
    const options = { keys: ['Some random keys'] };
    req.cookies = new Cookies(req, res, options);
    next();
  }
);
```

This new Express middleware initializes the [cookies] package under the [req.cookies] property for every request that it processes. The first parameter of the [Cookies] constructor is the request, the second is the response object, and the last one is an [options] parameter. It takes an array of [keys], with which the cookies are signed. The [keys] are required if you want to sign your cookies for security reasons. You should take care of this in a production environment. You can specify a [secure] property, which ensures that the cookies are only transmitted on secure HTTPS connections.

We can now extract the [authorization] cookie and verify the authentication of the user. To do this, replace the beginning of the SSR route with the following code in the server's [index.js] file:

```
app.get('*', async (req, res) => {
  const token = req.cookies.get('authorization', { signed: true });
  var loggedIn;
  try {
    await JWT.verify(token, JWT_SECRET);
    loggedIn = true;
  } catch(e) {
    loggedIn = false;
  }
```

Here, I have added the [async] declaration to the callback function, because we use the [await] statement inside it. The second step is to extract the [authorization] cookie from the request object with [req.cookies.get]. Importantly, we specify the [signed] field in the [options] parameter, because only then will it successfully return the signed cookies.

The extracted value represents the JWT that we generate when a user logs in. We can verify this with the typical approach that we implemented in Lab 6, *Authentication with Apollo and React*. We use the [await] statement while verifying the JWT. If an error is thrown, the user is not logged in. The state is saved in the [loggedIn] variable. Pass the [loggedIn] variable to the [Graphbook] component, as follows:

```
const App = (<Graphbook client={client} loggedIn={loggedIn} location={req.url}
context={context}/>);
```

Now, we can access the [loggedIn] property inside [index.js] from the [ssr] folder. Extract the [loggedIn] sate from the properties, and pass it to the [App] component in the [ssr] [index.js] file, as follows:

```
<App location={location} context={context} loggedIn={loggedIn}/>
```

Inside the [App] component, we do not need to set the [loggedIn] state directly to false, but we can take the property's value, because it is determined before the [App] class is rendered. This flow is different from the client procedure, where the [loggedIn] state is determined inside the [App] class. Change the [App] class in the [app.js] file in order to match the following code:

```
class App extends Component {
  state = {
    loggedIn: this.props.loggedIn
  }
```

The result is that we pass down the [loggedIn] value from our Express.js route, over the [Graphbook] and [App] components, to our [Router]. It already accepts the [loggedIn] property, in order to render the correct path for the user. At the moment, we still do not set the cookie on the back end when a user successfully logs in.

Open the [resolvers.js] file of our GraphQL server to fix that. We will change a few lines for the [login] and [signup] functions. Both resolver functions need the same changes, as both set the authentication token after login or signup. Insert the following code directly above the return statement:

```
context.cookies.set(
  'authorization',
  token, { signed: true, expires: expirationDate, httpOnly: true,
  secure: false, sameSite: 'strict' }
);
```

The preceding function sets the cookies for the user's browser. The context object is only the Express.js request object where we have initialized the cookies package. The properties of the [cookies.set] function are pretty self-explanatory, as follows:

- The [signed] field specifies whether the keys entered during the initialization of the [cookies] object should be used to sign the cookie's value.
- The [expires] property takes a [date] object. It represents the time until which the cookie is valid. You can set the property to whatever date you want, but I would recommend a short period, such as one day. Insert the following code above the [context.cookies.set] statement, in order to initialize the [expirationDate] variable correctly:

```
const cookieExpiration = 1;
var expirationDate = new Date();
expirationDate.setDate(
  expirationDate.getDate() + cookieExpiration
);
```

- The [httpOnly] field secures the cookie so that it is not accessible by client-side JavaScript.
- The [secure] property has the same meaning as it did when initializing the [Cookie] package. It restricts cookies to SSL connections only. This is a must when going to production, but it cannot be used while developing, since most developers develop locally, without an SSL certificate.
- The [sameSite] field takes either [strict] or [lax] as a value. I recommend setting it to [strict], since you only want your GraphQL API or server to receive the cookie with every request, but to exclude all cross-site requests, as this could be dangerous.

Now, we should clean up our code. Since we are using cookies, we can remove the [localStorage] authentication flow in the front end code. Open the [App.js] of the [client] folder. Remove the [componentWillMount] method, as we are reading from the [localStorage] there.

The cookies are automatically sent with any request, and they do not need a separate binding, like the [localStorage]. That also means that we need a special [logout] mutation that removes the cookie from the browser. JavaScript is not able to access or remove the cookie, because we specified it as [httpOnly]. Only the server can delete it from the client.

Create a new [logout.js] inside the [mutations] folder, in order to create the new [LogoutMutation] class. The content should look as follows:

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

const LOGOUT = gql`
  mutation logout {
    logout {
      success
    }
  }
`;

export default class LogoutMutation extends Component {
  render() {
    const { children } = this.props;
    return (
      <Mutation
        mutation={LOGOUT}>
        {(logout, { loading, error}) =>
          React.Children.map(children, function(child){
            return React.cloneElement(child, { logout, loading, error });
          })
        }
      </Mutation>
    )
  }
}
```

The preceding mutation component only sends a simple [logout] mutation, without any parameters or further logic. We should use the [LogoutMutation] component inside the [index.js] file of the [bar] folder in order to send the GraphQL request. Import the component at the top of the file, as follows:

```
import LogoutMutation from '../mutations/logout';
```

The [Logout] component renders our current [Log out] button in the application bar. It removes the token and cache from the client upon clicking it. Use the [LogoutMutation] class as a wrapper for the [Logout] component, to pass the mutation function:

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

Inside the [bar] folder, we have to edit the [logout.js] file, because we should make use of the [logout] mutation that this component receives from its parent [LogoutMutation] component. Replace the [logout] method with the following code, in order to send the mutation upon clicking the [logout] button:

```
logout = () => {
  this.props.logout().then(() => {
    localStorage.removeItem('jwt');
    this.props.client.resetStore();
  });
}
```

We have wrapped the original two functions inside the call to the parent [logout] mutation function. It sends the GraphQL request to our server.

To implement the mutation on the back end, add one line to the GraphQL [RootMutation] type, inside [schema.js]:

```
logout: Response @auth
```

It's required that the user that's trying to log out is authorized, so we use the [@auth] directive. The corresponding resolver function looks as follows. Add it to the [resolvers.js] file, in the [RootMutation] property:

```
logout(root, params, context) {
  context.cookies.set(
    'authorization',
    '', { signed: true, expires: new Date(), httpOnly: true, secure:
    false, sameSite: 'strict' }
  );
  return {
    message: true
  };
},
```

The resolver function is minimal. It removes the cookie by setting the expiration date to the current time. This removes the cookie on the client when the browser receives the response, because it is expired then. This behavior is an advantage, in comparison to the [localStorage].

We have completed everything to make the authorization work with SSR. It is a very complex task, since authorization, server-side rendering, and client-side rendering have effects on the whole application. Every framework out there has its own approach to implementing this feature, so please take a look at them too.

If you look at the source code returned from our server after the rendering, you should see that the login form is returned correctly, like before. Furthermore, the server now recognizes whether the user is logged in. However, the server does not return the rendered news feed, the application bar, or the chats yet. Only a loading message is included in the returned HTML. The client-side code also does not recognize that the user is logged in. We will take a look at these problems in the next section.

Running Apollo queries with SSR

By nature, GraphQL queries via [HttpLink] are asynchronous. We have implemented a [loading] component to show the user a loading message while the data is being fetched.

This is the same thing that is happening while rendering our React code on the server. All of the routing is evaluated, including whether we are logged in. If the correct route is found, all GraphQL requests are sent. The problem is that the first rendering of React returns the loading state, which is sent to the client by our server. The server does not wait until the GraphQL queries are finished and it has received all of the responses to render our React code.

We will fix this problem now. The following is a list of things that we have to do:

- We need to implement authentication for the SSR Apollo Client. We already did this for the routing, but now we need to pass the cookie to the server-side GraphQL request too.
- We need to use a React Apollo specific method to render the React code asynchronously, to wait for all responses of the GraphQL requests.
- Importantly, we need to return the Apollo cache state to the client. Otherwise, the client will re-fetch everything, as its state is empty upon the first load of the page.

Let's get started, as follows:

1. The first step is to pass the [loggedIn] variable from the Express.js SSR route to the [ApolloClient] function, as a second parameter. Change the [ApolloClient] call inside the server's [index.js] file to the following line of code:

```
const client = ApolloClient(req, loggedIn);
```

Change the signature of the exported function from the [apollo.js] file to also include this second parameter.

2. Replace the [AuthLink] function inside the Apollo Client's setup for SSR with the following code:

```
const AuthLink = (operation, next) => {
  if(loggedIn) {
    operation.setContext(context => ({
      ...context,
      headers: {
        ...context.headers,
        Authorization: req.cookies.get('authorization')
      },
    }));
  }
  return next(operation)
};
```

This [AuthLink] adds the cookies to the GraphQL requests by using the [request] object given by Express. The request object already holds the initialized cookies package, which we use to extract the authorization cookie. This only needs to be done if the user has been verified as logged in previously.

3. Import a new function from the [react-apollo] package inside the server's [index.js] file. Replace the import of the ReactDOM package with the following line of code:

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

4. Originally, we used the ReactDOM server methods to render the React code to HTML. These functions are synchronous; that is why the GraphQL request did not finish. To wait for all GraphQL requests, replace all of the lines, beginning from the [rendertoString] function until the end of the SSR route inside the server's [index.js] file. The result should look as follows:

```
renderToStringWithData(App).then((content) => {
  if (context.url) {
    res.redirect(301, context.url);
  } else {
    const head = Helmet.renderStatic();
    res.status(200);
    res.send(`<!doctype html>\n${template(content, head)}`);
    res.end();
```

```
  }
});
```

The [renderToStringWithData] function renders the React code, including the data received by the Apollo requests. Since the method is asynchronous, we wrap the rest of our code inside a callback function.

Now, if you take a look at the HTML returned by your server, you should see the correct markup, including chats, images, and everything else. The problem is that the client does not know that all of the HTML is already there, and can be reused. The client would rerender everything.

5. To let the client reuse the HTML that our server sends, we have to include the Apollo Client's state with our response. Inside the preceding callback, access the Apollo Client's state by inserting the following code:

```
const initialState = client.extract();
```

The [client.extract] method returns a big object, holding all cache information that the client has stored after using the [renderToStringWithData] function.

6. The state must be passed to the [template] function as a third parameter. Change the [res.send] call to the following:

```
res.send(`<!doctype html>\n${template(content, head, initialState)}`);
```

7. Inside the [template.js] file, extend the function declaration and append the [state] variable as a third parameter, after the [head] variable.
8. Insert the [state] variable, with the following line of code, inside the HTML body and above the [bundle.js] file. If you add it below the [bundle.js] file, it won't work correctly:

```
${ReactDOM.renderToStaticMarkup(<script dangerouslySetInnerHTML=
{{__html: `window.__APOLLO_STATE__=${JSON.stringify(state).replace
(/</g, '\\u003c')}`}}/>)}
```

We use the [renderToStaticMarkup] function to insert another [script] tag. It sets a large, stringified JSON object as Apollo Client's starting cache value. The JSON object holds all results of the GraphQL requests returned while rendering our server-side React application. We directly store the JSON object as a string, in a new field inside the [window] object. The [window] object is helpful, since you can directly access the field globally.

9. Apollo has to know about the state variable. It can be used by the Apollo Client, in order to initialize its cache with the specified data, instead of sending all GraphQL requests again. Open the [index.js] from the client's [apollo] folder. The last property of the initialization process is the cache. We need to set our [__APOLLO_STATE__] as the starting value of the cache. Replace the [cache] property with the following code:

```
cache: new InMemoryCache().restore(window.__APOLLO_STATE__)
```

We create the [InMemoryCache] instance and run its [restore] method, where we insert the value from the window object. The Apollo Client should recreate its cache from this variable.

10. We have now set up the cache for Apollo. It will no longer run unnecessary requests, for which the results already exist. Now, we can finally reuse the HTML, with one last change. We have to change [ReactDOM.render] to [ReactDOM.hydrate] in the client's [index.js] file. The difference between these functions is that React reuses the HTML if it was correctly rendered by our server. In this case, React only attaches some necessary event listeners. If you use the [ReactDOM.render] method, it dramatically slows down the initial rendering process, because it compares the initial DOM with the current DOM and may change it accordingly.

The last problem that we have is that the client-side code does not show the logged in state of our application after refreshing a page. The server returns the correct markup with all the data, but the front end redirects us to the login form. The reason for this is that we statically set the [loggedIn] state variable to false in the [App.js] file of the client-side code.

The best way to check whether the user is authenticated is to verify whether the [__APOLLO_STATE__] field on the window object is filled and has a [currentUser] object attached. If that is the case, we can assume that the user was able to fetch their own data record, so they must be [loggedIn]. To change our [App.js] file accordingly, add the following condition to the [loggedIn] state variable:

```
(typeof window.__APOLLO_STATE__ !== typeof undefined && typeof
window.__APOLLO_STATE__.ROOT_QUERY !== typeof undefined && typeof
window.__APOLLO_STATE__.ROOT_QUERY.currentUser !== typeof undefined)
```

As you can see in the preceding code, we verify whether the Apollo starting cache variable includes a [ROOT_QUERY] property with the subfield [currentUser]. The [ROOT_QUERY] property is filled if any query can be fetched successfully. The [currentUser] field is only filled if the authenticated was successfully requested.

If you execute [npm run server], you will see that now everything works perfectly. Take a look at the markup that's returned; you will see either the login form or, when logged in, all of the content of the page that you are visiting. You can log in on the client, the news feed is fetched dynamically, you can refresh the page, and all of the posts are directly there, without the need for a single GraphQL request, because the server returned the data side by side with the HTML. This works not only for the [/app] path, but for any path that you implement.

We are now finished with the SSR setup.

Summary

In this lab, we changed a lot of the code that we have programmed so far. You learned the advantages and disadvantages of offering server-side rendering. The main principles behind React Router, Apollo, and authentication with cookies while using SSR should be clear by now. There is much work required to get SSR running, and it needs to be managed with every change made to your application. Nevertheless, it has excellent performance and experience benefits for your users.

In the next lab, we will look at how to offer real-time updates through Apollo Subscriptions, instead of using the old and inefficient polling.