Lab 11: Writing Tests

This lab covers the following topics:

- How to use Mocha for testing
- Testing a GraphQL API with Mocha and Chai
- Testing React with Enzyme and JSDOM

Lab Solution

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

```
cd ~/Desktop/react-graphql-course/labs/Lab11
```

Testing with Mocha

To get started, we have to install all the dependencies to test our application with npm:

```
npm install --save-dev mocha chai @babel/polyfill request
```

The [mocha] package includes almost everything to run tests. Along with Mocha, we also install [chai], which is an assertion library. It offers excellent ways to chain tests with many variables and types for use inside a Mocha test. We also install the [@babel/polyfill] package, which allows our test to support ES2015+ syntax. This package is crucial because we use this syntax everywhere throughout our React code. Finally, we install the [request] package as a library to send all the queries or mutations within our test. I recommend you set the [NODE_ENV] environment variable to [production] to test every functionality, as in a live environment. Be sure that you set the environment variable correctly so that all production features are used.

Our first Mocha test

First, let's add a new command to the [scripts] field of our [package.json] file:

```
"test": "mocha --exit test/ --require babel-hook --require @babel/polyfill --
recursive"
```

If you now execute [npm run test], we'll run the [mocha] package in the [test] folder, which we'll create in a second. The preceding [--require] option loads the specified file or package. We'll also load a [babel-hook.js] file, which we'll create as well. The [--recursive] parameter tells Mocha to run through the complete file tree of the [test] folder, not just the first layer. This behavior is useful because it allows us to structure our tests in multiple files and folders.

Let's begin with the [babel-hook.js] file by adding it to the root of our project, next to the [package.json] file. Insert the following code:

```
require("@babel/register")({
  "plugins": [
    "require-context-hook"
  ],
  "presets": ["@babel/env","@babel/react"]
});
```

The purpose of this file is to give us an alternative Babel configuration file to our standard [.babelrc] file. If you compare both files, you should see that we use the [require-context-hook] plugin. We already use this plugin when

starting the back end with [npm run server]. It allows us to import our Sequelize models using a regular expression.

If we start our test with [npm run test], we require this file at the beginning. Inside the [babel-hook.js] file, we load [@babel/register], which compiles all the files that are imported afterward in our test according to the preceding configuration.

This gives us the option to start our back end server from within our test file and render our application on the server. The preparation for our test is now finished. Create a folder named [test] inside the root of our project to hold all runnable tests. Mocha will scan all files or folders, and all tests will be executed. To get a basic test running, create [app.test.js]. This is the main file, which makes sure that our back end is running and in which we can subsequently define further tests. The first version of our test looks as follows:

```
const assert = require('assert');
const request = require('request');
const expect = require('chai').expect;
const should = require('chai').should();

describe('Graphbook application test', function() {

  it('renders and serves the index page', function(done) {
    request('http://localhost:8000', function(err, res, body) {
      should.not.exist(err);
      should.exist(res);
      expect(res.statusCode).to.be.equal(200);
      assert.ok(body.indexOf('<html') !== -1);
      done(err);
    });
  });

});
```

Let's take a closer look at what's happening here:

1. We import the Node.js [assert] function. It gives us the ability to verify the value or the type of a variable.
2. We import the [request] package, which we use to send queries against our back end.
3. We import two Chai functions, [expect] and [should], from the [chai] package. Neither of these is included in Mocha, but they both improve the test's functionality significantly.
4. The beginning of the test starts with the [describe] function. Because Mocha executes the [app.test.js] file, we're in the correct scope and can use all Mocha functions. The [describe] function is used to structure your test and its output.
5. We use the [it] function, which initiates the first test.

The [describe] function is the header of our test's output. Then, we have a new row for each [it] function we execute. Each row represents a single test step. The [it] function passes a [done] function to the callback. The [done] function has to be executed once all assertions are finished and there's nothing left to do. If it isn't executed in a certain amount of time, the current test is marked as failed. In the preceding code, the first thing we did was send an HTTP [GET] request to [http://localhost:8000], which is accepted by our back end server. The expected answer will be in the form of server-side rendered HTML created through React.

If you execute [npm run test] now, you'll receive the following error:

```
Graphbook application test
  1) renders and serves the index page


0 passing (1s)
1 failing

1) Graphbook application test
     renders and serves the index page:
   Uncaught AssertionError: expected [Error: connect ECONNREFUSED 127.0.0.1:8000] to not exist
    at Object.should.not.exist (node_modules\chai\lib\chai\interface\should.js:207:38)
    at Request._callback (E:/Arbeit/Buch/chapter 11 - final/test/app.test.js:24:18)
    at self.callback (E:\node_modules\request\request.js:185:22)
    at Request.onRequestError (E:\node_modules\request\request.js:877:8)
    at Socket.socketErrorListener (_http_client.js:387:9)
    at emitErrorNT (internal/streams/destroy.js:64:8)
    at _combinedTickCallback (internal/process/next_tick.js:138:11)
    at process._tickCallback (internal/process/next_tick.js:180:9)
```

Our first [should.not.exist] assertion failed and threw an error. This is because we didn't start the back end when we ran the test. Start the back end in a second terminal with the correct environment variables using [npm run server] and rerun the test. Now, the test is successful:

```
Graphbook application test
  √ renders and serves the index page (52ms)


1 passing (238ms)
```

The output is good, but the process isn't very intuitive. The current workflow is hard to implement when running the tests automatically while deploying your application or pushing new commits to your version-control system. We'll change this behavior next.

Starting the back end with Mocha

When we want to run a test, the server should start automatically. There are two options to implement this behavior:

- We add the [npm run server] command to the [test] script inside our [package.json] file.
- We import all the necessary files to launch the server within our [app.test.js]. This allows us to run further assertions or commands against the back end.

The best option is to start the server within our test and not rely on a second command, because we can run further tests on the back end. We to import a further package to allow the server to start within our test:

```
require('babel-plugin-require-context-hook/register')();
```

We use and execute this package because we load the Sequelize models using the [require.context] function. By loading the package, the [require.context] function is executable for the server-side code. Before we started the server within the test, the plugin hadn't been used, although it was loaded in the [babel-hooks.js] file.

Now we can load the server directly in the test. Add the following lines at the top of the [describe] function, just before the test we've just written:

```
var app;
this.timeout(50000);
```

```
before(function(done) {
  app = require('../src/server').default;
  app.on("listening", function() {
    done();
  });
});
```

The idea is to load the server's [index.js] file inside of our test, which starts the back end automatically. To do this, we define an empty variable called [app]. Then, we use [this.timeout] to set the timeout for all tests inside Mocha to [50000], because starting our server, including Apollo Server, takes some time. Otherwise, the test will probably fail because the start time is too long for the standard Mocha timeout.

We must make sure that the server has been completely started before any of our tests are executed. This logic can be achieved with Mocha's [before] function. Using this function, you can set up and configure things such as starting a back end in our scenario. To continue and process all the tests, we need to execute the [done] function to complete the callback of the [before] function. To be sure that the server has started, we do not just run the [done] function after loading the [index.js] file. We bind the [listening] event of the server using the [app.on] function. If the server emits the [listening] event, we can securely run the [done] function, and all tests can send requests to the server. We could also require the server directly into the [app] variable at the top. The problem with this order, however, is that the server may start listening before we can bind the [listening] event. The way we are doing it now makes sure the server hasn't yet started.

The test, however, still isn't working. You'll see an error message that says ['TypeError: app.on is not a function']. Take a closer look at the server's [index.js] file. At the end of the file, we aren't exporting the server object because we only used it to start the back end. This means that the [app] variable in our test is empty and we can't run the [app.on] function. The solution is to export the [server] object at the end of the server's [index.js] file:

```
export default server;
```

You can now execute the test again. Everything should look fine, and all tests should pass.

There is, however, one last problem. If you compare the behavior from the test before importing the server directly in our test or starting it in a second terminal, you might notice that the test isn't finished, or at least the process isn't stopped. Previously, all steps were executed, we returned to the normal shell, and we could execute the next command. The reason for this is that the server is still running in our [app.test.js] file. Therefore, we must stop the back end after all tests have been executed. Insert the following code after the [before] function:

```
after(function(done) {
  app.close(done);
});
```

The [after] function is run when all tests are finished. Our [app] object offers the [close] function, which terminates the server. As a callback, we hand over the [done] function, which is executed once the server has stopped. It means that our test has also finished.

Verifying the correct routing

We now want to check whether all the features of our application are working as expected. One major feature of our application is that React Router redirects the user in two cases:

- The user visits a route that cannot be matched.
- The user visits a route that can be matched, but they aren't allowed to view the page.

In both cases, the user should be redirected to the login form. In the first case, we can follow the same approach as for our first test. We send a request to a path that isn't inside our router. Add the code to the bottom of the [describe] function:

```
describe('404', function() {
  it('redirects the user when not matching path is found', function(done) {
    request({
      url: 'http://localhost:8000/path/to/404',
    }, function(err, res, body) {
      should.not.exist(err);
      should.exist(res);
      expect(res.statusCode).to.be.equal(200);
      assert.ok(res.req.path === '/');
      assert.ok(body.indexOf('<html') !== -1);
      assert.ok(body.indexOf('class="authModal"') !== -1);
      done(err);
    });
  });
});
```

Let's quickly go through all steps of the preceding test:

1. We add a new [describe] function to structure our test's output.
2. We send a request inside another [it] function to an unmatched path.
3. The checks are the same as the ones we used when starting the server.
4. We verify that the response's path is the [/] root. That happens when the redirect is executed. Therefore, we use the [res.req.path === '/'] condition.
5. We check whether the returned [body] includes an HTML tag with the [authModal] class. This should happen when the user isn't logged in, and the login or register form is rendered.

If the assertions are successful, we know that the React Router works correctly in the first scenario. The second scenario relates to the private routes that can only be accessed by authenticated users. We can copy the preceding check and replace the request. The assertions we are doing stay the same, but the URL of the request is different. Add the following test under the previous one:

```
describe('authentication', function() {
  it('redirects the user when not logged in', function(done) {
    request({
      url: 'http://localhost:8000/app',
    }, function(err, res, body) {
      should.not.exist(err);
      should.exist(res);
      expect(res.statusCode).to.be.equal(200);
      assert.ok(res.req.path === '/');
      assert.ok(body.indexOf('<html') !== -1);
      assert.ok(body.indexOf('class="authModal"') !== -1);
      done(err);
    });
  });
});
```

If an unauthenticated user requests the [/app] route, they're redirected to the [/] root path. The assertions verify whether the login form is displayed as before. To differentiate the tests, we add a new [describe] function so that it has a better structure.

Next, we want to test the GraphQL API that we built, not only the SSR functionality.

Testing GraphQL with Mocha

We must verify that all the API functions we're offering work correctly. I'm going to show you how to do this with two examples:

- The user needs to sign up or log in. This is a critical feature where we should verify that the API works correctly.
- The user queries or mutates data via the GraphQL API. For our test case, we will request all chats the logged-in user is related to.

Those two examples should explain all the essential techniques to test every part of your API. You can add more functions that you want to test at any point.

Testing the authentication

We extend the authentication tests of our test with the signup functionality. We're going to send a simple GraphQL request to our back end, including all the required data to sign up a new user. We've already sent requests, so there's nothing new here. In comparison to all the requests before, however, we have to send a [POST] request, not a [GET] request. Also, the endpoint for the signup is the [/graphql] path, where our Apollo Server listens for incoming mutations or queries. Normally, when a user signs up for Graphbook, the authentication token is returned directly, and the user is logged in. We must preserve this token to make future GraphQL requests. We don't use Apollo Client for our test as we don't need to test the GraphQL API.

Create a global variable next to the [app] variable, where we can store the JWT returned after signup:

```
var authToken;
```

Inside the test, we can set the returned JWT. Add the following code to the [authentication] function:

```
it('allows the user to sign up', function(done) {
  const json = {
    operationName: null,
    query: "mutation signup($username: String!, $email : String!,
    $password : String!) { signup(username: $username, email: $email,
    password : $password) { token }}",
    variables: {
      "email": "mocha@test.com",
      "username": "mochatest",
      "password": "123456789"
    }
  };

  request.post({
    url: 'http://localhost:8000/graphql',
    json: json,
  }, function(err, res, body) {
    should.not.exist(err);
    should.exist(res);
    expect(res.statusCode).to.be.equal(200);
    body.should.be.an('object');
    body.should.have.property('data');
```

```
    authToken = body.data.signup.token;
    done(err);
  });
});
```

We begin by creating a [json] variable. This object is sent as a JSON body to our GraphQL API. The content of it should be familiar to you. It's nearly the same format we used when testing the GraphQL API in Postman.

ProTip

The JSON we send represents a manual way of sending GraphQL requests. There are libraries that you can easily use to save this and directly send the query without wrapping it inside an object, such as [graphql-request]: https://github.com/prisma/graphql-request.

The [json] object includes fake signup [variables] to create a user with the [mochatest] username. We'll send an HTTP [Post] with the [request.post] function. To use the [json] variable, we pass it into the [json] field. The [request.post] function automatically adds the body as a JSON string and the correct [Content-Type] header for you. When the response arrives, we run the standard checks, such as checking for an error or checking an HTTP status code. We also check the format of the returned [body], because the response's [body] won't return HTML, but will return JSON instead. We make sure that it's an object with the [should.be.an('object')] function. The [should] assertion can directly be used and chained to the [body] variable. If [body] is an object, we check whether there's a [data] property inside. That's enough security to read the token from the [body.data.signup.token] property.

The user is now created in our database. We can use this token for further requests. Be aware that running this test a second time on your local machine is likely to result in a failure because the user already exists. In this case, you can delete it manually from your database. This problem won't occur when running this test while using Continuous Integration. We'll focus on this topic in the last lab. Next, we'll make an authenticated query to our Apollo Server and test the result of it.

Testing authenticated requests

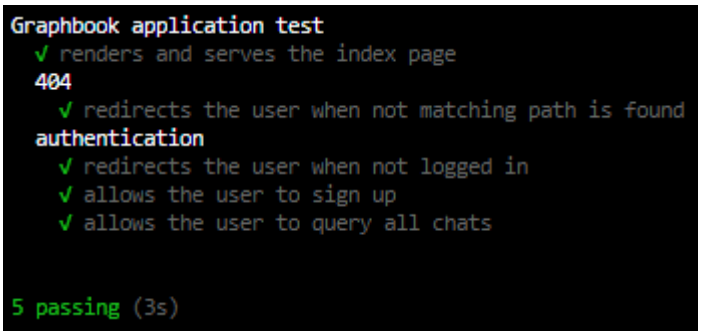
We set the [authToken] variable after the signup request. You could also do this with a login request if a user already exists while testing. Only the query and the assertions we are using are going to change. Also insert the following code into the [before] authentication function:

```
it('allows the user to query all chats', function(done) {
  const json = {
    operationName: null,
    query: "query {chats {id users {id avatar username}}}",
    variables: {}
  };

  request.post({
    url: 'http://localhost:8000/graphql',
    headers: {
      'Authorization': authToken
    },
    json: json,
  }, function(err, res, body) {
    should.not.exist(err);
    should.exist(res);
    expect(res.statusCode).to.be.equal(200);
    body.should.be.an('object');
    body.should.have.property('data');
```

```
    body.data.should.have.property('chats').with.lengthOf(0);
    done(err);
  });
});
```

As you can see in the preceding code, the [json] object doesn't include any variables because we only query the chats of the logged-in user. We changed the [query] string accordingly. Compared to the login or signup request, the chat query requires the user to be authenticated. The [authToken] we saved is sent inside the [Authorization] header. We now verify again whether the request was successful and check for a [data] property in the [body]. Notice that, before running the [done] function, we verify that the [data] object has a field called [chats]. We also check the length of the [chats] field, which proves that it's an array. The length can be statically set to [0] because the user who's sending the query just signed up and doesn't have any chats yet. The output from Mocha looks as follows:



This is all you need to know to test all the features of your API.

Testing React with Enzyme

So far, we've managed to test our server and all GraphQL API functions. Currently, however, we're still missing the tests for our front end code. While we render the React code when requesting any server route, such as the [/app] path, we only have access to the final result and not to each component. We should change this to execute the functions of certain components that aren't testable through the back end. First, install some dependencies before using npm:

```
npm install --save-dev enzyme enzyme-adapter-react-16 ignore-styles jsdom isomorphic-fetch
```

The various packages are as follows:

- The [enzyme] and [enzyme-adapter-react-16] packages provide React with specific features to render and interact with the React tree. This can either be through a real DOM or shallow rendering. We are going to use a real DOM in this lab because it allows us to test all features, while shallow rendering is limited to just the first layer of components.
- The [ignore-styles] package strips out all [import] statements for CSS files. This is very helpful, since we don't need CSS for our tests.
- The [jsdom] package creates a DOM object for us, which is then used to render the React code into.
- The [isomorphic-fetch] package replaces the [fetch] function that all browsers provide by default. This isn't available in Node.js, so we need a polyfill.

We start by importing the new packages directly under the other [require] statements:

```
require('isomorphic-fetch');
import React from 'react';
import { configure, mount } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
configure({ adapter: new Adapter() });
import register from 'ignore-styles';
register(['.css', '.sass', '.scss']);
```

To use Enzyme, we import React. Then, we create an adapter for Enzyme that supports React 16. We insert the adapter into Enzyme's [configure] statement. Before starting with the front end code, we import the [ignore-styles] package to ignore all CSS imports. I've also directly excluded SASS and SCSS files. The next step is to initialize our DOM object, where all the React code is rendered:

```
const { JSDOM } = require('jsdom');
const dom = new JSDOM('<!doctype html><html><body></body></html>', { url:
'http://graphbook.test' });
const { window } = dom;
global.window = window;
global.document = window.document;
```

We require the [jsdom] package and initialize it with a small HTML string. We don't take the template file that we're using for the server or client because we just want to render our application to any HTML, so how it looks isn't important. The second parameter is an options object. We specify a [url] field, which is the host URL, under which we render the React code. Otherwise, we might get an error when accessing [localStorage]. After initialization, we extract the [window] object and define two global variables that are required to mount a React component to our fake DOM. These two properties behave like the [document] and [window] objects in the browser, but instead of the browser they are global objects inside our Node.js server.

In general, it isn't a good idea to mix up the Node.js [global] object with the DOM of a browser and render a React application in it. Still, we're merely testing our application and not running it in production in this environment, so while it might not be recommended, it helps our test to be more readable. We'll begin the first front end test with our login form. The visitor to our page can either directly log in or switch to the signup form. Currently, we don't test this switch functionality in any way. This is a complex example, but you should be able to understand the techniques behind it quickly.

To render our complete React code, we're going to initialize an Apollo Client for our test. Import all the dependencies:

```
import { ApolloClient } from 'apollo-client';
import { InMemoryCache } from 'apollo-cache-inmemory';
import { ApolloLink } from 'apollo-link';
import { createUploadLink } from 'apollo-upload-client';
import App from '../src/server/ssr';
```

We also import the [index.js] component of the server-rendered React code. This component will receive our client, which we'll initialize shortly. Add a new [describe] function for all front end tests:

```
describe('frontend', function() {
  it('renders and switches to the login or register form',
  function(done) {
    const httpLink = createUploadLink({
      uri: 'http://localhost:8000/graphql',
      credentials: 'same-origin',
```

```
    });
    const client = new ApolloClient({
      link: ApolloLink.from([
        httpLink
      ]),
      cache: new InMemoryCache()
    });
  });
});
```

The preceding code creates a new Apollo Client. The client doesn't implement any logic, such as authentication or WebSockets, because we don't need this to test the switch from the login form to the signup form. It's merely a required property to render our application completely. If you want to test components that are only rendered when being authenticated, you can, of course, implement it easily. Enzyme requires us to pass a real React component, which will be rendered to the DOM. Add the following code directly beneath the [client] variable:

```
class Graphbook extends React.Component {
  render() {
    return(
      <App client={client} context={{}} loggedIn={false} location=
       {"/"}/>
    )
  }
}
```

The preceding code is a small wrapper around the [App] variable that we imported from the server's [ssr] folder. The [client] property is filled with the new Apollo Client. Follow the given instructions to render and test your React front end code. The following code goes directly under the [Graphbook] class:

1. We use the [mount] function of Enzyme to render the [Graphbook] class to the DOM:

```
const wrapper = mount(<Graphbook />);
```

2. The [wrapper] variable provides many functions to access or interact with the DOM and the components inside it. We use it to prove that the first render displays the login form:

```
expect(wrapper.html()).to.contain('<a>Want to sign up? Click here</a>');
```

The [html] function of the [wrapper] variable returns the complete HTML string that has been rendered by the React code. We check this string with the [contain] function of Chai. If the check is successful, we can continue.

3. Typically, the user clicks on the [Want to sign up?] message and React rerenders the signup form. We need to handle this via the [wrapper] variable. Enzyme comes with that functionality innately:

```
wrapper.find('LoginRegisterForm').find('a').simulate('click');
```

The [find] function gives us access to the [LoginRegisterForm] component. Inside the markup of the component, we search for an [a] tag, of which there can only be one. If the [find] method returns multiple results, we can't trigger things such as a click, because the [simulate] function is fixed to only one possible target. After running both [find] functions, we execute Enzyme's [simulate] function. The only parameter needed is the event that we want to trigger. In our scenario, we trigger a click event on the [a] tag, which lets React handle all the rest.

4. We check whether the form was changed correctly:

```
expect(wrapper.html()).to.contain('<a>Want to login? Click
 here</a>');
done();
```

We use the [html] and [contain] functions to verify that everything was rendered correctly. The [done] method of Mocha is used to finish the test.

ProTip

For a more detailed overview of the API and all the functions that Enzyme provides, have a look at the official documentation: https://airbnb.io/enzyme/docs/api/.

This was the easy part. How does this work when we want to verify whether the client can send queries or mutations with authentication? It's actually not that different. We already registered a new user and got a JWT in return. All we need to do is attach the JWT to our Apollo Client, and the Router needs to receive the correct [loggedIn] property. The final code for this test looks as follows:

```
it('renders the current user in the top bar', function(done) {
  const AuthLink = (operation, next) => {
    operation.setContext(context => ({
        ...context,
        headers: {
            ...context.headers,
            Authorization: authToken
        },
    }));
    return next(operation);
  };

  const httpLink = createUploadLink({
    uri: 'http://localhost:8000/graphql',
    credentials: 'same-origin',
  });

  const client = new ApolloClient({
    link: ApolloLink.from([
      AuthLink,
      httpLink
    ]),
    cache: new InMemoryCache()
  });

  class Graphbook extends React.Component {
    render() {
      return(
        <App client={client} context={{}} loggedIn={true} location=
         {"/app"}/>
      )
    }
  }

  const wrapper = mount(<Graphbook />);
  setTimeout(function() {
```

```
    expect(wrapper.html()).to.contain('<div class="user"><img>
     <span>mochatest</span></div>');
    done();
  },2000);
});
```

Here, we are using the `AuthLink` that we used in the original front end code. We pass the `authToken` variable to every request that's made by the Apollo Client. In the `Apollo.from` method, we add it before `httpLink`. In the `Graphbook` class, we set `loggedIn` to `true` and the `location` to `/app` to render the newsfeed. Because the requests are asynchronous by default and the `mount` method doesn't wait for the Apollo Client to fetch all queries, we couldn't directly check the DOM for the correct content. Instead, we wrapped the assertions and the `done` function in a `setTimeout` function. A timeout of 2,000 milliseconds should be enough for all requests to finish and React to have rendered everything. If this isn't enough time, you can increase the number. When all assertions are successful, we can be sure that the `currentUser` query has been run and the top bar has been rendered to show the logged-in user. With these two examples, you should now be able to run any test you want with your application's front end code.

Summary

In this lab, we learned all the essential techniques to test your application automatically, including testing the server, the GraphQL API, and the user's front end. You can apply the Mocha and Chai patterns you learned to other projects to reach a high software quality at any time. Your personal testing time will be greatly reduced.

In the next lab, we'll have a look at how to improve performance and error logging so we're always providing a good user experience.