

Full-Stack Web Development with GraphQL and React





Table of Contents

1. Preparing Your Development Environment: 4
2. Setting up GraphQL with Express.js: 61
3. Connecting to The Database: 114
4. Integrating React into the Back end with Apollo: 219
5. Reusable React Components: 306
6. Authentication with Apollo and React: 397
7. Handling Image Uploads: 462





Table of Contents

8. Routing in React : 500
9. Implementing Server-Side Rendering : 546
10. Real-Time Subscriptions : 600
11. Writing Tests : 649
12. Optimizing GraphQL with Apollo Engine: 683
13. Continuous Deployment with CircleCI and Heroku: 713





1: Preparing Your Development Environment



Preparing Your Development Environment

This lesson covers the following topics:

- Architecture and technology
- Thinking critically about how to architect a stack
- Building the React and GraphQL stack
- Installing and configuring Node.js
- Setting up a React development environment with webpack, Babel, and other requirements
- Debugging React applications using Chrome DevTools and React Developer Tools
- Using webpack-bundle-analyzer to check the bundle size

Application architecture

- Since its initial release in 2015, GraphQL has become the new alternative to the standard SOAP and REST APIs.
- GraphQL is a specification, like SOAP and REST, that you can follow to structure your application and data flow.
- It is so innovative because it allows you to query specific fields of entities, such as users and posts.

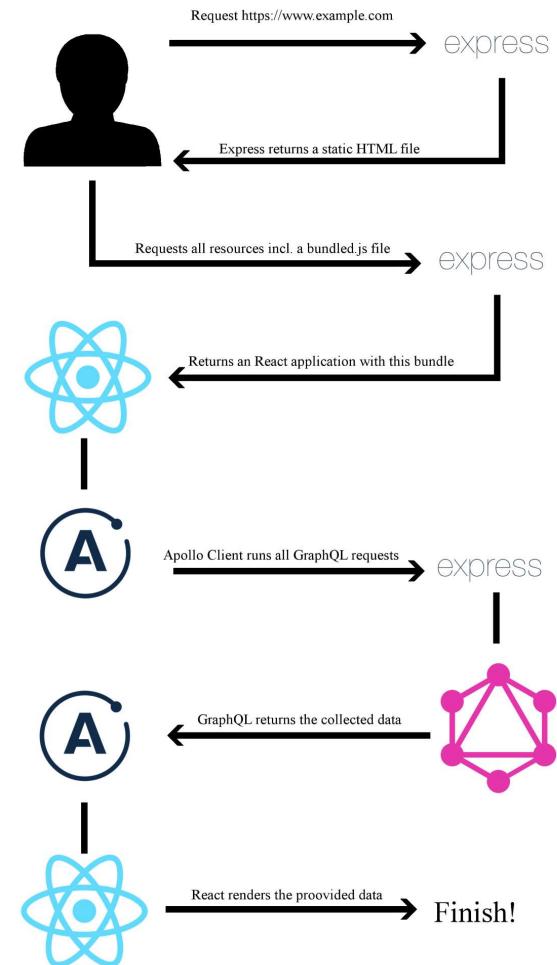
Application architecture

- For example, a query for a post may look like this:

```
post {  
  id  
  text  
  user {  
    user_id  
    name  
  }  
}
```

The basic setup

- The basic setup to make an application work is the logical request flow, which looks as follows:



Installing and configuring Node.js

The first step for preparing for our project is to install Node.js. There are two ways to do this:

- One option is to install the Node Version Manager (NVM).
- The benefit of using NVM is that you are easily able to run multiple versions of Node.js side by side and this handles the installation process for you on nearly all UNIX-based systems, such as Linux and macOS.
- Within this course, we do not need the option to switch between different versions of Node.js.

Installing and configuring Node.js

- First, let's add the correct repository for our package manager by running:

```
curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash -
```

- Next, install Node.js and the build tools for native modules, using the following command:

```
sudo apt-get install -y nodejs build-essential
```

- Finally, let's open a terminal now and verify that the installation was successful:

```
node --version
```

Setting up React

- The development environment for our project is ready.
- In this section, we are going to install and configure React, which is one primary aspect of this course.
- Let's start by creating a new directory for our project:

```
mkdir ~/graphbook  
cd ~/graphbook
```

Setting up React

- Just run npm init to create an empty package.json file:

`npm init`

Setting up React

You can see an example of the command line in the following screenshot:

This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.

```
package name: (graphbook)
version: (1.0.0) 0.0.1
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\Users\sebig\Desktop\testit\graphbook\package.json:
```

```
{
  "name": "graphbook",
  "version": "0.0.1",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Setting up React

- The first and most crucial dependency for this course is React.
- Use npm to add React to our project:

```
npm install --save react react-dom
```

Preparing and configuring webpack

- Create a separate directory for our index.html file:

```
mkdir public  
touch index.html
```

- Save this inside index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
      scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Graphbook</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

Preparing and configuring webpack

- To bundle our JavaScript code, we need to install webpack and all of its dependencies as follows:

```
npm install --save-dev @babel/core babel-eslint babel-loader  
@babel/preset-env @babel/preset-react clean-webpack-plugin css-  
loader eslint file-loader html-webpack-plugin style-loader url-loader  
webpack webpack-cli webpack-dev-server @babel/plugin-  
proposal-decorators @babel/plugin-proposal-function-sent  
@babel/plugin-proposal-export-namespace-from @babel/plugin-  
proposal-numeric-separator @babel/plugin-proposal-throw-  
expressions @babel/plugin-proposal-class-properties
```

Preparing and configuring webpack

- The following handy shortcut installs the eslint configuration created by the people at Airbnb, including all peer dependencies.
- Execute it straight away:

```
npx install-peerdeps --dev eslint-config-airbnb
```

Preparing and configuring webpack

- Create a `.eslintrc` file in the root of your project folder to use the airbnb configuration:

```
{  
  "extends": ["airbnb"],  
  "env": {  
    "browser": true,  
    "node": true  
  },  
  "rules": {  
    "react/jsx-filename-extension": "off"  
  }  
}
```

Preparing and configuring webpack

- Enter the following:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/1_1.txt

Preparing and configuring webpack

- With this in mind, let's move on.
- We are missing the src/client/index.js file from our webpack configuration, so let's create it as follows:

```
mkdir src/client  
cd src/client  
touch index.js
```

Preparing and configuring webpack

- Add this line to the scripts object inside package.json:

```
"client": "webpack-dev-server --devtool inline-source-map --hot --config webpack.client.config.js"
```

Render your first React component

- There are many best practices for React. The central philosophy behind it is to split up our code into separate components where possible.
- We are going to cover this approach in more detail later in lesson 5, Reusable React Components.

Render your first React component

- The index.js file should include this code:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
```

```
ReactDOM.render(<App/>,
document.getElementById('root'));
```

Render your first React component

- Create the App.js file next to your index.js file, with the following content:

```
import React, { Component } from 'react';
export default class App extends Component {
  render() {
    return (
      <div>Hello World!</div>
    )
  }
}
```

Render your first React component

- Let's create a `.babelrc` file in the root folder with this content:

```
{  
  "plugins": [  
    "@babel/plugin-proposal-decorators", { "legacy": true },  
    "@babel/plugin-proposal-function-sent",  
    "@babel/plugin-proposal-export-namespace-from",  
    "@babel/plugin-proposal-numeric-separator",  
    "@babel/plugin-proposal-throw-expressions",  
    ["@babel/plugin-proposal-class-properties", { "loose": false }]  
,  
  "presets": ["@babel/env", "@babel/react"]  
}
```

Rendering arrays from React state

- Hello World! is a must for every good programming course, but this is not what we are aiming for when we use React.
- A social network such as Facebook or Graphbook, which we are writing at the moment, needs a news feed and an input to post news.

Let's implement this.

- Define a new variable above your App class like this:

```
const posts = [  
  id: 2,  
  text: 'Lorem ipsum',  
  user: {  
    avatar: '/uploads/avatar1.png',  
    username: 'Test User'  
  },  
  {  
    id: 1,  
    text: 'Lorem ipsum',  
    user: {  
      avatar: '/uploads/avatar2.png',  
      username: 'Test User 2'  
    },  
  }];
```

- Replace the current content of your render method with the following code:

```
const { posts } = this.state;  
return (  
  <div className="container">  
    <div className="feed">  
      {posts.map((post, i) =>  
        <div key={post.id} className="post">  
          <div className="header">  
            <img src={post.user.avatar} />  
            <h2>{post.user.username}</h2>  
          </div>  
          <p className="content">  
            {post.text}  
          </p>  
        </div>  
      )}  
    </div>  
  )
```

Rendering arrays from React state

- To get our posts into the state, we can define them inside our class with property initializers.
- Add this to the top of the App class:

```
state = {  
  posts: posts  
}
```

Rendering arrays from React state

- The older way of implementing this—without using the ES6 feature—was to create a constructor:

```
constructor(props) {  
  super(props);  
  
  this.state = {  
    posts: posts  
  };  
}
```

Rendering arrays from React state



Test User

Lorem ipsum



Test User 2

Lorem ipsum

CSS with webpack

- The posts from the preceding picture have not been designed yet.
- I have already added CSS classes to the HTML our component returns.
- Instead of using CSS to make our posts look better, another method is to use CSS-in-JS using packages such as styled-components, which is a React package.

CSS with webpack

- What we've already done in our `webpack.client.config.js` file is to specify a CSS rule, as you can see in the following code snippet:

```
{  
  test: /\.css$/,  
  use: ['style-loader', 'css-loader'],  
},
```

CSS with webpack

- Create a style.css file in/assets/css and fill in the following:

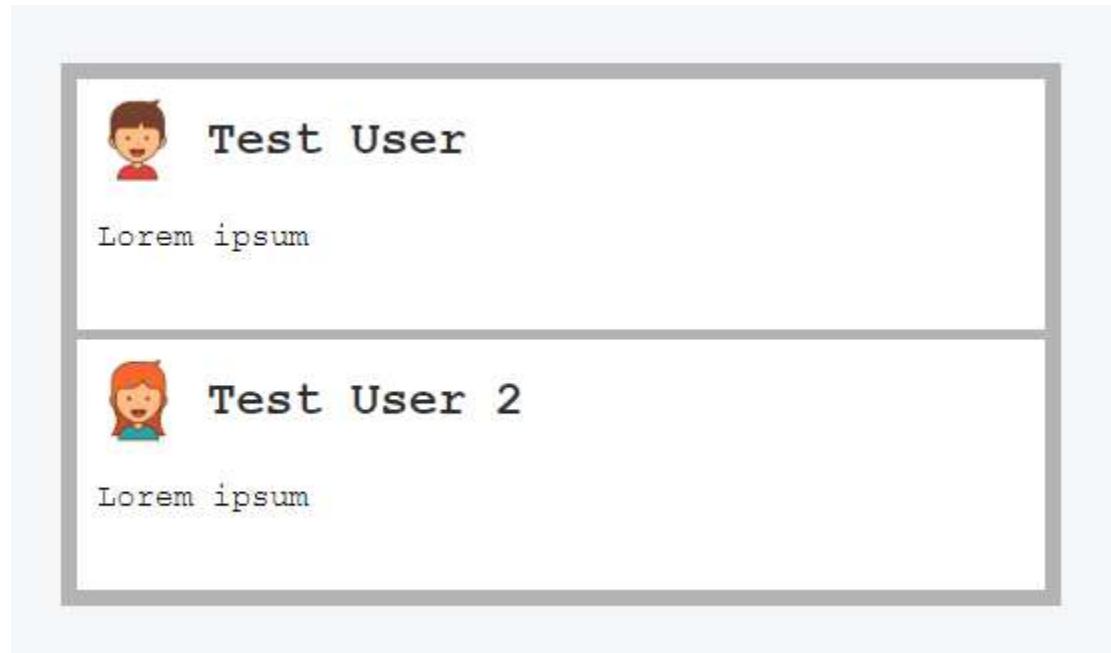
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/1_2.txt

CSS with webpack

- In your App.js file, add the following behind the React import statement:

```
import '../assets/css/style.css';
```

CSS with webpack



Event handling and state updates with React

- Add this above the div with the feed class:

```
<div className="postForm">
  <form onSubmit={this.handleSubmit}>
    <textarea value={postContent}
onChange={this.handlePostContentChange}
      placeholder="Write your custom post!"/>
    <input type="submit" value="Submit" />
  </form>
</div>
```

Event handling and state updates with React

- Create an empty string variable at the state property initializer, as follows:

```
state = {  
  posts: posts,  
  postContent: "  
"}  
• Then, extract this from the class state inside the  
render method:
```

```
const { posts, postContent } = this.state;
```

Event handling and state updates with React

- The logical step is to implement this function:

```
handlePostContentChange = (event) => {  
  this.setState({postContent: event.target.value})  
}
```

- Maybe you are used to writing this a little differently, like this:

```
handlePostContentChange(event) {  
  this.setState({postContent: event.target.value})  
}
```

Event handling and state updates with React

- In this case, you would need to write a constructor for your class and manually bind the scope to your function as follows:

```
this.handlePostContentChange =  
this.handlePostContentChange.bind(this);
```

- Look at your browser again.
- The form is there, but it is not pretty, so add this CSS:

```
form {  
    padding-bottom: 20px;  
}  
form textarea {  
    width: calc(100% - 20px);  
    padding: 10px;  
    border-color: #bbb;  
}  
form [type=submit] {  
    border: none;  
    background-color: #6ca6fd;  
    color: #fff;  
    padding: 10px;  
    border-radius: 5px;  
    font-size: 14px;  
    float: right;  
}
```

- The last step is to implement the `handleSubmit` function for our form:

```
handleSubmit = (event) => {
  event.preventDefault();
  const newPost = {
    id: this.state.posts.length + 1,
    text: this.state.postContent,
    user: {
      avatar: '/uploads/avatar1.png',
      username: 'Fake User'
    }
  };
  this.setState((prevState) => ({
    posts: [newPost, ...prevState.posts],
    postContent: ""
  }));
}
```

Controlling document heads with React Helmet

- When developing a web application, it is crucial that you can control your document heads.
- You might want to change the title or description, based on the content you are presenting.
- React Helmet is a great package that offers this on the fly, including overriding multiple headers and server-side rendering.

Controlling document heads with React Helmet

- Install it with the following command:

```
npm install --save react-helmet
```

Controlling document heads with React Helmet

- Import react-helmet at the top of the file:

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

- Add Helmet itself directly above postFormdiv:

```
<Helmet>
  <title>Graphbook - Feed</title>
  <meta name="description" content="Newsfeed of all
  your friends on
  Graphbook" />
</Helmet>
```

Production build with webpack

- A production bundle does merge all JavaScript files, but also CSS files into two separate files.
- Those can be used directly in the browser.
- To bundle CSS files, we will rely on another webpack plugin, called MiniCss:

```
npm install --save-dev mini-css-extract-plugin
```

Production build with webpack

- We do not want to change the current `webpack.client.config.js` file, because it is made for development work.
- Add this command to the `scripts` object of your `package.json`:

```
"client:build": "webpack --config  
webpack.client.build.config.js"
```

Production build with webpack

- The mode needs to be production, not development.
- Require the MiniCss plugin:

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
```

Production build with webpack

- Replace the current CSS rule:

```
{  
  test: /\.css$/,  
  use: [{ loader: MiniCssExtractPlugin.loader,  
    options: {  
      publicPath: '../'  
    }  
  }, 'css-loader'],  
},
```

Production build with webpack

- Lastly, add the plugin to the plugins at the bottom of the configuration file:

```
new MiniCssExtractPlugin({  
  filename: 'bundle.css',  
})
```

Useful development tools

- When working with React, you will want to know why your application rendered in the way that it did.
- You need to know which properties your components received and how their current state looks.
- Since this is not displayed in the DOM or anywhere else in Chrome DevTools, you need a separate plugin.

```
▼<App>
  ▼<div className="container">
    ▼<div className="postForm">
      ▼<form onSubmit=fn()>
        <textarea value="" onChange=fn() placeholder="Write your custom post!"></textarea>
        <input type="submit" value="Submit"></input>
      </form>
    </div>
    ▼<div className="feed">
      ▼<div key="2" className="post">
        ▼<div className="header">
          </img>
          <h2>Test User</h2>
        </div>
        <p className="content">Lorem ipsum</p>
      </div>
      ▼<div key="1" className="post">
        ▼<div className="header">
          </img>
          <h2>Test User 2</h2>
        </div>
        <p className="content">Lorem ipsum</p>
      </div>
    </div>
  </div>
</App>
```

App

Useful development tools

- By clicking a component, your right-hand panel will show its properties, state, and context.
- You can try this with the App component, which is the only real React component:

```
Props  
Empty object  
State  
postContent: ""  
▶ posts: Array[2]
```

Analyzing bundle size

- People that are trying to use as little bandwidth as possible will want to keep their bundle size low.
- I recommend that you always keep an eye on this, especially when requiring more modules via npm.
- In this case, you can quickly end up with a huge bundle size, since npm packages tend to require other npm packages themselves.

Analyzing bundle size

- Install this with the following:

```
npm install --save-dev webpack-bundle-analyzer
```

Analyzing bundle size

- The analyze command spins up the webpack-bundle-analyzer, showing us how our bundle is built together and how big each package that we use is.
- Do this as follows:

npm run stats

npm run analyze

Analyzing bundle size



Summary

- In this lesson, we completed a working React setup.
- This is a good starting point for our front end. We can write and build static web pages with this setup.
- The next lesson primarily focuses on our setup for the back end.

"Complete Lab 1"

2: Setting up GraphQL with Express.js



Setting up GraphQL with Express.js

This lesson covers the following points:

- Express.js installation and explanation
- Routing in Express.js
- Middleware in Express.js
- Binding Apollo Server to a GraphQL endpoint
- Serving static assets with Express.js
- Back end debugging and logging

Node.js and Express.js

- One primary goal of this course is to set up a GraphQL API, which is then consumed by our React front end.
- To accept network requests (especially GraphQL requests), we are going to set up a Node.js web server.
- The most significant competitors in the Node.js web server area are Express.js, Koa, and Hapi. In this course, we are going to use Express.js.

Node.js and Express.js

- Installing Express.js is pretty easy.
- We can use npm in the same way as in the first lesson:

```
npm install --save express
```

Node.js and Express.js

- In the first lesson, we created all JavaScript files directly in the src/client folder.
- Now, let's create a separate folder for our server-side code.
- This separation gives us a tidy directory structure.
- We will create the folder with the following command:

```
mkdir src/server
```

Setting up Express.js

- First, we import express from node_modules, which we just installed. We can use import here since our back end gets transpiled by Babel.
- We are also going to set up webpack for the server-side code in a later in lesson 9, Implementing Server-Side Rendering.

```
import express from 'express';
```

Setting up Express.js

- We initialize the server with the express command.
- The result is stored in the app variable.
- Everything our back end does is executed through this object.

```
const app = express();
```

Setting up Express.js

- Then, we specify the routes that accept requests.
- For this straightforward introduction, we accept all HTTP GET requests matching any path, by using the app.get method.
- Other HTTP Methods are catchable with app.post, app.put, and so on.

```
app.get('*', (req, res) => res.send('Hello World!'));
app.listen(8000, () => console.log('Listening on port 8000!'));
```

Running Express.js in development

- We will add the following line to the scripts property of the package.json file:

```
"server": "nodemon --exec babel-node --watch  
src/server src/server/index.js"
```

- As you can see, we are using a command called nodemon. We need to install it first:

```
npm install --save nodemon
```

Running Express.js in development

- Furthermore, we must install the `@babel/node` package, because we are transpiling the back end code with Babel, using the `--exec babel-node` option.
- It allows the use of the import statement:

```
npm install --save-dev @babel/node
```

Running Express.js in development

- Start the server now:

```
npm run server
```

- When you now go to your browser and enter <http://localhost:8000>, you will see the text Hello World! from our Express.js callback function.

Routing in Express.js

- Understanding routing is essential to extend our back end code.
- We are going to play through some simple routing examples.
- In general, routing stands for how an application responds to specific endpoints and methods.
- In Express.js, one path can respond to different HTTP methods and can have multiple handler functions.

Routing in Express.js

- Here is a simple example. Replace this with the current app.get line:

```
app.get('/', function (req, res, next) {  
  console.log('first function');  
  next();  
, function (req, res) {  
  console.log('second function');  
  res.send('Hello World!');  
});
```

Serving our production build

- Again, replace the previous routing example with the following:

```
import path from 'path';
```

```
const root = path.join(__dirname, '../..');
```

```
app.use('/', express.static(path.join(root, 'dist/client')));  
app.use('/uploads', express.static(path.join(root, 'uploads')));  
app.get('/', (req, res) => {  
  res.sendFile(path.join(root, '/dist/client/index.html'));  
});
```

Using Express.js middleware

- Express.js provides great ways to write efficient back ends without duplicating code.
- Every middleware function receives a request, a response, and next. It needs to run next to pass control further to the next handler function. Otherwise, you will receive a timeout.

Using Express.js middleware

- The root path '/' is used to catch any request.

```
app.get('/', function (req, res, next) {
```

- We randomly generate a number with Math.random between 1 and 10.

```
var random = Math.random() * (10 - 1) + 1;
```

Using Express.js middleware

- If the number is higher than 5, we run the `next('route')` function to skip to the next `app.get` with the same path.

```
if (random > 5) next('route')
```

Using Express.js middleware

- If the number is lower than 0.5, we execute the next function without any parameters and go to the next handler function. This handler will log us 'first'.

```
else next()
}, function (req, res, next) {
  res.send('first');
})
app.get('/', function (req, res, next) {
  res.send('second');
})
```

Installing important middleware

- For our application, we have already used one built-in Express.js middleware: express.static.
- Throughout this course, we continue to install further middleware:

```
npm install --save compression cors helmet
```

Installing important middleware

- Now, execute the import statement on the new packages inside the server index.js file so that all dependencies are available within the file:

```
import helmet from 'helmet';
import cors from 'cors';
import compress from 'compression';
```

Express Helmet

- We can enable the Express.js Helmet middleware as follows in the server index.js file:

```
app.use(helmet());
app.use(helmet.contentSecurityPolicy({
  directives: {
    defaultSrc: ["'self'"],
    scriptSrc: ["'self'", "'unsafe-inline'"],
    styleSrc: ["'self'", "'unsafe-inline'"],
    imgSrc: ["'self'", "data:", "*.amazonaws.com"]
  }
}));
app.use(helmet.referrerPolicy({ policy: 'same-origin' }));
```

Compression with Express.js

- Enabling compression for Express.js saves you and your user bandwidth, and this is pretty easy to do.
- The following code must also be added to the server index.js file:

```
app.use(compress());
```

CORS in Express.js

- We want our GraphQL API to be accessible from any website, app, or system.
- A good idea might be to build an app or offer the API to other companies or developers so that they can use it.
- When using APIs via Ajax, the main problem is that the API needs to send the correct Access-Control-Allow-Origin header.

CORS in Express.js

- Allow CORS (Cross-origin resource sharing) requests with the following command to the index.js file:

```
app.use(cors());
```

Combining Express.js with Apollo

- First things first; we need to install the Apollo and GraphQL dependencies:

```
npm install --save apollo-server-express graphql  
graphql-tools
```

Combining Express.js with Apollo

- Create a separate folder for services. A service can be GraphQL or other routes:

```
mkdir src/server/services/
```

```
mkdir src/server/services/graphql
```

Combining Express.js with Apollo

- We require the `apollo-server-express` and `graphql-tools` packages.

```
import { ApolloServer } from 'apollo-server-express';
import { makeExecutableSchema } from 'graphql-tools';
```

Combining Express.js with Apollo

- The GraphQL schema is the representation of the API, that is, the data and functions a client can request or run.
- Resolver functions are the implementation of the schema. Both need to match 100 percent.
- You cannot return a field or run a mutation that is not inside the schema.

```
import Resolvers from './resolvers';
import Schema from './schema';
```

Combining Express.js with Apollo

- The `makeExecutableSchema` function throws an error when you define a query or mutation that is not in the schema.
- The resulting schema is executable by our GraphQL server resolving the data or running the mutations we request.

```
const executableSchema = makeExecutableSchema({  
  typeDefs: Schema,  
  resolvers: Resolvers  
});
```

Combining Express.js with Apollo

- In our resolver functions, we can access the request if we need to.

```
const server = new ApolloServer({  
  schema: executableSchema,  
  context: ({ req }) => req  
});
```

- This index.js file exports the initialized server object, which handles all GraphQL requests.

```
export default server;
```

Combining Express.js with Apollo

- Create an index.js file in the services folder and enter the following code:

```
import graphql from './graphql';
```

```
export default {  
  graphql,  
};
```

Combining Express.js with Apollo

- To make our GraphQL server publicly accessible to our clients, we are going to bind the Apollo Server to the /graphql path.

Import the services index.js file in the server/index.js file as follows:

```
import services from './services';
```

Combining Express.js with Apollo

- The services object only holds the graphql index.
- Now we must bind the GraphQL server to the Express.js web server with the following code:

```
const serviceNames = Object.keys(services);

for (let i = 0; i < serviceNames.length; i += 1) {
  const name = serviceNames[i];
  if (name === 'graphql') {
    services[name].applyMiddleware({ app });
  } else {
    app.use(`/${name}`, services[name]);
  }
}
```

Writing your first GraphQL schema

- Let's start by creating a schema.js inside the graphql folder.
- You can also stitch multiple smaller schemas to one bigger schema.
- This would be cleaner and would make sense when your application, types, and fields grow.
- For this course, one file is okay and we insert the following code into the schema.js file:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/2_1.txt

Implementing GraphQL resolvers

- Create a `resolvers.js` file in the `graphql` folder as follows:

```
const resolvers = {  
  RootQuery: {  
    posts(root, args, context) {  
      return [];  
    },  
    },  
  };
```

```
export default resolvers;
```

Sending GraphQL queries

- You can test our new function when you send the following JSON as a POST request to <http://localhost:8000/graphQL>:

```
{  
  "operationName": null,  
  "query": "{  
    posts {  
      id  
      text  
    }  
  }",  
  "variables": {}  
}
```

Sending GraphQL queries

- You can insert the content of the query property and hit the play button.
- Because we set up Helmet to secure our application, we need to deactivate it in development.
- Otherwise, the GraphQLi instance is not going to work.
- Just wrap the Helmet initialization inside this if statement:

```
if(process.env.NODE_ENV === 'development')
```

Sending GraphQL queries

- The resulting answer of POST should look like the following code snippet:

```
{  
  "data": {  
    "posts": []  
  }  
}
```

Sending GraphQL queries

- Replace the content of the posts function in the GraphQL resolvers with this:

```
return posts;
```

Using multiples types in GraphQL schemas

- Let's create a User type and use it with our posts. First, add it somewhere to the schema:

```
type User {  
  avatar: String  
  username: String  
}
```

- Now that we have a User type, we need to use it inside the Post type. Add it to the Post type as follows:

```
user: User
```

- The user field allows us to have a sub-object inside our posts with the post's author information.
- Our extended query to test this looks like the following:

```
"query":{  
  posts {  
    id  
    text  
    user {  
      avatar  
      username  
    }  
  }  
}"
```

Writing your first GraphQL mutation

- One thing our client already offered was to add new posts to the fake data temporarily.
- We can realize this in the back end by using GraphQL mutations.
- Starting with the schema, we need to add the mutation as well as the input types as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/2_2.txt

Writing your first GraphQL mutation

- The last step is to enable the mutations in our schema for the Apollo Server:

```
schema {  
query: RootQuery  
mutation: RootMutation  
}
```

- The addPost resolver function needs to be implemented now in the resolvers.js file.
- Add the following RootMutation object to the RootQuery in resolvers.js:

```
RootMutation: {  
  addPost(root, { post, user }, context) {  
    const postObject = {  
      ...post,  
      user,  
      id: posts.length + 1,  
    };  
    posts.push(postObject);  
    return postObject;  
  },  
},
```

Writing your first GraphQL mutation

- You can run this mutation via your preferred HTTP client like this:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/2_3.txt

Back end debugging and logging

- There are two things that are very important here: the first is that we need to implement logging for our back end in case we receive errors from our users
- The second is that we need to look into Postman to debug our GraphQL API efficiently.

Logging in Node.js

The most popular logging package for Node.js is called `winston`. Configure `winston` by following the steps below:

- Install `winston` with `npm`:

```
npm install --save winston
```

- We create a new folder for all of the helper functions of the back end:

```
mkdir src/server/helpers
```

Logging in Node.js

- Then, insert a logger.js file in the new folder with the following content:

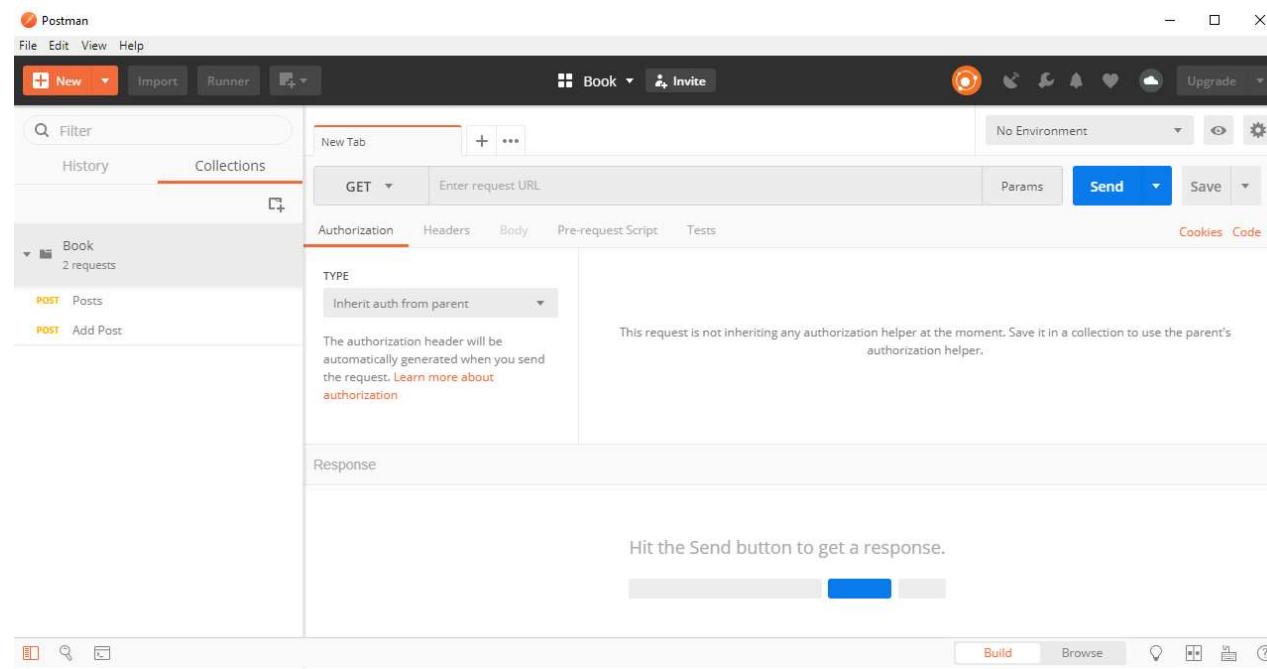
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/2_4.txt

Logging in Node.js

- To test this, we can try the winston logger in the only mutation we have.
- In resolvers.js, add this to the top of the file:
`import logger from '../..../helpers/logger';`
- Now, we can extend the addPost function by logging the following:
`logger.log({ level: 'info', message: 'Post was created' });`

Debugging with Postman

- When you have finished the installation, it should look something like this:



- As an example, the following screenshot shows you how the Add Post mutation looks in Postman:

The screenshot displays the Postman application interface. The main window title is "Postman". The top navigation bar includes "File", "Edit", "View", "Help", "New", "Import", "Runner", and a "+" button. The toolbar features icons for "Book", "Invite", "Upgrade", and various status indicators.

The left sidebar shows a collection named "Book" containing 2 requests: "Posts" and "Add Post". The "Add Post" request is currently selected.

The central workspace shows a "POST" request to "http://localhost:8000/graphql". The "Body" tab is selected, showing the following GraphQL mutation code:

```

1  {
2    "operationName":null,
3    "query": "mutation addPost($post : PostInput!, $user: UserInput!) { addPost(post : $post, user: $user) { id text user { username
4      avatar }}"
4    "variables":{
5      "post": {
6        "text": "You just added a post."
7      },
8      "user": {
9        "avatar": "/uploads/avatar3.png",
10       "username": "Fake User"
11     }
12   }
13 }
```

The "Headers" tab shows one header: "Content-Type: application/json". The "Params" tab is empty. The "Send" and "Save" buttons are visible at the bottom of the request configuration area.

The "Response" section is currently empty.

Summary

- At this point, we have set up our Node.js server with Express.js and bound Apollo Server to respond to requests on a GraphQL endpoint.
- We are able to handle queries, return fake data, and mutate that data with GraphQL mutations.
- Furthermore, we can log every process in our Node.js server.

"Complete Lab 2"

3: Connecting to The Database



Connecting to The Database

This lesson will cover the following points:

- Using databases with GraphQL
- Using Sequelize in Node.js
- Writing database models
- Performing database migrations with Sequelize
- Seeding data with Sequelize
- Using Apollo together with Sequelize

Using databases in GraphQL

- GraphQL is a protocol for sending and receiving data. Apollo is one of the many libraries that you can use to implement that protocol.
- Neither GraphQL (in its specifications) nor Apollo work directly on the data layer.
- Where the data that you put into your response comes from, and where the data that you send with your request is saved, are up to the user to decide.

Installing MySQL for development

- MySQL is an excellent starting point for getting on track in a developmental career. It is also well-suited to local development on your machine, since the setup is pretty easy.
- How to set up MySQL on your machine depends on the operating system.
- As we mentioned in lesson 1, Preparing Your Development Environment, we are assuming that you are using a Debian-based system..

Installing MySQL for development

- First, you should always install all of the updates available for your system:
`sudo apt-get update && sudo apt-get upgrade -y`
- We want to install MySQL and a GUI, in order to see what we have inside of our database.
- The most common GUI for a MySQL server is phpMyAdmin.
- It requires the installation of a web server and PHP,
We are going to install Apache as our web server.

Installing MySQL for development

- Install all dependencies with the following command:
`sudo apt-get install apache2 mysql-server php php-pear
php-mysql`
- After the installation, you will need to run the MySQL setup in the root shell.
- You will have to enter the root password for this.
Alternatively, you can run `sudo -i`:
`SU -`

Installing MySQL for development

- Now, you can execute the MySQL installation command; follow the steps as prompted.
- From my point of view, you can ignore most of these steps, but be careful when you are asked for the root password of your MySQL instance.
- Since this is a development server on your local machine, you can skip the security settings:

mysql_secure_installation

Installing MySQL for development

- We must create a separate user for development, aside from the root and phpMyAdmin user.
- It is discouraged to use the root user at all.
- Log in to our MySQL Server with the root user in order to accomplish this:

```
mysql -u root
```

Installing MySQL for development

- Now, run the following SQL command. You can replace the PASSWORD string with the password that you want.
- It is the password that you will use for the database connection in your application, but also when logging in to phpMyAdmin.
- This command creates a user called devuser, with root privileges that are acceptable for local development:

```
GRANT ALL PRIVILEGES ON *.* TO 'devuser'@'%'  
IDENTIFIED BY 'PASSWORD';
```

Installing MySQL for development

- Furthermore, phpMyAdmin will want you to enter a password.
- I recommend that you choose the same password that you chose for the root user:

```
sudo apt-get install phpmyadmin
```

Installing MySQL for development

- After the installation, we will need to set up Apache, in order to serve phpMyAdmin.
- The following ln command creates a symbolic link in the root folder of the Apache public HTML folder.
- Apache will now serve phpMyAdmin:

```
cd /var/www/html/  
sudo ln -s /usr/share/phpmyadmin
```

phpMyAdmin

Server: localhost:3306

Datenbanken SQL Status Benutzerkonten Exportieren Importieren Einstellungen Mehr

Letzte Favoriten

Neu information_schema mysql performance_schema phpmyadmin sys

Allgemeine Einstellungen

Passwort ändern Zeichensatz/Kollation der MySQL-Verbindung: utf8mb4_unicode_ci

Anzeige-Einstellungen

Sprache - Language: Deutsch - German Design: pmahomme Schriftgröße: 82% Weitere Einstellungen

Datenbank-Server

- Server: Localhost via UNIX socket
- Server-Typ: MySQL
- Server-Version: 5.7.23-0ubuntu0.18.04.1 - (Ubuntu)
- Protokoll-Version: 10
- Benutzer: devuser@localhost
- Server-Zeichensatz: UTF-8 Unicode (utf8)

Webserver

- Apache/2.4.29 (Ubuntu)
- Datenbank-Client Version: libmysql - mysqld 5.0.12-dev - 20150407 - \$Id: 38fea24f2847fa7519001be390c98ae0acafe387 \$
- PHP-Erweiterung: mysqli curl mbstring
- PHP-Version: 7.2.7-0ubuntu0.18.04.2

phpMyAdmin

- Versionsinformationen: 4.6.6deb5
- Dokumentation
- Offizielle Homepage
- Mitmachen
- Unterstützung erhalten
- Liste der Änderungen
- Lizenz

Konsole

The screenshot shows the configuration page of the phpMyAdmin interface. It includes sections for general settings (language, design, font size), database server details (server type: MySQL, version: 5.7.23-0ubuntu0.18.04.1), webserver information (Apache 2.4.29, PHP 7.2.7), and phpMyAdmin links (version, documentation, homepage). The left sidebar lists databases: Neu, information_schema, mysql, performance_schema, phpmyadmin, and sys.

Creating a database in MySQL

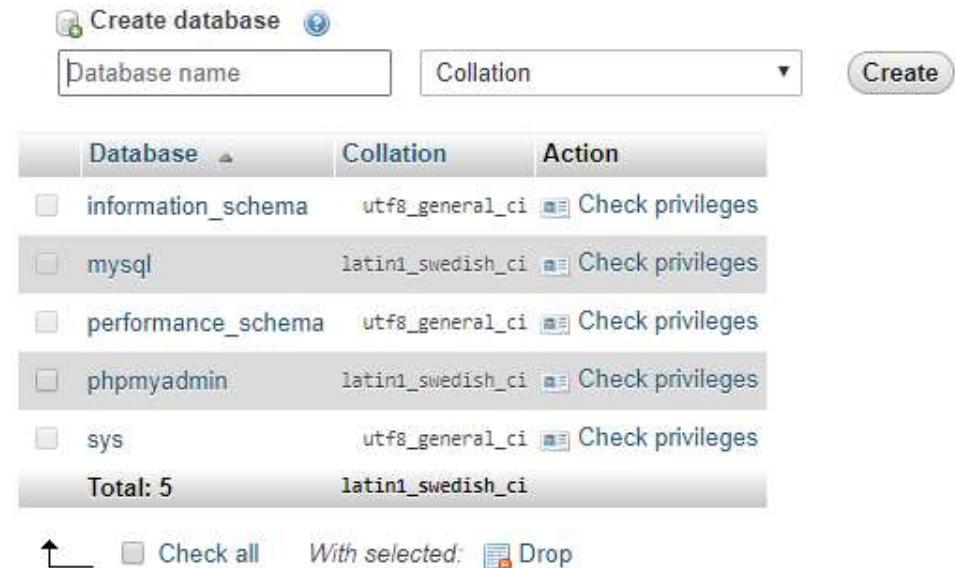
- You can run raw SQL commands in the SQL tab of phpMyAdmin.
- The corresponding command to create a new database looks as follows:

```
CREATE DATABASE graphbook_dev CHARACTER  
SET utf8 COLLATE utf8_general_ci;
```

Creating a database in MySQL

- You will be presented with a screen like the following.
- It shows all databases including their collation of your MySQL server:

Databases



The screenshot shows the MySQL Databases interface. At the top, there is a header with a 'Create database' button, a 'Database name' input field, a 'Collation' dropdown, and a 'Create' button. Below this is a table listing the databases:

Database	Collation	Action
information_schema	utf8_general_ci	<input type="button" value="Check privileges"/>
mysql	latin1_swedish_ci	<input type="button" value="Check privileges"/>
performance_schema	utf8_general_ci	<input type="button" value="Check privileges"/>
phpmyadmin	latin1_swedish_ci	<input type="button" value="Check privileges"/>
sys	utf8_general_ci	<input type="button" value="Check privileges"/>
Total: 5	latin1_swedish_ci	

At the bottom, there are buttons for 'Check all' and 'With selected: Drop'.

Integrating Sequelize into our stack

Install Sequelize in your project via npm.

- We will also install a second package, called mysql2:
`npm install --save sequelize mysql2`
- The mysql2 package allows Sequelize to speak with our MySQL server.
- Sequelize is just a wrapper around the various libraries for the different database systems.
- It offers great features for intuitive model usage, as well as functions for creating and updating database structures and inserting development data.

Connecting to a database with Sequelize

- The first step is to initialize the connection from Sequelize to our MySQL server.
- To do this, we will create a new folder and file, as follows:

```
mkdir src/server/database  
touch src/server/database/index.js
```

Connecting to a database with Sequelize

- Inside of the index.js database, we will establish a connection to our database with Sequelize.
- Internally, Sequelize relies on the mysql2 package, but we do not use it on our own, which is very convenient:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_1.txt

Connecting to a database with Sequelize

- The operatorsAliases property specifies which strings can be used as aliases by Sequelize, or whether they can be used at all.
- An example would look as follows:

[Op.gt]: 6 // > 6

\$gt: 6 // same as using Op.gt (> 6)

Using a configuration file with Sequelize

- For this, create a new index.js file inside a separate folder (called config), next to the database folder:

```
mkdir src/server/config  
touch src/server/config/index.js
```

Using a configuration file with Sequelize

- Your sample configuration should look like the following code, if you have followed the instructions for creating a MySQL database.
- The only thing that we did here was to copy our current configuration into a new object indexed with the development or production environment:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_2.txt

Using a configuration file with Sequelize

- We can remove the configuration that we hardcoded earlier and replace the contents of our index.js database file to require our configfile, instead.
- This should look like the following code snippet:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_3.txt

Writing database models

- After creating a connection to our MySQL server via Sequelize, we want to use it.
- However, our database is missing a table or structure that we can query or manipulate.
- Creating those is the next thing that we need to do.
- Currently, we have two GraphQL entities: User and Post.
- Sequelize lets us create a database schema for each of our GraphQL entities.

Writing database models

- Let's create the first model for our posts.
- Create two new folders (one called models, and the other, migrations) next to the database folder:

```
mkdir src/server/models
```

```
mkdir src/server/migrations
```

Your first database model

- We will use the Sequelize CLI to generate our first database model. Install it globally with the following command:

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

- This gives you the ability to run the sequelize command inside of your Terminal.
- The Sequelize CLI allows us to generate the model automatically. This can be done by running the following command:

```
sequelize model:generate --models-path src/server/models --  
migrations-path src/server/migrations --name Post --attributes  
text:text
```

- The following model file was created for us:

```
'use strict';
```

```
module.exports = (sequelize, DataTypes) => {
  var Post = sequelize.define('Post', {
    text: DataTypes.TEXT
  }, {});
};
```

```
Post.associate = function(models) {
  // associations can be defined here
};
```

```
return Post;
};
```

Your first database migration

A migration file has multiple advantages, such as the following:

1. Migrations allow us to track database changes through our regular version control system, such as Git or SVN. Every change to our database structure should be covered in a migration file.
2. It also enables us to write updates that automatically apply database changes for new versions of our application.

Your first database migration

- Our first migration file creates a Posts table and adds all required columns, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_4.txt

Your first database migration

- To execute this migration, we use the Sequelize CLI again, as follows:

```
sequelize db:migrate --migrations-path  
src/server/migrations --config src/server/config/index.js
```

Your first database migration

- Look inside of phpMyAdmin, Here, you will find the new table, called Posts.
- The structure of the table should look as follows:

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
1	id 	int(11)			No	None		AUTO_INCREMENT
2	text	text	utf8_general_ci		Yes	NULL		
3	createdAt	datetime			No	None		
4	updatedAt	datetime			No	None		

Your first database migration

- Every time that you use Sequelize and its migration feature, you will have an additional table, called SequelizeMeta.
- The contents of the table should look as follows:



Importing models with Sequelize

- We want to import all of our database models at once, in a central file.
- Our database connection instantiator will then use this file on the other side.
- Create an index.js file in the models folder, and fill in the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_5.txt

Importing models with Sequelize

- In development, we must execute the babel-plugin-require-context-hook/register hook to load the require.context function at the top.
- This package must be installed with npm, with the following command:

```
npm install --save-dev babel-plugin-require-context-hook
```

Importing models with Sequelize

- We need to load the plugin with the start of our development server, so, open the package.json file and edit the server script, as follows:

```
nodemon --exec babel-node --plugins require-context-hook --watch src/server src/server/index.js
```

Importing models with Sequelize

- Now, we want to use our models.
- Go back to the index.js database file and import all models through the aggregation index.js file that we just created:

```
import models from '../models';
```

Importing models with Sequelize

- Before exporting the db object at the end of the file, we need to run the models wrapper to read all model .js files.
- We pass our Sequelize instance as a parameter, as follows:

```
const db = {  
  models: models(sequelize),  
  sequelize,  
};
```

Importing models with Sequelize

- We create the global database instance in the index.js file of the root server folder.
- Add the following code:

```
import db from './database';
```

Seeding data with Sequelize

- Create a new folder, called seeders:
`mkdir src/server/seeders`
- Now, we can run our next Sequelize CLI command, in order to generate a boilerplate file:
`sequelize seed:generate --name fake-posts --seeders-path src/server/seeders`

Seeding data with Sequelize

- Seeders are great for importing test data into a database for development.
- Our seed file has the timestamp and the words fake-posts in the name, and should look as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_6.txt

Seeding data with Sequelize

- It is just an empty boilerplate file.
- We need to edit this file to create the fake posts that we already had in our backend.
- This file looks like our migration from the previous section.
- Replace the contents of the file with the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_7.txt

Seeding data with Sequelize

- The down migration bulk deletes all rows in the table, since this is the apparent reverse action of the up migration.
- Execute all of the seeds from the seeders folder with the following command:

```
sequelize db:seed:all --seeders-path src/server/seeders  
--config src/server/config/index.js
```

Seeding data with Sequelize

- The following screenshot shows a filled Posts table:

				id	text	createdAt	updatedAt
<input type="checkbox"/>	 Edit	 Copy	 Delete	1	Lorem ipsum 1	2018-08-13 15:28:40	2018-08-13 15:28:40
<input type="checkbox"/>	 Edit	 Copy	 Delete	2	Lorem ipsum 2	2018-08-13 15:28:40	2018-08-13 15:28:40

Using Sequelize with Apollo

- The database object is initialized upon starting the server within the root index.js file.
- We pass it from this global location down to the spots where we rely on the database. This way, we do not import the database file repeatedly, but have a single instance that handles all database queries for us.
- The services that we want to publicize through the GraphQL API need access to our MySQL database.

Global database instance

- To pass the database down to our GraphQL resolvers, we create a new object in the server index.js file:

```
import db from './database';
```

```
const utils = {  
  db,  
};
```

Global database instance

- Replace the line where we import the services folder, as follows:

```
import servicesLoader from './services';
const services = servicesLoader(utils);
```

Global database instance

- To do this, go to the services index.js file and change the contents of the file, as follows:

```
import graphql from './graphql';
```

```
export default utils => ({  
  graphql: graphql(utils),  
});
```

- Open the index.js file from the graphql folder and replace everything but the require statements at the top with the following code:

```
export default (utils) => {
  const executableSchema = makeExecutableSchema({
    typeDefs: Schema,
    resolvers: Resolvers.call(utils),
  });

  const server = new ApolloServer({
    schema: executableSchema,
    context: ({ req }) => req,
  });

  return server;
};
```

Global database instance

- Surround the resolvers object in this file with a function, and return the resolvers object from inside of the function:

```
export default function resolver() {  
  ...  
  return resolvers;  
}
```

Running the first database query

- Now, we want to finally use the database.
- Add the following code to the top of the export default function resolver statement:

```
const { db } = this;  
const { Post } = db.models;
```

Running the first database query

- We can query all posts through the Sequelize model, instead of returning the fake posts from before.
- Replace the posts property within the RootQuery with the following code:

```
posts(root, args, context) {  
  return Post.findAll({order: [['createdAt', 'DESC']]});  
},
```

Global database instance

- You can start the server with `npm run server` and execute the GraphQL posts query from lesson 2, Setting Up GraphQL with Express.js, again.
- The output will look as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_8.txt

One-to-one relationships in Sequelize

- We need to associate each post with a user, to fill the gap that we have created in our GraphQL response.
- A post has to have an author. It would not make sense to have a post without an associated user.
- First, we will generate a User model and migration. We will use the Sequelize CLI again, as follows:

```
sequelize model:generate --models-path src/server/models --  
migrations-path src/server/migrations --name User --  
attributes avatar:string,username:string
```

Updating the table structure with migrations

- We have to write a third migration, adding the userId column to our Post table, but also including it in our database Post model.
- Generating a boilerplate migration file is very easy with the Sequelize CLI:

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

Updating the table structure with migrations

- You can directly replace the content, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_9.txt

Updating the table structure with migrations

- Rerun the migration, in order to see what changes occurred:

```
sequelize db:migrate --migrations-path  
src/server/migrations --config src/server/config/index.js
```

Updating the table structure with migrations

- Take a look at the relation view of the Posts table in phpMyAdmin.
- You can find it under the Structure view, by clicking on Relation view:

The screenshot shows the 'Relation view' interface in phpMyAdmin. At the top, there are tabs for 'Actions' and 'Constraint properties'. Below these are two rows of constraint configuration fields. The first row contains fields for a foreign key constraint named 'fk_user_id'. It includes dropdowns for 'ON DELETE' (set to 'CASCADE'), 'ON UPDATE' (set to 'CASCADE'), and 'Column' (set to 'userId'). It also includes dropdowns for 'Database' (set to 'graphbook_dev'), 'Table' (set to 'Users'), and 'Column' (set to 'id'). There is also a '+ Add column' button. The second row contains similar fields for another constraint, with 'ON DELETE' set to 'RESTRICT' and 'ON UPDATE' set to 'RESTRICT'. Both rows have a 'Constraint name' field and a '+ Add column' button.

Updating the table structure with migrations

- If you receive an error when running migrations, you can easily undo them, as follows:

```
sequelize db:migrate:undo --migrations-path  
src/server/migrations --config src/server/config/index.js
```

Updating the table structure with migrations

- You can also revert all migrations at once, or only revert to one specific migration, so that you can go back to a specific timestamp:

```
sequelize db:migrate:undo:all --to XXXXXXXXXXXXXXXX-  
create-posts.js --migrations-path src/server/migrations --  
config src/server/config/index.js
```

Model associations in Sequelize

- Now that we have the relationship configured with the foreign key, it also needs to be configured inside of our Sequelize model.
- Go back to the Post model file and replace the associate function with the following:

```
Post.associate = function(models) {  
  Post.belongsTo(models.User);  
};
```

Model associations in Sequelize

- Do not forget to add the userId as a queryable field to the Post model itself, as follows:

`userId: DataTypes.INTEGER,`

- The User model needs to implement the reverse association, too. Add the following code to the User model file:

```
User.associate = function(models) {  
  UserhasMany(models.Post);  
};
```

Model associations in Sequelize

- We must extend our current resolvers.js file.
- Add the Post property to the resolvers object, as follows:

```
Post: {  
  user(post, args, context) {  
    return post.getUser();  
  },  
},
```

Seeding foreign key data

- We use the Sequelize CLI to generate an empty seeders file, as follows:

```
sequelize seed:generate --name fake-users --seeders-path src/server/seeder
```

Seeding foreign key data

- Fill in the following code to insert the fake users:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_10.txt

- We must maintain the relationships as configured in our database.
Adjust the posts seed file to reflect this, and add a userId to both posts in the up migration:

```
return queryInterface.bulkInsert('Posts', [{

  text: 'Lorem ipsum 1',
  userId: usersRows[0].id,
  createdAt: new Date(),
  updatedAt: new Date(),

}, {
  text: 'Lorem ipsum 2',
  userId: usersRows[1].id,
  createdAt: new Date(),
  updatedAt: new Date(),
}],
{});
```

Seeding foreign key data

- There are two options here. You can either manually truncate the tables through phpMyAdmin and SQL statements, or you can use the Sequelize CLI.
- It is easier to use the CLI, but the result will be the same either way. The following command will undo all seeds:

```
sequelize db:seed:undo:all --seeders-path  
src/server/seeders --config src/server/config/index.js
```

Seeding foreign key data

- You can fix this by selecting all users with a raw query in the post seed file.
- We can pass the retrieved user IDs statically.
- Replace the up property with the following:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_11.txt

Seeding foreign key data

- We have not gotten any further now, since the posts are still inserted before the users.
- From my point of view, the easiest way to fix this is to rename the seeder files.
- Simply adjust the timestamp of the fake user seed file to be before the post seed file's timestamp, or the other way around. Again, execute all seeds, as follows:

```
sequelize db:seed:all --seeders-path src/server/seeders --  
config src/server/config/index.js
```

Seeding foreign key data

- If you take a look inside your database, you should see a filled Posts table, including the userId.
- The Users table should look like the following screenshot:

	← T →	▼	id	avatar	username	createdAt	updatedAt			
	<input type="checkbox"/>	<input type="checkbox"/>	Edit	Copy	Delete	1	/uploads/avatar1.png	Test User	2018-08-13 17:04:15	2018-08-13 17:04:15
	<input type="checkbox"/>	<input type="checkbox"/>	Edit	Copy	Delete	2	/uploads/avatar2.png	Test User 2	2018-08-13 17:04:15	2018-08-13 17:04:15

Mutating data with Sequelize

- Requesting data from our database via the GraphQL API works.
- Now comes the tough part: adding a new post to the Posts table.
- Before we start, we must extract the new database model from the db object at the top of the exported function in our resolvers.js file:

```
const { Post, User } = db.models;
```

Mutating data with Sequelize

- We have to edit the GraphQL resolvers to add the new post.
- Replace the old addPost function with the new one, as shown in the following code snippet:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_12.txt

Mutating data with Sequelize

- Everything is set now, To test our API, we are going to use Postman again, We need to change the addPost request.
- The userInput that we added before is not needed anymore, because the backend statically chooses the first user out of our database.
- You can send the following request body:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_13.txt

Mutating data with Sequelize

- Your GraphQL schema must reflect this change, so remove the userInput from there, too:

```
addPost (  
  post: PostInput!  
): Post
```

Mutating data with Sequelize

- Running the addPost GraphQL mutation now adds a post to the Posts table, as you can see in the following screenshot:

			id	text	createdAt	updatedAt	userId
<input type="checkbox"/>	 Edit	 Copy	 Delete	1	Lorem ipsum 1	2018-08-14 11:08:28	2018-08-14 11:08:28
<input type="checkbox"/>	 Edit	 Copy	 Delete	2	Lorem ipsum 2	2018-08-14 11:08:28	2018-08-14 11:08:28
<input type="checkbox"/>	 Edit	 Copy	 Delete	3	You just added a post.	2018-08-14 11:08:46	2018-08-14 11:08:46

Many-to-many relationships

- Facebook provides users with various ways to interact.
- Currently, we only have the opportunity to request and insert posts.
- As in the case of Facebook, we want to have chats with our friends and colleagues.
- We will introduce two new entities to cover this.

Model and migrations

- When transferring this into real code, we first generate the Chat model.
- The problem here is that we have a many-to-many relationship between users and chats.
- In MySQL, this kind of relationship requires a table, to store the relations between all entities separately.

Chat model

- Let's start by creating the Chat model and migration.
- A chat itself does not store any data; we use it for grouping specific users' messages:

```
sequelize model:generate --models-path  
src/server/models --migrations-path  
src/server/migrations --name Chat --attributes  
firstName:string,lastName:string,email:string
```

Chat model

- Generate the migration for our association table, as follows:

```
sequelize migration:create --migrations-path  
src/server/migrations --name create-user-chats
```

Chat model

- References inside of a migration automatically create foreign key constraints for us.
- The migration file should look like the following code snippet:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_14.txt

Chat model

- Associate the user to the Chat model via the new relation table in the User model, as follows:

```
User.belongsToMany(models.Chat, { through:  
'users_chats' });
```

- Do the same for the Chat model, as follows:

```
Chat.belongsToMany(models.User, { through:  
'users_chats' });
```

- Rerun the migrations to let the changes take effect:
`sequelize db:migrate --migrations-path
src/server/migrations --config src/server/config/index.js`
- The following screenshot shows how your database should look now:

Table	Action	Rows	Type	Collation	Size	Overhead
Chats		0	InnoDB	utf8_general_ci	16 KiB	-
Posts		3	InnoDB	utf8_general_ci	32 KiB	-
SequelizeMeta		5	InnoDB	utf8_unicode_ci	32 KiB	-
Users		2	InnoDB	utf8_general_ci	16 KiB	-
users_chats		0	InnoDB	utf8_general_ci	48 KiB	-
5 tables	Sum	10	InnoDB	utf8_general_ci	144 KiB	0 B

Chat model

- You should see two foreign key constraints in the relation view of the users_chats table.
- The naming is done automatically:

Actions Constraint properties		Column <small>(i)</small>	Foreign key constraint (INNODB)			
		Database	Table	Column		
<input type="button" value="Drop"/>	users_chats_ibfk_1	ON DELETE	RESTRICT	ON UPDATE	RESTRICT	userId
<input type="button" value="Drop"/>	users_chats_ibfk_2	ON DELETE	RESTRICT	ON UPDATE	RESTRICT	chatId

Message model

- A message is much like a post, except that it is only readable inside of a chat, and is not public to everyone.
- Generate the model and migration file with the CLI, as follows:

```
sequelize model:generate --models-path  
src/server/models --migrations-path src/server/migrations  
--name Message --attributes  
text:string,userId:integer,chatId:integer
```

- Add the missing references in the created migration file, as follows:

```
userId: {
  type: Sequelize.INTEGER,
  references: {
    model: 'Users',
    key: 'id'
  },
  onDelete: 'SET NULL',
  onUpdate: 'cascade',
},
chatId: {
  type: Sequelize.INTEGER,
  references: {
    model: 'Chats',
    key: 'id'
  },
  onDelete: 'cascade',
  onUpdate: 'cascade',
},
```

- Now, we can run the migrations again, in order to create the Messages table using the sequelize db:migrate Terminal command.

```
{  
  "operationName":null,  
  "query": "mutation addPost($post : PostInput!) { addPost(post :  
$post) {  
    id text user { username avatar }}}}","  
  "variables":{  
    "post": {  
      "text": "You just added a post."  
    }  
  }  
}
```

Message model

- Replace the associate function of the Message model with the following code:

```
Message.associate = function(models) {  
  Message.belongsTo(models.User);  
  Message.belongsTo(models.Chat);  
};
```

Message model

- In the preceding code, we define that every message is related to exactly one user and chat.
- On the other side, we must also associate the Chat model with the messages.
- Add the following code to the associate function of the Chat model:

`ChathasMany(models.Message);`

Chats and messages in GraphQL

- We have introduced some new entities with messages and chats.
- Let's include those in our Apollo schema.
- In the following code, you can see an excerpt of the changed entities, fields, and parameters of our GraphQL schema:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_15.txt

Chats and messages in GraphQL

- These factors should be implemented in our resolvers, too.
- Our resolvers should look as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_16.txt

Chats and messages in GraphQL

- It is important that we are using the new models here.
- We should not forget to extract them from the db.models object inside of the resolver function.
- It must look as follows:

```
const { Post, User, Chat, Message } = db.models;
```

Chats and messages in GraphQL

- You can send this GraphQL request to test the changes:

```
{  
  "operationName":null,  
  "query": "{ chats { id users { id } messages { id text  
    user { id username  
      } } } }",  
  "variables":{}  
}
```

Chats and messages in GraphQL

- The response should give us an empty chats array, as follows:

```
{  
  "data": {  
    "chats": []  
  }  
}
```

Seeding many-to-many data

- Testing our implementation requires data in our database.
- We have three new tables, so we will create three new seeders, in order to get some test data to work with.
- Let's start with the chats, as follows:

```
sequelize seed:generate --name fake-chats --seeders-path src/server/seeders
```

Seeding many-to-many data

- Now, replace the new seeder file with the following code.
- Running the following code creates a chat in our database.
- We do not need more than two timestamps, because the chat ID is generated automatically:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_17.txt

Seeding many-to-many data

- Next, we insert the relation between two users and the new chat.
- We do this by creating two entries in the `users_chats` table where we reference them.
- Now, generate the boilerplate seed file, as follows:
`sequelize seed:generate --name fake-chats-users-relations --seeders-path src/server/seeders`
- Our seed should look much like the previous ones, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_18.txt

Seeding many-to-many data

- The last table without any data is the Messages table. Generate the seed file, as follows:

```
sequelize seed:generate --name fake-messages --  
seeders-path src/server/seeders
```

- Again, replace the generated boilerplate code, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_19.txt

Seeding many-to-many data

- Try to run the GraphQL chats query again, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_20.txt

Seeding many-to-many data

- Add a RootQuery chat that takes a chatId as a parameter:

```
chat(root, { chatId }, context) {  
    return Chat.findById(chatId, {  
        include: [{  
            model: User,  
            required: true,  
        },  
        {  
            model: Message,  
        },  
    });  
},
```

- Add the new query to the GraphQL schema, under RootQuery:

chat(chatId: Int): Chat

- Send the GraphQL request to test the implementation, as follows:

```
{  
  "operationName":null,  
  "query": "query($chatId: Int!){ chat(chatId: $chatId) {  
    id users { id } messages { id text user { id username } } } }",  
  "variables":{ "chatId": 1 }  
}
```

Creating a new chat

- Add the addChat function to the RootMutation in the resolvers.js file, as follows:

```
addChat(root, { chat }, context) {
  logger.log({
    level: 'info',
    message: 'Message was created',
  });
  return Chat.create().then((newChat) => {
    return Promise.all([
      newChat.setUsers(chat.users),
    ]).then(() => {
      return newChat;
    });
  });
},
```

Creating a new chat

- We have to add the new input type and mutation, as follows:

```
input ChatInput {  
    users: [Int]  
}
```

```
type RootMutation {  
    addPost (  
        post: PostInput!  
    ): Post  
    addChat (  
        chat: ChatInput!  
    ): Chat  
}
```

Creating a new chat

- Test the new GraphQL addChat mutation as your request body:

```
{  
  "operationName":null,  
  "query": "mutation addChat($chat: ChatInput!) { addChat(chat:  
    $chat) { id users { id } }}",  
  "variables":{  
    "chat": {  
      "users": [1, 2]  
    }  
  }  
}
```

Creating a new message

- Add the `addMessage` function to the `RootMutation` in the `resolvers.js` file, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/3_21.txt

Then, add the new mutation to your GraphQL schema.

- We also have a new input type for our messages:

```
input MessageInput {  
    text: String!  
    chatId: Int!  
}  
  
type RootMutation {  
    addPost (  
        post: PostInput!  
    ): Post  
    addChat (  
        chat: ChatInput!  
    ): Chat  
    addMessage (  
        message: MessageInput!  
    ): Message  
}
```

Creating a new message

- You can send the request in the same way as the addPost request:

```
{  
  "operationName":null,  
  "query": "mutation addMessage($message : MessageInput!) {  
    addMessage(message : $message) { id text }}",  
  "variables":{  
    "message": {  
      "text": "You just added a message.",  
      "chatId": 1  
    }  
  }  
}
```

Summary

- Our goal in this lesson was to create a working backend with a database as storage, which we have achieved pretty well.
- We can add further entities and migrate and seed them with Sequelize.
- Migrating our database changes won't be a problem for us when it comes to going into production.

"Complete Lab 3"

4: Integrating React into the Back end with Apollo



Integrating React into the Back end with Apollo

This lesson will cover the following points:

- Installing and configuring Apollo Client
- Sending requests with GQL and Apollo's Query component
- Mutating data with Apollo
- Debugging with Apollo Client Developer Tools

Setting up Apollo Client

- To start, we must install the React Apollo Client library.
- Apollo Client is a GraphQL client that offers excellent integration with React, and the ability to easily fetch data from our GraphQL API.
- Furthermore, it handles actions such as caching and subscriptions, to implement real-time communication with your GraphQL back end.

Installing Apollo Client

- We use npm to install our client dependencies, as follows:

```
npm install --save apollo-client apollo-cache-inmemory  
apollo-link-http apollo-link-error apollo-link react-apollo
```

Installing Apollo Client

- To get started with the manual setup of the Apollo Client, create a new folder and file for the client, as follows:

```
mkdir src/client/apollo  
touch src/client/apollo/index.js
```

Installing Apollo Client

- Just insert the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_1.txt

Testing the Apollo Client

- First, install this package with npm, as follows:
`npm install --save graphql-tag`
- Import the package at the top of the Apollo Client setup, as follows:
`import gql from 'graphql-tag';`

- Then, add the following code before the client is exported:

```
client.query({
  query: gql` 
    posts {
      id
      text
      user {
        avatar
        username
      }
    }
  `
}).then(result => console.log(result));
```

Testing the Apollo Client

- However, we have forgotten something: the client is set up in our new file, but it is not yet used anywhere.
- Import it in the index.js root file of our client React app, below the import of the App class:

```
import client from './apollo';
```

Testing the Apollo Client

- The output should look like the following screenshot:

```
▼ {data: {...}, loading: false, networkStatus: 7, stale: false} ⓘ
  ▼ data:
    ▼ posts: Array(2)
      ▼ 0:
        id: 1
        text: "Lorem ipsum 1"
        ▼ user:
          avatar: "/uploads/avatar1.png"
          username: "Test User"
          __typename: "User"
          Symbol(id): "$Post:1.user"
          ► __proto__: Object
          __typename: "Post"
          Symbol(id): "Post:1"
          ► __proto__: Object
      ▼ 1:
        id: 2
        text: "Lorem ipsum 2"
        ▼ user:
          avatar: "/uploads/avatar2.png"
          username: "Test User 2"
          __typename: "User"
          Symbol(id): "$Post:2.user"
          ► __proto__: Object
          __typename: "Post"
          Symbol(id): "Post:2"
          ► __proto__: Object
          length: 2
          ► __proto__: Array(0)
          Symbol(id): "ROOT_QUERY"
          ► __proto__: Object
          loading: false
          networkStatus: 7
          stale: false
          ► __proto__: Object
```

Binding the Apollo Client to React

- We have tested Apollo Client, and have confirmed that it works.
- However, React does not yet have access to it.
- Since Apollo Client is going to be used everywhere in our application, we can set it up in our root index.js file, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_2.txt

Using the Apollo Client in React

Follow the instructions below to connect your first React component with the Apollo Client:

- Clone the App.js file to another file, called Feed.js.
- Remove all parts where React Helmet is used, and rename the class Feed, instead of App.
- From the App.js file, remove all of the parts that we have left in the Feed class.

Using the Apollo Client in React

- Furthermore, we must render the Feed class inside of the App class.
- It should like the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_3.txt

Querying in React with the Apollo Client

- There are two main approaches offered by Apollo that can be used to request data.
- The first one is a higher-order component (HoC), provided by the react-apollo package.
- The second one is the Query component of Apollo, which is a special React component.
- Both approaches have their advantages and disadvantages.

Apollo HoC query

- A higher-order component is a function that takes a component as input and returns a new component.
- This method is used in many cases wherein we have multiple components all relying on the same functionalities, such as querying for data.

Using the Apollo Client in React

To see a real example of this, we use the posts feed.
Follow these instructions to get a working Apollo Query HoC:

- Remove the demo posts from the top of the Feed.js file.
- Remove the posts field from the state initializer.
- Import graphl-tag and parse our query with it, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_4.txt

Using the Apollo Client in React

- Replace everything in the render function, before the final return statement, with the following code:

```
const { posts, loading, error } = this.props;  
const { postContent } = this.state;
```

```
if(loading) {  
  return "Loading...";  
}  
if(error) {  
  return error.message;  
}
```

Using the Apollo Client in React

- Remove the export statement from the Feed class.
- We will export the new component returned from the HoC at the end of the file.
- The export must look as follows:

```
export default graphql(GET_POSTS, {  
  props: ({ data: { loading, error, posts } }) => ({  
    loading,  
    posts,  
    error  
  })  
})(Feed)
```

The Apollo Query component

- We will now take a look at the second approach, which is also the approach of the official Apollo documentation.
- Before getting started, undo the HoC implementation to send requests from the previous section.
- The new way of fetching data through the Apollo Client is via render props, or render functions

The Apollo Query component

- Remove the demo posts from the top of the Feed.js file.
- Remove the posts from the state and stop extracting them from the component state in the render method, too.
- Import the Query component from the react-apollo package and graphl-tag, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_5.txt

The Apollo Query component

- The Query component can now be rendered.
- The only parameter, for now, is the parsed query that we want to send.
- Replace the complete render method with the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_6.txt

Mutations with the Apollo Client

- We have replaced the way that we get the data in our client, The next step is to switch the way that we create new posts, too.
- Before Apollo Client, we had to add the new fake posts to the array of demo posts manually, within the memory of the browser.
- Now, everything in our text area is sent with the addPost mutation to our GraphQL API, through Apollo Client.

The Apollo Mutation HoC

Follow these instructions to set up the mutation HoC:

- Import the compose method from the react-apollo package, as follows:

```
import { graphql, compose } from 'react-apollo';
```

- Add the addPost mutation and parse it with graphql-tag:

```
const ADD_POST = gql`  
mutation addPost($post : PostInput!) {  
  addPost(post : $post) {  
    id  
    text  
    user {  
      username  
      avatar  
    }  
  }  
};`;
```

The Apollo Mutation HoC

- Now, we will use the compose function of react-apollo, which takes a set of GraphQL queries, or mutations.
- These are run or passed as functions to the component.
- Add the following code to the bottom, and remove the old export statement:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_7.txt

The Apollo Mutation HoC

- The addPost function is available under the properties of the Feed component, as we specified it in the preceding code.
- When giving a variables object as a parameter, we can fill in our input fields in the mutation, as is expected by our GraphQL schema:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_8.txt

The Apollo Mutation component

- Import the Mutation component from the react-apollo package, as follows:

```
import { Query, Mutation } from 'react-apollo';
```

- Export the Feed component again, as we did in the previous Query component example.
- Remove the ADD_POST_MUTATION and GET_POSTS_QUERY variables when doing so:

```
export default class Feed extends Component
```

The Apollo Mutation component

- Next, add the Mutation component inside of the render function.
- We will surround the form with this component, as this is the only part of our content where we need to send a mutation:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_9.txt

Updating the UI with the Apollo Client

We use these solutions in different scenarios.

- Let's take a look at some examples, Refetching makes sense if further logic is implemented on the server, which is hidden from the client when requesting a list of items, and which is not applied when inserting only one item.
- In these cases, the client cannot simulate the state of the typical response of a server.
- Updating the cache, however, makes sense when adding or updating items in a list, like our post feed.
- The client can insert the new post at the top of the feed.

Refetching queries

- As mentioned previously, this is the easiest method to update your user interface.
- The only step is to set an array of queries to be refetched.
- The Mutation component should look as follows:

```
<Mutation  
refetchQueries={[{query: GET_POSTS}]}
```

Updating the Apollo cache

- We want to explicitly add only the new post to the cache of the Apollo Client.
- Using the cache helps us to save data, by not refetching the complete feed or rerendering the complete list.
- To update the cache, you should remove the refetchQueries property.
- You can then introduce a new property, called update, as shown in the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_10.txt

The Apollo Mutation component

- The advantage is that the user can see the new result, instead of waiting for the response of the server.
- This solution makes the application feel faster and more responsive.
- This section expects the update function of the Mutation component to already be implemented.
- Otherwise, this UI feature will not work. We need to add the optimisticResponse property to our mutation, as follows:

Refer to the file 4_11.txt

- To set a particular class on the list item, we conditionally set the correct className in our map loop.
- Insert the following code into the render method:

```
{posts.map((post, i) =>
  <div key={post.id} className={'post ' + (post.id < 0 ? 'optimistic':
    '')}>
    <div className="header">
      <img src={post.user.avatar} />
      <h2>{post.user.username}</h2>
    </div>
    <p className="content">
      {post.text}
    </p>
  </div>
)}
```

The Apollo Mutation component

- An example CSS style for this might look as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_12.txt

The Apollo Mutation component

- CSS animations make your applications more modern and flexible.
- If you experience issues when viewing these in your browser, you may need to check whether your browser supports them.
- You can see the result in the following screenshot:



Polling with the Query component

Polling is nothing more than rerunning a request after a specified interval. This procedure is the simplest way to implement real-time updates for our news feed. However, multiple issues are associated with polling, as follows:

- It is inefficient to send requests without knowing whether there is any new data. The browser might send dozens of requests without ever receiving a new post.

Polling with the Query component

- There are some use cases in which polling makes sense.
- One example is a real-time graph, in which every axis tick is displayed to the user, whether there is data or not.
- You do not need to use an interrupt-based solution, since you want to show everything.
- Despite the issues that come with polling, let's quickly run through how it works.
- All you need to do is fill in the pollInterval property, as follows:

```
<Query query={GET_POSTS} pollInterval={5000}>
```

Implementing chats and messages

- In the previous lesson, we programmed a pretty dynamic way of creating chats and messages with your friends and colleagues, either one-on-one or in a group.
- There are some things that we have not discussed yet, such as authentication, real-time subscriptions, and friend relationships.

Fetching and displaying chats

- Our news feed is working as we expected. Now, we also want to cover chats.
- As with our feed, we need to query for every chat that the current user (or, in our case, the first user) is associated with.
- The initial step is to get the rendering working with some demo chats.
- Instead of writing the data on our own, as we did in the first lesson, we can now execute the chats query.

Fetching and displaying chats

- Send the GraphQL query. The best options involve Apollo Client Developer Tools, if you already know how they work.
- Otherwise, you can rely on Postman, as you did previously:

```
query {  
  chats {  
    id  
    users {  
      avatar  
      username  
    }  
  }  
}
```

- Copy the complete response over to an array inside of the Chats.js file, as follows. Add it to the top of the file:

```
const chats = [{}  
  "id": 1,  
  "users": [{}  
    "id": 1,  
    "avatar": "/uploads/avatar1.png",  
    "username": "Test User"  
  },  
  {}  
    "id": 2,  
    "avatar": "/uploads/avatar2.png",  
    "username": "Test User 2"  
  ]]  
}
```

Fetching and displaying chats

- Import React ahead of the chats variable.
- Otherwise, we will not be able to render any React components:

```
import React, { Component } from 'react';
```

Fetching and displaying chats

Set up the React Component. I have provided the basic markup here.

- Just copy it beneath the chats variable.
- I am going to explain the logic of the new component shortly:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_13.txt

Fetching and displaying chats

- To save the file size in our CSS file, replace the two post header styles to also cover the style of the chats, as follows:

```
.post .header > *, .chats .chat .header > * {  
    display: inline-block;  
    vertical-align: middle;  
}
```

```
.post .header img, .chats .chat .header img {  
    width: 50px;  
    margin: 5px;  
}
```

Fetching and displaying chats

- We must append the following CSS to the bottom of the style.css file:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_14.txt

Fetching and displaying chats

- To get the code working, we must also import the Chats class in our App.js file:

```
import Chats from './Chats';
```

- Render the Chats class inside of the render method, beneath the Feed class inside of the App.js file.

- The current code generates the following screenshot:

The screenshot shows a user interface for a social media or blog platform. At the top, there is a text input field with the placeholder "Write your custom post!" and a blue "Submit" button. Below this, there is a list of posts:

- A post by "Test User" with the content "This is a test post!".
- A post by "Test User" with the content "Lorem ipsum 1".
- A post by "Test User 2" with the content "Lorem ipsum 2".

On the right side of the screen, there is a sidebar with a profile picture of a person with red hair and the text "Test User 2".

Fetching and displaying chats

- Instead, we will add a new property to the chat entity, called `lastMessage`.
- That way, we will only get the newest message.
- We will add the new field to the GraphQL schema of our chat type, in the back end code, as follows:

`lastMessage: Message`

Fetching and displaying chats

- If you return the promise directly, you will receive null in the response from the server, because an array is not a valid response for a single message entity:

```
lastMessage(chat, args, context) {  
    return chat.getMessages({limit: 1, order: [['id',  
'DESC']]})  
.then((message) => {  
    return message[0];  
});  
},
```

Fetching and displaying chats

- You can add the new property to our static data, inside of Chats.js.
- Rerunning the query (as we did in step 1) would also be possible:

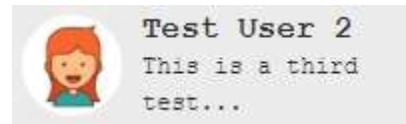
```
"lastMessage": {  
  "text": "This is a third test message."  
}
```

Fetching and displaying chats

- We can render the new message with a simple span tag beneath the h2 of the username.
- Copy it directly into the render method, inside of our Chats class:

```
<span>{this.shorten(chat.lastMessage.text)}</span>
```

- The result of the preceding changes renders every chat row with the last message inside of the chat.
- This looks like the following screenshot:



Fetching and displaying chats

- Since everything is displayed correctly from our test data, we can introduce the Query component, in order to fetch all of the data from our GraphQL API, We can remove the chats array.
- Then, we will import all of the dependencies and parse the GraphQL query, as in the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_15.txt

Fetching and displaying chats

- Our new render method does not change much.
- We just include the Apollo Query component, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_16.txt

Fetching and displaying messages

- We will start with the Query component from the beginning.
- First, however, we have to store the chats that were opened by a click from the user.
- Every chat is displayed in a separate, small chat window, like in Facebook.
- Add a new state variable to save the ids of all of the opened chats to the Chats class:

```
state = {  
  openChats: []  
}
```

- To let our component insert something into the array of open chats, we will add the new openChat method to our Chats class:

```
openChat = (id) => {
  var openChats = this.state.openChats.slice();

  if(openChats.indexOf(id) === -1) {
    if(openChats.length > 2) {
      openChats = openChats.slice(1);
    }
    openChats.push(id);
  }

  this.setState({ openChats });
}
```

Fetching and displaying messages

- The last step is to bind the onClick event to our component.
- In the map function, we can replace the wrapping div tag with the following line:

```
<div key={"chat" + chat.id} className="chat"  
onClick={() => self.openChat(chat.id)}>
```

Fetching and displaying messages

- Add a surrounding wrapper div tag to the whole render method:

```
<div className="wrapper">
```

- We insert the new markup for the open chats next to the chats panel.
- You cannot insert it inside the panel directly, due to the CSS that we are going to use:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_17.txt

Fetching and displaying messages

- As you can see in the preceding code, we are not only passing the chat id as a parameter to the variables property of the Query component, but we also use another query stored in the GET_CHAT variable.
- We must parse this query first, with graphql-tag. Add the following code to the top of the file:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_18.txt

Fetching and displaying messages

- Because we rely on the `openChats` state variable, we must extract it in our render method.
- Add the following code before the return state, in the render method:

```
const self = this;  
const { openChats } = this.state;
```

Fetching and displaying messages

- The close button function relies on the closeChat method, which we will implement in our Chats class:

```
closeChat = (id) => {
  var openChats = this.state.openChats.slice();

  const index = openChats.indexOf(id);
  openChats.splice(index, 1),

  this.setState({ openChats });
}
```

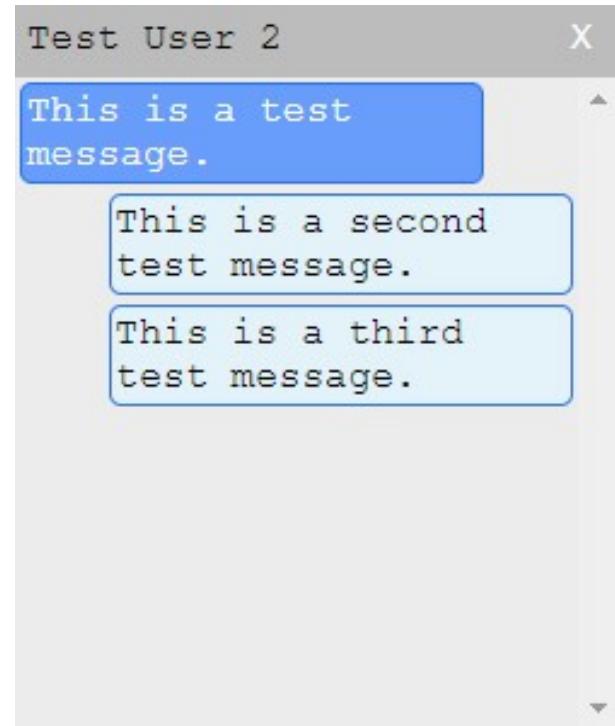
Fetching and displaying messages

- The last thing missing is some styling. The CSS is pretty big.
- Every message from the other users should be displayed on the left, and our own messages on the right, in order to differentiate them.
- Insert the following CSS into the style.css file:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_19.txt

Fetching and displaying messages

- Take a look at the following screenshot:



Sending messages through Mutations

- First, parse the mutation at the top, next to the other requests:

```
const ADD_MESSAGE = gql`  
  mutation addMessage($message : MessageInput!) {  
    addMessage(message : $message) {  
      id  
      text  
      user {  
        id  
      }  
    }  
  }  
`;
```

Sending messages through Mutations

- For each open chat, we will have one input where the user can type his message.
- There are multiple solutions to save all of the inputs' text inside the React component's state.
- For now, we will keep it simple, but we will take a look at a better way to do this in the lesson 5, Reusable React Components.
- Open a new object inside of the state initializer in our Chats class:

`textInputs: {}`

Sending messages through Mutations

- Import the Mutation component from the react-apollo package, as follows:

```
import { Query, Mutation } from 'react-apollo';
```

- Replace the existing openChat and closeChat methods with the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_20.txt

Sending messages through Mutations

- Now, we must also handle the change event of the input by implementing a special function, as follows:

```
onChangeChatInput = (event, id) => {
  event.preventDefault();
  var textInputs = Object.assign({}, this.state.textInputs);
  textInputs[id] = event.target.value;
  this.setState({ textInputs });
}
```

Sending messages through Mutations

- The Mutation component is rendered before the input, so that we can pass the mutation function to the input.
- The input inside receives the onChange property, in order to execute the onChangeChatInput function while typing:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_21.txt

- The implementation of the handleKeyPress method is pretty straightforward. Just copy it into our component, as follows:

```
handleKeyPress = (event, id, addMessage) => {
  const self = this;
  var textInputs = Object.assign({}, this.state.textInputs);

  if (event.key === 'Enter' && textInputs[id].length) {
    addMessage({ variables: { message: { text: textInputs[id], chatId: id } } }).then(() => {
      textInputs[id] = '';
      self.setState({ textInputs });
    });
  }
}
```

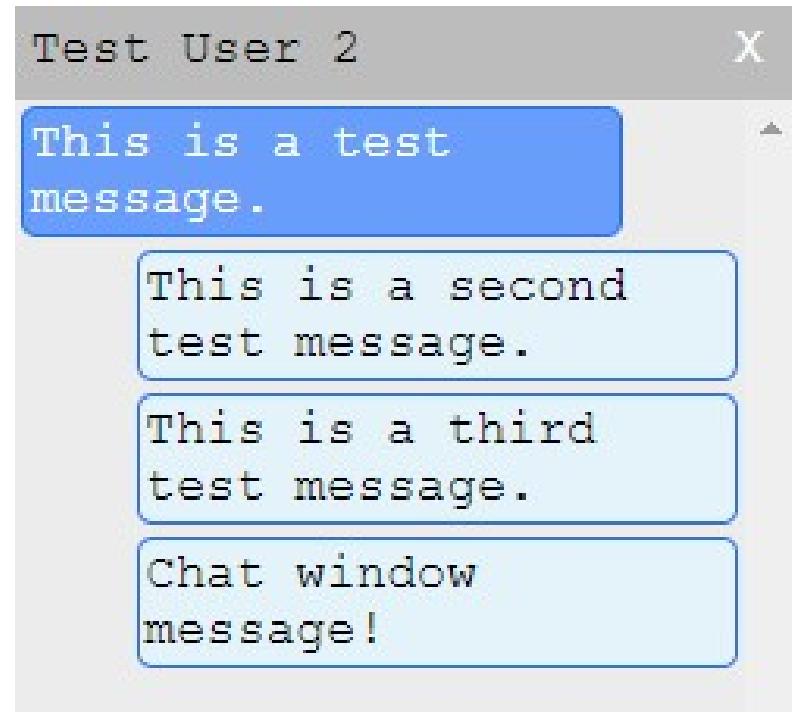
Sending messages through Mutations

- Let's quickly add some CSS to our style.css file, to make the input field look good:

```
.chatWindow .input self. SetState {  
    width: calc(100% - 4px);  
    border: none;  
    padding: 2px;  
}  
.chatWindow .input input:focus {  
    outline: none;  
}
```

Sending messages through Mutations

- The following screenshot shows the chat window, with a new message inserted through the chat window input:



Pagination in React and GraphQL

- By pagination, most of the time, we mean the batch querying of data.
- Currently, we query for all posts, chats, and messages in our database.
- If you think about how much data Facebook stores inside one chat with your friends, you will realize that it is unrealistic to fetch all of the messages and data ever shared at once.

Pagination in React and GraphQL

- Add a new RootQuery to our GraphQL schema, as follows:

postsFeed(page: Int, limit: Int): PostFeed

- The PostFeed type only holds the posts field. Later on, in the development of the application, you can return more information, such as the overall count of items, the page count, and so on:

```
type PostFeed {  
  posts: [Post]  
}
```

Pagination in React and GraphQL

- Next, we must implement the PostFeed entity in our resolvers.js file.
- Copy the new resolver function over to the resolvers file, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_22.txt

Pagination in React and GraphQL

- Our front end needs some adjustments to support pagination.
- Install a new React package with npm, which provides us with an infinite scroll implementation:

```
npm install react-infinite-scroller –save
```

Pagination in React and GraphQL

- You are free to program this on your own, but we are not going to cover that here.
- Go back to the Feed.js file, replace the GET_POSTS query, and import the react-infinite-scroller package, with the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_23.txt

Pagination in React and GraphQL

- Since the postsFeed query expects parameters other than the standard query from before, we need to edit our Query component in the render method.
- The changed lines are as follows:

```
<Query query={GET_POSTS} variables={{page: 0, limit: 10}}>
```

```
 {{ loading, error, data, fetchMore }} => {  
   if (loading) return <p>Loading...</p>;  
   if (error) return error.message;  
 };
```

```
const { postsFeed } = data;
```

```
const { posts } = postsFeed;
```

Pagination in React and GraphQL

- According to the new data structure defined in our GraphQL schema, we extract the posts array from the postsFeed object.
- Replace the markup of the div tag of our current feed to make use of our new infinite scroll package:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_24.txt

Pagination in React and GraphQL

- It is important that we initialize the hasMore and page index state variable in our class first.
- Insert the following code:

```
state = {  
  postContent: "",  
  hasMore: true,  
  page: 0,  
}
```

Pagination in React and GraphQL

- Of course, we must also extract the `hasMore` variable in the render method of our class:

```
const { postContent, hasMore } = this.state;
```

Pagination in React and GraphQL

- We need to implement the loadMore function before running the infinite scroller.
- It relies on the page variable that we just configured.
- The loadMore function should look like the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/4_25.txt

Pagination in React and GraphQL

- We have a second layer, postsFeed, as the parent of the posts array.
- Change the update function to get it working again, as follows:

```
update = {(store, { data: { addPost } }) => {
  const variables = { page: 0, limit: 10 };
  const data = store.readQuery({ query: GET_POSTS,
variables });
  data.postsFeed.posts.unshift(addPost);
  store.writeQuery({ query: GET_POSTS, variables, data });
}}
```

Debugging with the Apollo Client Developer Tools

- Apollo Client Developer Tools is another Chrome extension, allowing you to send Apollo requests.
- While Postman is great in many ways, it does not integrate with our application, and does not implement any GraphQL-specific features.
- Apollo Client Developer Tools rely on the Apollo Client that we set up very early on in this lesson.

A

GraphiQL

GraphQL Queries Mutations Cache

query { posts { id text user { avatar username } } }

Query Variables 1

```
1 * query {
2 *   posts {
3 *     id
4 *     text
5 *     user {
6 *       avatar
7 *       username
8 *     }
9 *   }
10 }
```

```
* {
*   "data": {
*     "posts": [
*       {
*         "id": 3,
*         "text": "This is a test post!",
*         "user": {
*           "avatar": "/uploads/avatar1.png",
*           "username": "Test User"
*         }
*       },
*       {
*         "id": 1,
*         "text": "Lorem ipsum 1",
*         "user": {
*           "avatar": "/uploads/avatar1.png",
*           "username": "Test User"
*         }
*       },
*       {
*         "id": 2,
*         "text": "Lorem ipsum 2",
*         "user": {
*           "avatar": "/uploads/avatar2.png",
*           "username": "Test User 2"
*         }
*       }
*     ]
*   }
* }
```

Documentation Explorer

Search Schema...

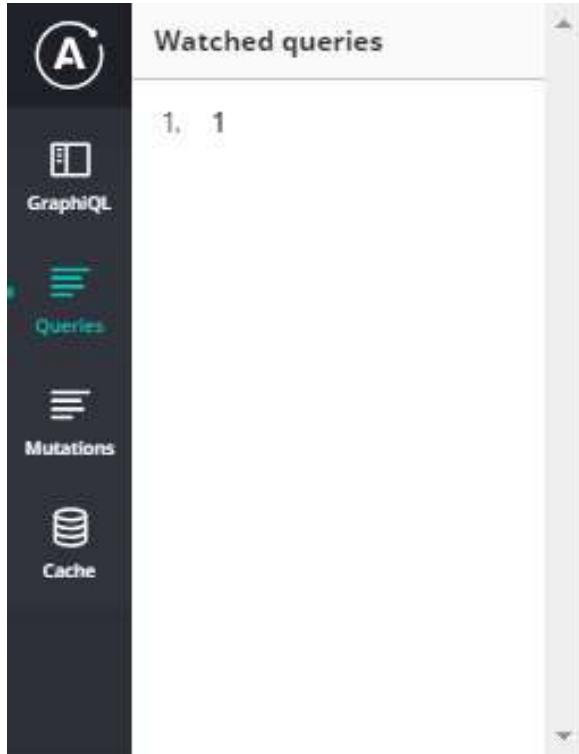
A GraphQL schema provides a root type for each kind of operation.

ROOT TYPES

query: Query

mutation: Mutation

Debugging with the Apollo Client Developer Tools

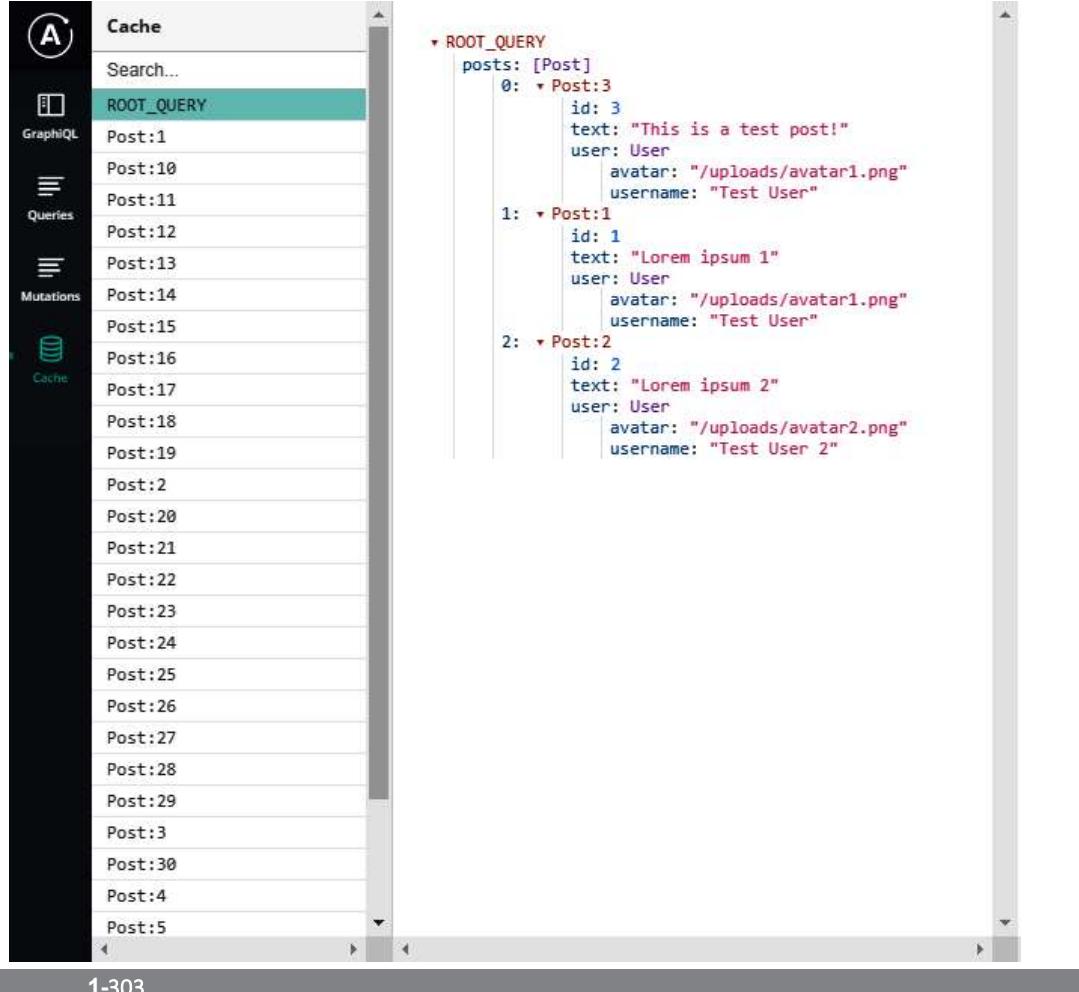


The screenshot shows the Apollo Client Developer Tools interface. On the left is a sidebar with icons for GraphiQL, Queries, Mutations, and Cache. The main area is titled "Watched queries" and shows one query listed. The query details are as follows:

- 1 <Query> [Run in GraphiQL](#)
- Variables**
- Query string**

```
{  
  posts {  
    id  
    text  
    user {  
      avatar  
      username  
    }  
  }  
}
```

- The last window is Cache.
- Here, you are able to see all of the data stored inside the Apollo cache:



```

Cache
Search...
ROOT_QUERY
Post:1
Post:10
Post:11
Post:12
Post:13
Post:14
Post:15
Post:16
Post:17
Post:18
Post:19
Post:2
Post:20
Post:21
Post:22
Post:23
Post:24
Post:25
Post:26
Post:27
Post:28
Post:29
Post:3
Post:30
Post:4
Post:5

▼ ROOT_QUERY
  posts: [Post]
    0: ▼ Post:3
      id: 3
      text: "This is a test post!"
      user: User
        avatar: "/uploads/avatar1.png"
        username: "Test User"
    1: ▼ Post:1
      id: 1
      text: "Lorem ipsum 1"
      user: User
        avatar: "/uploads/avatar1.png"
        username: "Test User"
    2: ▼ Post:2
      id: 2
      text: "Lorem ipsum 2"
      user: User
        avatar: "/uploads/avatar2.png"
        username: "Test User 2"
  
```

Summary

- In this lesson, you learned how to connect our GraphQL API to React.
- To do this, we used Apollo Client to manage the cache and the state of our components, and to update the React and the actual DOM of the browser.
- We looked at how to send queries and mutations against our server in two different ways.

"Complete Lab 4"

5: Reusable React Components



Reusable React Components

This lesson will cover everything you need to know in order to write efficient and reusable React components. It will cover the following topics:

- React patterns
- Structured React components
- Rendering nested components
- The React Context API
- The Apollo Consumer component

Introducing React patterns

We will go over the most commonly used patterns that React offers, as follows:

- Controlled components
- Stateless functions
- Conditional rendering
- Rendering children

Controlled components

- By definition, a component is uncontrolled whenever the value is not set by a property through React, but only saved and taken from the real browser DOM.
- The value of an input is then retrieved from a reference to the DOM Node, and is not managed and taken from React's component state.

Controlled components

- The following code shows the post form where the user will be able to submit new posts.
- I have excluded the rendering logic for the complete feed, as it is not a part of the pattern that I want to show you:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_1.txt

Stateless functions

- One fundamental and efficient solution for writing well-structured and reusable React components is the use of stateless functions.
- As you might expect, stateless functions are functions, not React components.
- They are not able to store any states; only properties can be used to pass and render data.

Stateless functions

- Beginning with the file structure, we will create a new folder for our new components (or stateless functions), as follows:

```
mkdir src/client/components
```

- Many parts of our application need to be reworked. Create a new file for our first stateless function, as follows:

```
touch src/client/components/loading.js
```

Stateless functions

- Currently, we display a dull and boring Loading... message when our GraphQL requests are running.
- Let's change this by inserting the following code into the loading.js file:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_2.txt

Stateless functions

- Lastly, we are returning a simple div tag with the CSS style and the bouncer class.
- What's missing here is the CSS styling.
- The code should look as follows; just add it to our style.css file:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_3.txt

Stateless functions

- First, import the new loading spinner to the top of your files, as follows:

```
import Loading from './components/loading';
```

- You can then render the stateless function like any normal component, as follows:

```
if (loading) return <Loading />;
```

- Generally, you can accomplish conditional rendering by using the curly brace syntax. An example of an if statement is as follows:

```
render() {  
  const { shouldRender } = this.state;  
  
  return (  
    <div className="conditional">  
      {(shouldRender === true) && (  
        <p>Successful conditional rendering!</p>  
      )}  
    </div>  
  )  
}
```

Rendering child components

- In all of the code that we have written so far, we have directly written the markup like it is rendered to real HTML.
- A great feature that React offers is the ability to pass children to other components.
- The parent component decides what is done with its children.

- Create an error.js file next to the loading.js file in the components folder, as follows:

```
import React, { Component } from 'react';
```

```
export default class Error extends Component {
  render() {
    const { children } = this.props;

    return (
      <div className="error message">
        {children}
      </div>
    );
  }
}
```

Rendering child components

- To start using the new Error component, you can simply import it.
- The markup for the new component is as follows:

```
if (error) return <Error><p>{error.message}</p></Error>;
```

Rendering child components

- Add some CSS, and everything should be finished, as shown in the following code snippet:

```
.message {  
  margin: 20px auto;  
  padding: 5px;  
  max-width: 400px;  
}  
.error.message {  
  border-radius: 5px;  
  background-color: #FFF7F5;  
  border: 1px solid #FF9566;  
  width: 100%;  
}
```

Rendering child components

- A working result might look as follows:

```
GraphQL error: connect ETIMEDOUT
```

Structuring our React application

- We have already improved some things by using React patterns.
- You should do some homework and introduce those patterns wherever possible.
- When writing applications, one key objective is to keep them modular and readable, but also as understandable as possible.

The React file structure

- We have already saved our Loading and Error components in the components folder.
- Still, there are many parts of our components that we did not save in separate files, to improve the readability of this course.
- I will explain the most important solution for unreadable React code in one example.

The React file structure

- Instead of creating a post.js file in our components folder, we should first create another post folder, as follows:

```
mkdir src/client/components/post
```

The React file structure

- Create a new header.js file in the components/post folder, as follows:

```
import React from 'react';
```

```
export default ({post}) =>
  <div className="header">
    <img src={post.user.avatar} />
    <div>
      <h2>{post.user.username}</h2>
    </div>
  </div>
```

The React file structure

- Up next is the post content, which represents the body of a post item.
- Add the following code inside of a new file, called content.js:

```
import React from 'react';
```

```
export default ({post}) =>
  <p className="content">
    {post.text}
  </p>
```

- The main file is a new index.js file in the new post folder. It should look as follows:

```
import React, { Component } from 'react';
import PostHeader from './header';
import PostContent from './content';

export default class Post extends Component {
  render() {
    const { post } = this.props;

    return (
      <div className={"post " + (post.id < 0 ? "optimistic": "")}>
        <PostHeader post={post}/>
        <PostContent post={post}/>
      </div>
    )
  }
}
```

The React file structure

- You can now use the new Post component in the feed list with ease.
- Just replace the old code inside the loop, as follows:

```
<Post key={post.id} post={post} />
```

Efficient Apollo React components

- We have successfully replaced the post items in our feed with a React component, instead of raw markup.
- A major part, which I dislike very much, is the Apollo Query component and Mutation component, and how we are using these at the moment directly inside the render method of our components.
- I will show you a quick workaround to make these components more readable.

The Apollo Query component

- Create a new queries folder inside of the components folder, as follows:

```
mkdir src/client/components/queries
```

The Apollo Query component

- The query that we want to remove from our view layer is the postsFeed query.
- You can define the naming conventions for this, but I would recommend using the RootQuery name as the filename, as long as it works.
- So, we should create a postsFeed.js file in the queries folder, and insert the following code:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_4.txt

The Apollo Query component

- Preparing our next component, we split the infinite scroll area into a second file.
- Place a feedlist.js into the components/posts folder, as follows:

Refer to the file 5_5.txt

The Apollo Query component

- Import the new components in the Feed.js, as follows:

```
import FeedList from './components/post/feedlist';
import PostsFeedQuery from
'./components/queries/postsFeed';
```

The Apollo Query component

- Replace the div tag with the feed class name and our two new components, as follows:

```
<PostsFeedQuery>
  <FeedList />
</PostsFeedQuery>
```

The Apollo Mutation component

- Create a new folder for all your mutations, as follows:
`mkdir src/client/components/mutations`
- Next, we want to outsource the mutation into a special file.
- To do so, create the `addPost.js` file, named after the GraphQL mutation itself, Insert the following code:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_6.txt

The Apollo Mutation component

- Going on, we should build a post form component that only handles the creation of new posts.
- Just call it form.js, and place it inside of the post's components folder.
- The code must look like the following snippet:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_7.txt

The Apollo Mutation component

- Lastly, we finalize the Feed.js main file. It should look as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_8.txt

Extending Graphbook

- Adding a drop-down menu to the posts, in order to allow for deleting or updating the content.
- Creating a global user object with the React Context API.
- Using Apollo Consumer as an alternative to the React Context API.
- Implementing a top bar as the first component rendered above all of the views. We can search for users in our database from a search bar, and we can show the logged-in user from the global user object.

The React context menu

We will layout the plan that we want to follow:

- Rendering a simple icon with FontAwesome
- Building React helper components
- Handling the onClick event and setting the correct component state
- Using the conditional rendering pattern to show the drop-down menu, if the component state is set correctly
- Adding buttons to the menu and binding mutations to them

The React context menu

- The following is a preview screenshot, showing how the final implemented feature should look:



FontAwesome in React

- As you may have noticed, we have not installed FontAwesome yet.
- Let's fix this with npm:

```
npm i --save @fortawesome/fontawesome-svg-core  
@fortawesome/free-solid-svg-icons  
@fortawesome/free-brands-svg-icons  
@fortawesome/react-fontawesome
```

FontAwesome in React

- Creating a separate file for FontAwesome will help us to have a clean import.
- Save the following code under the fontawesome.js file, inside of the components folder:

```
import { library } from '@fortawesome/fontawesome-svg-core';
import { faAngleDown } from '@fortawesome/free-solid-svg-icons';
library.add(faAngleDown);
```

FontAwesome in React

- The only place where we need this file is within our root App.js file.
- It ensures that all of our custom React components can display the imported icons.
- Add the following import statement to the top:

```
import './components/fontawesome';
```

React helper components

- Production-ready applications need to be polished as much as possible. Implementing reusable React components is one of the most important things to do.
- You should notice that drop-down menus are a common topic when building client-side applications.
- They are global parts of the front end and appear everywhere throughout our components.

React helper components

- Logically, the first step is to create a new folder to store all of the helper components, as follows:

`mkdir src/client/components/helpers`

- Create a new file, called dropdown.js, as the helper component:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_9.txt

- Replace the handleClick method and add the componentWillMount React method, as follows:

```
componentWillUnmount() {  
  document.removeEventListener('click', this.handleClick, true);  
}  
handleClick = () => {  
  const { show } = this.state;  
  
  this.setState({show: !show}, () => {  
    if(!show) {  
      document.addEventListener('click', this.handleClick);  
    } else {  
      document.removeEventListener('click', this.handleClick);  
    }  
  });  
}
```

The GraphQL updatePost mutation

- There is a new mutation that we need to insert into our schema, as follows:

```
updatePost (  
    post: PostInput!  
    postId: Int!  
) : Post
```

The GraphQL updatePost mutation

- Once it is inside of our schema, the implementation to execute the mutation will follow.
- Copy the following code over to the resolvers.js file, in the RootMutation field:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_10.txt

The GraphQL updatePost mutation

- Add the new updatePost mutation to the new file, as follows:

```
const UPDATE_POST = gql`  
  mutation updatePost($post : PostInput!, $postId : Int!) {  
    updatePost(post : $post, postId : $postId) {  
      id  
      text  
    }  
  }  
`;
```

The GraphQL updatePost mutation

- Create an `UpdatePostMutation` class, as follows:

```
export default class UpdatePostMutation extends  
Component {  
  state = {  
    postContent: this.props.post.text  
  }  
  changePostContent = (value) => {  
    this.setState({postContent: value})  
  }  
}
```

The GraphQL updatePost mutation

- A React component always needs a render method.
- This one is going to be a bit bigger:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_11.txt

The GraphQL updatePost mutation

- We will handle this within the Post component itself, because we want to edit the post in place, and do not want to open a modal or a specific Edit page.
- Go to your post's index.js file and exchange it with the new one, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_12.txt

The GraphQL updatePost mutation

- We already have a post form that we can reuse with some adjustments, as you can see in the following code snippet.
- To use our standard post submission form as an update form, we must make some small adjustments.
- Open and edit the form.js file, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_13.txt

- Go to your post header file. The header is a great place to insert the drop-down component, as follows:

```
import React from 'react';
import Dropdown from '../helpers/dropdown';
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome';
export default ({post, changeState}) =>
  <div className="header">
    <img src={post.user.avatar} />
    <div>
      <h2>{post.user.username}</h2>
    </div>
    <Dropdown trigger={<FontAwesomeIcon icon="angle-down" />}>
      <button onClick={changeState}>Edit</button>
    </Dropdown>
  </div>
```

The Apollo deletePost mutation

- Edit the GraphQL schema. The deletePost mutation needs to go inside of the RootMutation object.
- The new Response type serves as a return value, as deleted posts cannot be returned because they do not exist.
- Note that we only need the postId parameter, and do not send the complete post:

```
type Response {  
    success: Boolean  
}  
deletePost (  
    postId: Int!  
): Response
```

The Apollo deletePost mutation

- Add the missing GraphQL resolver function, The code is pretty much the same as from the update resolver, except that only a number is returned by the destroy method of Sequelize, not an array.
- It represents the number of deleted rows, We return an object with the success field.
- This field indicates whether our front end should throw an error:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_14.txt

The Apollo deletePost mutation

- Add the new deletePost mutation, as follows:

```
const DELETE_POST = gql`  
  mutation deletePost($postId : Int!) {  
    deletePost(postId : $postId) {  
      success  
    }  
  }  
`;
```

The Apollo deletePost mutation

- Lastly, insert the new component's code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_15.txt

The Apollo deletePost mutation

- Open the header.js file and import the following mutation:

```
import DeletePostMutation from './mutations/deletePost';
```

- Instead of directly adding the new button to our header, we will create another stateless function, as follows:

```
const DeleteButton = ({deletePost, postId}) =>
  <button onClick={() => {
    deletePost({ variables: { postId } })
  }}>
    Delete
  </button>
```

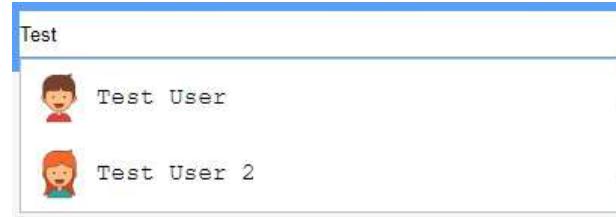
The Apollo deletePost mutation

- Insert both the mutation and the delete button into the header function, below the 'Edit' button, as follows:

```
<DeletePostMutation post={post}>
  <DeleteButton />
</DeletePostMutation>
```

The React application bar

- The first thing that we will implement is the simple search for users and the information about the logged-in user.
- We will begin with the search component, because it is really complex.
- The following screenshot shows a preview of what we are going to build:



The React application bar

- Edit the GraphQL schema and fill in the new RootQuery and type, as follows:

```
type UsersSearch {  
    users: [User]  
}
```

```
usersSearch(page: Int, limit: Int, text: String!):  
    UsersSearch
```

The React application bar

- Furthermore, the resolver function looks pretty much the same as the postsFeed resolver function.
- You can add the following code straight into the resolvers.js file, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_16.txt

The React application bar

- To enable this operator, we must import the `sequelize` package and extract the `Op` object from it, as follows:

```
import Sequelize from 'sequelize';
const Op = Sequelize.Op;
```

The React application bar

- Import all of the dependencies, and parse the new GraphQL query with the graphql-tag package, Note that we have three parameters.
- The text field is a required property for the variables that we send with our GraphQL request:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_17.txt

The React application bar

- Paste in the UsersSearchQuery class, as shown in the following code.
- In comparison to the PostsFeedQuery class, I have added the text property to the variables and handed over the refetch method to all subsequent children:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_18.txt

The React application bar

- Continuing with our plan, we will create the application bar in a separate file.
- Create a new folder, called bar, below the components folder and the index.js file.
- Fill it in with the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_19.txt

The React application bar

- The SearchBar class lives inside of a separate file. Just create a search.js file in the bar folder, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_20.txt

The React application bar

- Next, we will implement the SearchList. This behaves like the posts feed, but only renders something if a response is given with at least one user.
- The list is displayed as a drop-down menu and is hidden whenever the browser window is clicked on.
- Create a file called searchList.js inside of the bar folder, with the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_21.txt

The React Context API versus Apollo Consumer

- There are two ways to handle global variables in the stack that we are using at the moment.
- These are the new React Context API and the Apollo Consumer functionality.
- From version 16.3 of React, there is a new Context API that allows you to define global providers offering data through deeply nested components.

The React Context API versus Apollo Consumer

- Both of the approaches will result in the following output:



- The best option is to show you the two alternatives right away, so that you can identify your preferred method.

The React Context API

The following is a short explanation of this method:

- Context: This is a React approach for sharing data between components, without having to pass it through the complete tree.
- Provider: This is a global component, mostly used at just one point in your code. It enables you to access the specific context data.
- Consumer: This is a component that can be used at many different points in your application, reading the data behind the context that you are referring to.

The React Context API

- As always, we need to import all of the dependencies.
- Furthermore, we will set up a new empty context.
- The createContext function will return one provider and consumer to use throughout the application, as follows:

```
import React, { Component, createContext } from 'react';
const { Provider, Consumer } = createContext();
```

The React Context API

- Now, we want to use the provider, The best option here is to create a special UserProvider component.
- Later, when we have authentication, we can adjust it to do the GraphQL query, and then share the resultant data in our front end.
- For now, we will stick with fake data. Insert the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_22.txt

The React Context API

- We will set up a special `UserConsumer` component that takes care of passing the data to the underlying components by cloning them with React's `cloneElement` function:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_23.txt

The React Context API

- We need to introduce the provider at an early point in our code base.
- The best approach is to import the `UserProvider` in the `App.js` file, as follows:

```
import { UserProvider } from './components/context/user';
```

- Use the provider as follows, and wrap it around all essential components:

```
<UserProvider>
  <Bar />
  <Feed />
  <Chats />
</UserProvider>
```

The React Context API

- We could also have written the UserBar class as a stateless function, but we might need to extend this component in a later lesson.
- Insert the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_24.txt

The React Context API

- Open the index.js file for the top bar and add the following code to the render method, next to the SearchBar component:

```
<UserConsumer>
  <UserBar />
</UserConsumer>
```

The React Context API

- Obviously, you need to import both of the components at the top of the file, as follows:

```
import UserBar from './user';
import { UserConsumer } from '../context/user';
```

Apollo Consumer

- Nearly all of the code that we have written can stay as it was in the previous section.
- We just need to remove the `UserProvider` from the `App` class, because it is not needed anymore for the Apollo Consumer.
- Open up the `user.js` in the `context` folder and replace the contents with the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_25.txt

Documenting React applications

- We have put a lot of work and code into our React application.
- To be honest, we can improve upon our code base by documenting it.
- We did not comment on our code, we did not add React component property type definitions, and we have no automated documentation tool.

Setting up React Styleguidist

React Styleguidist and our application rely on webpack.
Just follow these instructions to get a working copy of it:

- Install React Styleguidist using npm, as follows:

```
npm install --save-dev react-styleguidist
```

- Usually, the folder structure is expected to be src/components, but we have a client folder between the src and components folder, So, we must configure React Styleguidist to let it understand our folder structure.
- Create a styleguide.config.js in the root folder of the project to configure it, as follows:

```
const path = require('path')
module.exports = {
  components: 'src/client/components/**/*.js',
  require: [
    path.join(__dirname, 'assets/css/style.css')
  ],
  webpackConfig: require('./webpack.client.config')
}
```

Setting up React Styleguidist

- Styleguidist provides two ways to view the documentation.
- One is to build the documentation statically, in production mode, with this command:

```
npx styleguidist build
```

Setting up React Styleguidist

- The second method, for development cases, lets Styleguidist run and create the documentation on the fly, using webpack:

```
npx styleguidist server
```

Graphbook Style Guide

Filter by name

- Error
- Fontawesome
- Dropdown
- Loading
- AddPostMutation
- DeletePostMutation
- UpdatePostMutation
- Content
- FeedList
- PostForm
- Header
- Post
- PostsFeedQuery

Error

[src/client/components/error.js](#) 

[Add examples to this component](#)



Fontawesome

[src/client/components/fontawesome.js](#) 

[Add examples to this component](#)



Dropdown

[src/client/components/helpers/dropdown.js](#) 

[Add examples to this component](#)



Loading

[src/client/components/loading.js](#) 

[Add examples to this component](#)



AddPostMutation

[src/client/components/mutations/addPost.js](#) 

[Add examples to this component](#)



React PropTypes

There are two React features that we did have covered yet, as follows:

- If your components have optional parameters, it can make sense to have default properties in the first place. To do this, you can specify `defaultProps` as a static property, in the same way as with the state initializers.
- The important part is the `propTypes` field, which you can fill for all of your components with the custom properties that they accept.

React PropTypes

- A new package is required to define the property types, as follows:

```
npm install --save prop-types
```

React PropTypes

- Now, open your Post component's index.js file.
- We need to import the new package at the top of the Post component's index.js file:

```
import PropTypes from 'prop-types';
```

- Next, we will add the new field to our component, above the state initializers:

```
static propTypes = {  
  /** Object containing the complete post. */  
  post: PropTypes.object.isRequired,  
}
```

React PropTypes

Post

src\client\components\post\index.js □

PROPS & METHODS

Prop name	Type	Default	Description
post	object	Required	Object containing the complete post.

- The best way to document a post object is to define which properties a post should include, at least for this specific component.
- Replace the property definition, as follows:

```
static propTypes = {
  /** Object containing the complete post. */
  post: PropTypes.shape({
    id: PropTypes.number.isRequired,
    text: PropTypes.string.isRequired,
    user: PropTypes.shape({
      avatar: PropTypes.string.isRequired,
      username: PropTypes.string.isRequired,
    }).isRequired
  }).isRequired,
}
```

React PropTypes

- The output from React Styleguidist now looks like the following screenshot:

Post

src\client\components\post\index.js 

PROPS & METHODS

Prop name	Type	Default	Description
post	shape	Required	Object containing the complete post.
			<code>id: number</code> — Required
			<code>text: string</code> — Required
			<code>user: shape</code> — Required

[Add examples to this component](#)

React PropTypes

- For our Post component, we need to create an index.md file, next to the index.js file in the post folder.
- React Styleguidist proposes creating either a Readme.md or Post.md file, but those did not work for me.
- The index.md file should look as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/5_26.txt

React PropTypes

- React Styleguidist automatically rerenders the documentation and generates the following output:

Post example:

The screenshot shows a component documentation page for a 'Post' component. At the top, there's a preview window displaying a user profile icon labeled 'Test User' and the text 'This is a test post!'. Below the preview is a 'VIEW CODE' button, followed by the component's code:

```
const post = {
  id: 3,
  text: "This is a test post!",
  user: {
    avatar: "/uploads/avatar1.png",
    username: "Test User"
  }
};

<Post key={post.id} post={post} />
```

Summary

Through this lesson, you have gained a lot of experience in writing a React application.

- You have applied multiple React patterns to different use cases, such as children passing through a pattern and conditional rendering.
- Furthermore, you now know how to document your code correctly.

"Complete Lab 5"

6: Authentication with Apollo and React



Authentication with Apollo and React

This lesson covers the following topics:

- What is a JWT?
- Cookies versus localStorage
- 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

JSON Web Tokens

- JSON Web Tokens (JWTs) are still a pretty new standard for carrying out authentication; not everyone knows about them, and even fewer people use them.
- This section does not provide a theoretical excursion through the mathematical or cryptographic basics of JWTs.
- In traditional web applications written in PHP, for example, you commonly have a session cookie.

localStorage versus cookie

- Let's take a look at another critical question, It is crucial to understand at least the basics of how authentication works and how it is secured.
- You are responsible for any faulty implementation that allows data breaches, so always keep this in mind, Where do we store the token we receive from the server?
- In whichever direction you send a token, you should always be sure that your communication is secure.
- For web applications like ours, be sure that HTTPS is enabled and used for all requests

localStorage versus cookie

- Now, let's take a look at cookies, These are great, despite their bad press due to the cookie compliance law initiated by the EU.
- Putting aside the more negative things that cookies can enable companies to do, such as tracking users, there are many good things about them.
- One significant difference compared to localStorage is that cookies are sent with every request, including the initial request for the site your application is hosted on.

Authentication with GraphQL

- Let's prepare and add a password and an email field.
- As we learned in lesson 3, Connecting to the Database, we create a migration to edit our user table.
- You can look up the commands in the third lesson if you have forgotten them:

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

Authentication with GraphQL

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_1.txt

Authentication with GraphQL

- All fields are simple strings. You can execute the migration, as stated in the lesson 3, Connecting to The Database, 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.zCjwtNFq9ukg9h4rhKiHCb6x5n
cKife',
email: 'test1@example.com',

Authentication with GraphQL

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

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

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
```

Apollo login mutation

- 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  
}
```

Apollo login mutation

- 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
```

Apollo login mutation

- The 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';
```

Apollo login mutation

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

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_2.txt

Apollo login mutation

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

```
export  
JWT_SECRET=awv4BclzsRysXkhoSAb8t8INENgXSqB  
ruVILwd45kGdYjeJHLap9LUJ1t9DTdw36DvLcWs3qEk  
PyCY6vOyNljlh2Er952h2gDzYwG82rs1qfTzdVlg89KTa  
Q4SWI1YGY
```

Apollo login mutation

- 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.

Apollo login mutation

- 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"  
  }  
}
```

Apollo login mutation

- This request should return a token:

```
{  
  "data": {  
    "login": {  
      "token":  
        "eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InRlc3Qx  
        QGV4YW1wbGUuY29tliwiaWQiOjEsImlhdCI6MTUzNzlwNjI0Mywi  
        ZXhwIjoxNTM3MjkyNjQzfQ.HV4dPIBzvU1yn6REMv42N0DS0Zdge  
        bFDX-Uj0MPHvIY"  
    }  
  }  
}
```

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 React login form

- The result for our form looks as follows:

Email

Password

Login

The React login form

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

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_3.txt

The React login form

- 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 React login form

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_4.txt

- 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>  
    )  
  }  
}
```

The React login form

- 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  
}
```

The React login form

- 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});  
  }  
}
```

The React login form

- 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/>  
}
```

The React login form

- We call this function `changeLoginState` and implement it inside the `App.js` file as follows:

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

The React login form

- 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}>
```

The React login form

- 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>  
  )  
}
```

The React login form

- 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);  
}
```

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
```

Apollo sign up mutation

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_5.txt

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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_6.txt

React sign up form

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_7.txt

React sign up form

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_8.txt

React sign up form

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_9.txt

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.

- 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);
};
```

Authenticating GraphQL requests

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_10.txt

- 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;
```

Authenticating GraphQL requests

- 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';
```

Authenticating GraphQL requests

- 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  
  },  
});
```

Authenticating GraphQL requests

- 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

Authenticating GraphQL requests

- 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
```

Authenticating GraphQL requests

- If you reload the page and manually set the loggedIn state variable to true via the React Developer Tools, you will see the following error message:

GraphQL error: You need to be logged in.

Authenticating GraphQL requests

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_11.txt

Authenticating GraphQL requests

- 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 });  
});  
}
```

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.

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:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_12.txt

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

CurrentUser GraphQL query

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

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

CurrentUser GraphQL query

- 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  
  }  
};`
```

CurrentUser GraphQL query

- 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>
```

- 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  
  }  
};`;
```

CurrentUser GraphQL query

- 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 });
  });
}}
```

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

Logging out using React

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_13.txt

Logging out using React

- 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}>
```

Logging out using React

- 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}/>
```

Logging out using React

- 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();  
}
```

Logging out using React

- 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);`

Logging out using React

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/6_14.txt

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.

"Complete Lab 6"



7: Handling Image Uploads



Handling Image Uploads

This lesson will cover the following topics:

- Setting up Amazon Web Services
- Configuring an AWS S3 bucket
- Accepting file uploads on the server
- Uploading images with React through Apollo
- Cropping images

Setting up Amazon Web Services

First, I have to mention that Amazon (or, to be specific, Amazon Web Services (AWS)) is not the only provider of hosting, storage, or computing systems. There are many such providers, including the following:

- Heroku
- Digital Ocean
- Google Cloud
- Microsoft Azure

AWS Services

Services ▾ Resource Groups ▾ 🔍

Sebastian Grebe ▾ Frankfurt ▾ Support ▾

AWS services

Find a service by name or feature (for example, EC2, S3 or VM, storage). 🔎

> Recently visited services

> All services

Build a solution

Get started with simple wizards and automated workflows.

Launch a virtual machine With EC2 ~2-3 minutes

Build a web app With Elastic Beanstalk ~6 minutes

Build using virtual servers With Lightsail ~1-2 minutes

Connect an IoT device With AWS IoT ~5 minutes

Start a development project With CodeStar ~5 minutes

Register a domain With Route 53 ~3 minutes

See more

Learn to build

Learn to deploy your solutions through step-by-step guides, labs, and videos.

See all ↗

Websites 3 videos, 3 tutorials, 3 labs

DevOps 6 videos, 2 tutorials, 3 labs

Backup and recovery 3 videos, 2 tutorials, 3 labs

Big data

Databases

Mobile

Helpful tips

Manage your costs Monitor your AWS costs, usage, and reservations using AWS Budgets. [Start now](#)

Create an organization Use AWS Organizations for policy-based management of multiple AWS accounts. [Start now](#)

Explore AWS

Machine Learning with Amazon SageMaker The fastest way to build, train, and deploy machine learning models. [Learn more](#) ↗

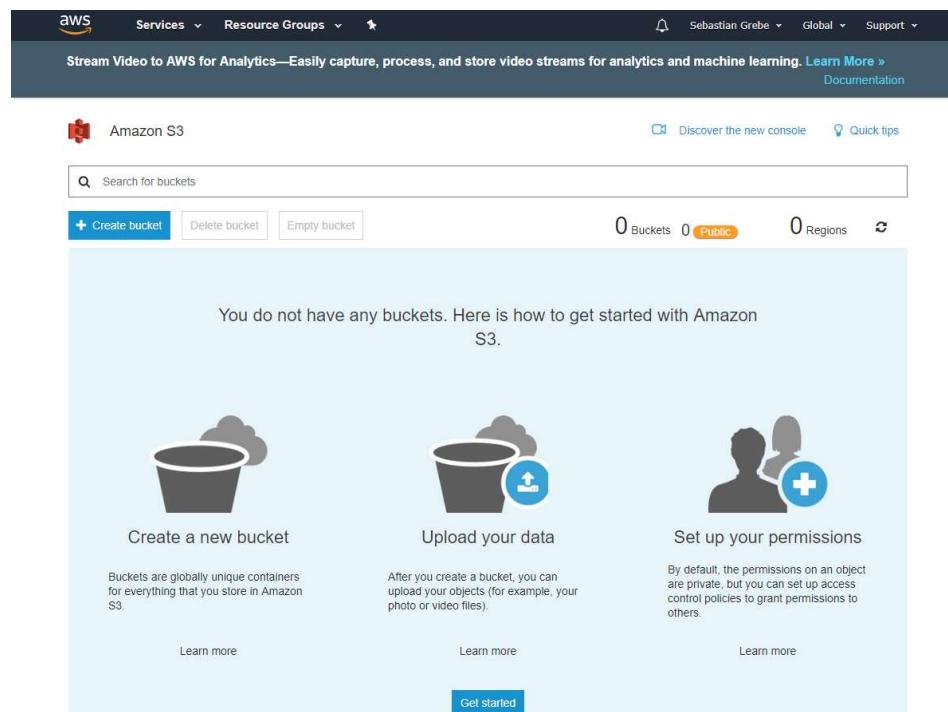
Amazon Relational Database Service (RDS) RDS manages and scales your database for you. RDS supports Aurora, MySQL, PostgreSQL, MariaDB, Oracle, and SQL Server. [Learn more](#) ↗

AWS Fargate Runs Containers for You AWS Fargate works with Amazon ECS to run and scale your containers for you. Pay only for the compute resources you need, scale quickly, and run any size application. [Learn more](#) ↗

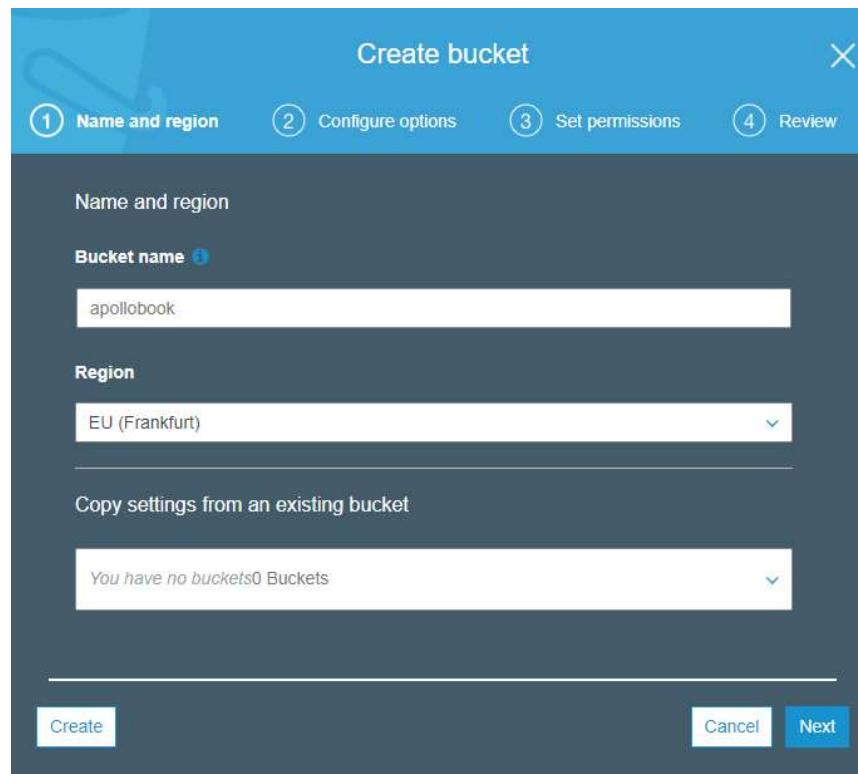
AWS Marketplace Find, buy, and deploy popular software products that run on AWS. [Learn more](#) ↗

LEARNING VOYAGE

Creating an AWS S3 bucket

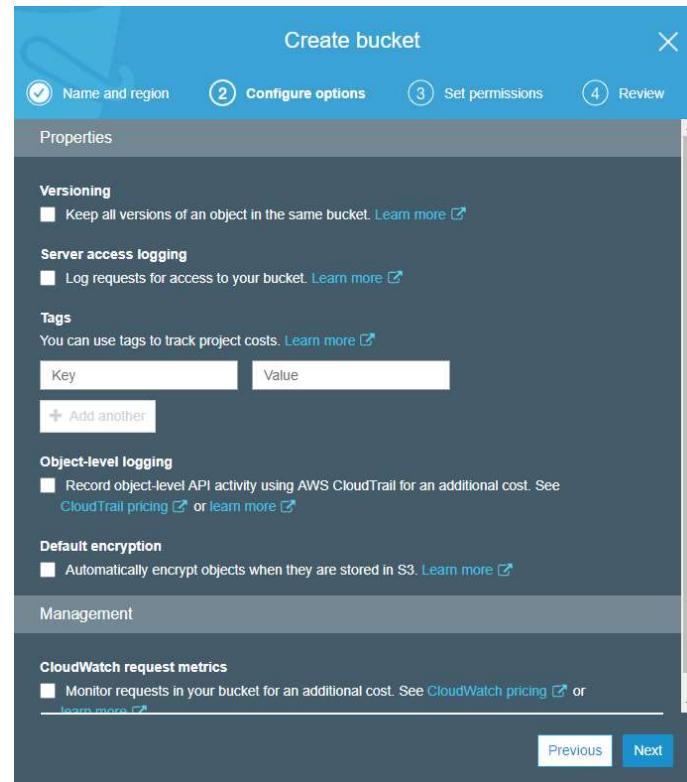


Creating an AWS S3 bucket



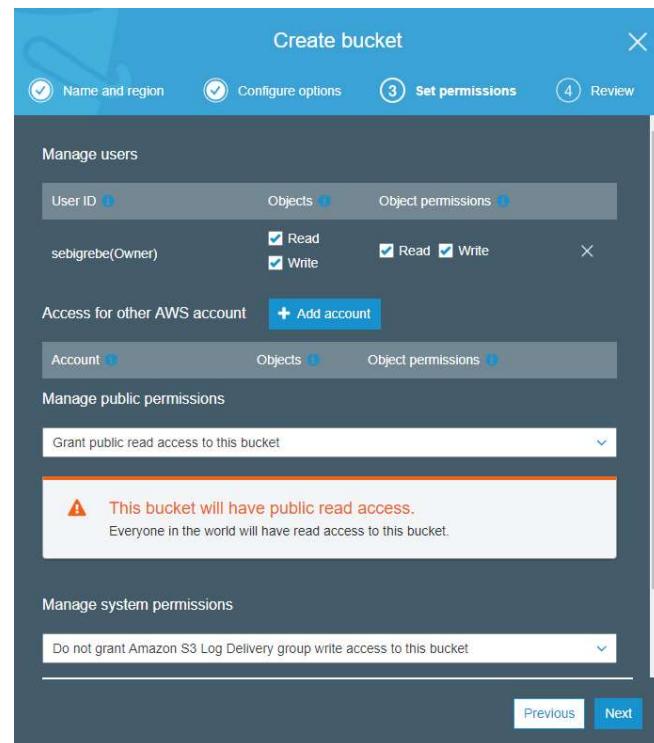
Creating an AWS S3 bucket

- Once you have picked a region, continue by clicking on Next.
- You will be confronted with a lot of new options:



Creating an AWS S3 bucket

- Under Manage public permissions, you have to select Grant public read access to this bucket to enable public access to all files saved in your S3 bucket.
- Take a look at the following screenshot to ensure that everything is correct:



Generating AWS access keys

- Before implementing the upload feature, we must create an AWS API key to authorize our back end at AWS, in order to upload new files to the S3 bucket.
- Click on your username in the top bar of AWS.
- There, you find a tab called My Security Credentials, which navigates to a screen offering various options to secure access to your AWS account.

Generating AWS access keys



You are accessing the security credentials page for your AWS account. The account credentials provide unlimited access to your AWS resources.

To help secure your account, follow an [AWS best practice](#) by creating and using AWS Identity and Access Management (IAM) users with limited permissions.

[Continue to Security Credentials](#)

[Get Started with IAM Users](#)

Don't show me this message again

Generating AWS access keys

Your Security Credentials

Use this page to manage the credentials for your AWS account. To manage credentials for AWS Identity and Access Management (IAM) users, use the [IAM Console](#).

To learn more about the types of AWS credentials and how they're used, see [AWS Security Credentials](#) in AWS General Reference.

- ▲ Password
- ▲ Multi-factor authentication (MFA)
- ▼ Access keys (access key ID and secret access key)

You use access keys to sign programmatic requests to AWS services. To learn how to sign requests using your access keys, see the [signing documentation](#). For your protection, store your access keys securely and do not share them. In addition, AWS recommends that you rotate your access keys every 90 days.

Note: You can have a maximum of two access keys (active or inactive) at a time.

Created	Deleted	Access Key ID	Last Used	Last Used Region	Last Used Service	Status	Actions
Oct 15th 2018		[REDACTED]	N/A	N/A	N/A	Active	Make Inactive Delete
Oct 15th 2018		[REDACTED]	2018-12-11 22:22 UTC+0100	eu-central-1	ecr	Active	Make Inactive Delete
Oct 15th 2018	Oct 15th 2018	[REDACTED]	N/A	N/A	N/A	Deleted	
Oct 15th 2018	Oct 15th 2018	[REDACTED]	N/A	N/A	N/A	Deleted	

[Create New Access Key](#)



Important Change - Managing Your AWS Secret Access Keys

As described in a [previous announcement](#), you cannot retrieve the existing secret access keys for your AWS root account, though you can still create a new root access key at any time. As a [best practice](#), we recommend [creating an IAM user](#) that has access keys rather than relying on root access keys.

- ▲ CloudFront key pairs
- ▲ X.509 certificate
- ▲ Account identifiers

Generating AWS access keys

Create Access Key ×

✓ Your access key (access key ID and secret access key) has been created successfully.

Download your key file now, which contains your new access key ID and secret access key. If you do not download the key file now, you will not be able to retrieve your secret access key again.

To help protect your security, store your secret access key securely and do not share it.

▼ Hide Access Key

Access Key ID: AKIAIY3YUAH5G6UPBEDA
Secret Access Key: /AOaMAXp1Hlmc3I1o1QiEgrgs2o4o62MTw2bSAAF

[Download Key File](#) [Close](#)

Uploading images to Amazon S3

- Implementing file uploads and storing files is always a huge task, especially for image uploads in which the user may want to edit his files again.
- For our front end, the user should be able to drag and drop his image into a dropzone, crop the image, and then submit it when he is finished.
- The back end needs to accept file uploads in general, which is not easy at all.

GraphQL image upload mutation

- When uploading images to S3, it is required to use an API key, which we have already generated.
- Because of this, we cannot directly upload the files from the client to S3 with the API key.
- Anyone accessing our application could read out the API key from the JavaScript code and access our bucket without us knowing.

GraphQL image upload mutation

- Interact with AWS to install the official npm package.
- It provides everything that's needed to use any AWS feature, not just S3:

`npm install --save aws-sdk`

- The next thing to do is edit the GraphQL schema and add a scalar Upload to the top of it.
- The scalar is used to resolve details such as the MIME type and encoding when uploading files:

`scalar Upload`

GraphQL image upload mutation

- Add the File type to the schema.
- This type returns the filename and the resulting URL under which the image can be accessed in the browser:

```
type File {  
  filename: String!  
  url: String!  
}
```

GraphQL image upload mutation

- Create the new `uploadAvatar` mutation.
- It is required that the user is logged in to upload avatar images, so append the `@auth` directive to the mutation.
- The mutation takes the previously mentioned `Upload` scalar as input:

```
uploadAvatar (  
    file: Upload!  
): File @auth
```

GraphQL image upload mutation

- Next, we will implement the mutation's resolver function in the resolvers.js file.
- For this, we will import and set up our dependencies at the top of the resolvers.js file, as follows:

```
import aws from 'aws-sdk';
const s3 = new aws.S3({
  signatureVersion: 'v4',
  region: 'eu-central-1',
});
```

GraphQL image upload mutation

- Inside the mutation property, insert the uploadAvatar function, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/7_1.txt

GraphQL image upload mutation

- The last step is to set the new avatar picture on the user in our database.
- We execute the User.update model function from Sequelize by setting the new URL from response.Location, which S3 gave us after we resolved the promise.
- An example link to an S3 image is as follows:
<https://apollobook.s3.eu-central-1.amazonaws.com/1/test.png>

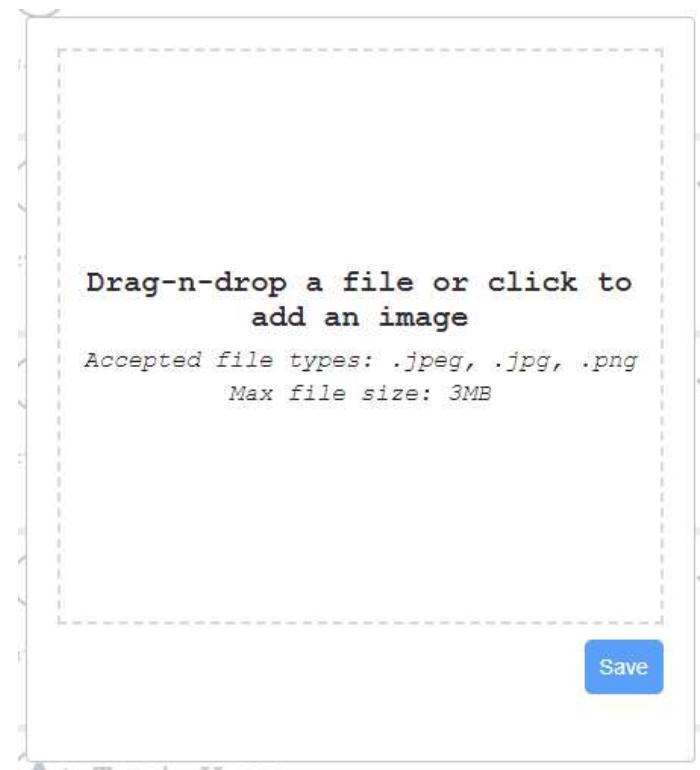
GraphQL image upload mutation

- In the previous section, we generated the access tokens, in order to authorize our back end at AWS.
- By default, the AWS SDK expects both tokens to be available in our environment variables.
- Like we did before with the `JWT_SECRET`, we will set the tokens as follows:

```
export AWS_ACCESS_KEY_ID=YOUR_AWS_KEY_ID  
export AWS_SECRET_ACCESS_KEY=YOUR_AWS_SECRET_KEY
```

React image cropping and uploading

- The result that we are targeting looks like the following screenshot:



React image cropping and uploading

- To get the image upload working, we will install two new packages.
- To do this, you can follow these instructions:
 - Install the packages with npm:
`npm install --save react-modal @synapsestudios/react-drop-n-crop`

React image cropping and uploading

- When using the react-drop-n-crop package, we can rely on its included CSS package.
- In your main App.js, import it straight from the package itself, as follows:

```
import '@synapsestudios/react-drop-n-crop/lib/react-drop-n-crop.min.css';
```

React image cropping and uploading

- The next package that we will install is an extension for the Apollo Client, which will enable us to upload files, as follows:

```
npm install --save apollo-upload-client
```

React image cropping and uploading

- To get the apollo-upload-client package running, we have to edit the index.js from the apollo folder where we initialize the Apollo Client and all of its links.
- Import the createUploadLink function at the top of the index.js file, as follows:

```
import { createUploadLink } from 'apollo-upload-client';
```

React image cropping and uploading

- You must replace the old `HttpLink` at the bottom of the link array with the new upload link.
- Instead of having a new `HttpLink`, we will now pass the `createUploadLink`, but with the same parameters.
- When executing it, a regular link is returned.
- The link should look like the following code:

```
createUploadLink({  
  uri: 'http://localhost:8000/graphql',  
  credentials: 'same-origin',  
}),
```

React image cropping and uploading

- Now that the packages are prepared, we can start to implement our uploadAvatar mutation component for the client.
- Create a new file, called uploadAvatar.js, in the mutations folder.
- At the top of the file, import all dependencies and parse all GraphQL requests with graphql-tag in the conventional way, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/7_2.txt

React image cropping and uploading

- Next, you can copy the UploadAvatarMutation class.
- It passes the uploadAvatar mutation function to the underlying children, and sets the newly uploaded avatar image inside of the cache for the currentUser query.
- It shows the new user avatar directly in the top bar when the request is successful:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/7_3.txt

React image cropping and uploading

Execute the following steps to get this logic running:

- It is always good to make your components as reusable as possible, so create an `avatarModal.js` file inside of the `components` folder.
- As always, you will have to import the two new `react-modal` and `react-drop-n-crop` packages first, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/7_4.txt

React image cropping and uploading

- Generally, this is not a problem, but our GraphQL API expects that we sent a real file, not just a string, as we explained previously.
- Consequently, we have to convert the data URI to a blob that we can send with our GraphQL request.
- Add the following function to take care of the conversion:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/7_5.txt

React image cropping and uploading

- The Modal component takes an onRequestClose method, which executes the showModal function when the user tries to close the modal (by clicking outside of it, for example).
- We receive the showModal function from the parent component, which we are going to cover in the next step.
- It tells the package to start with an empty dropzone. Switching between file selection and cropping is handled by the package on its own:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/7_6.txt

React image cropping and uploading

- Now, switch over to the user.js file in the bar folder, where all of the other application bar-related files are stored.
- Import the mutation and the new AvatarUpload component that we wrote before, as follows:

```
import UploadAvatarMutation from '../mutations/uploadAvatar';
import AvatarUpload from '../avatarModal';
```

React image cropping and uploading

- We introduce an `isOpen` state variable and catch the `onClick` event on the avatar of the user.
- Copy the following code to the `UserBar` class:

```
state = {  
  isOpen: false,  
}  
showModal = () => {  
  this.setState({ isOpen: !this.state.isOpen });  
}
```

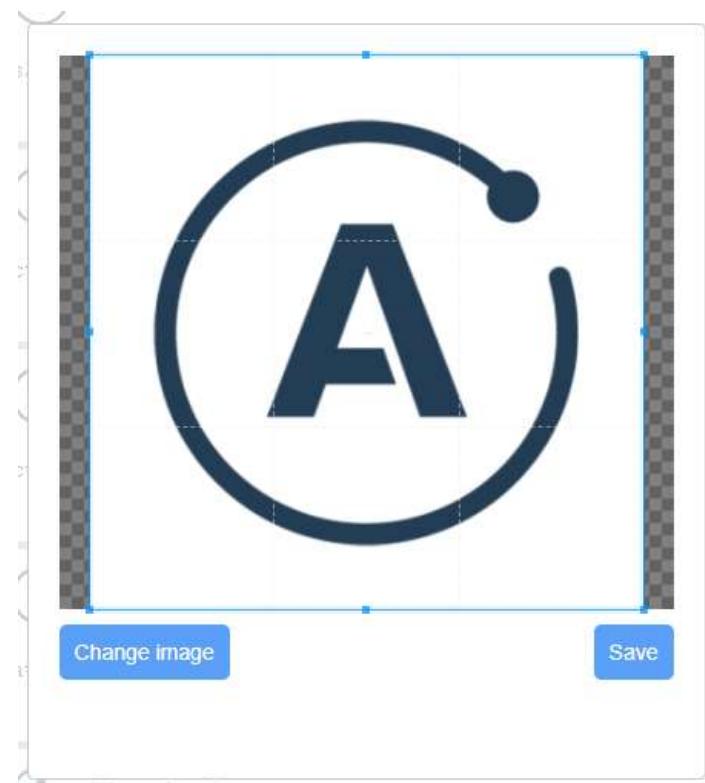
React image cropping and uploading

- Replace the return value of the render method with the following code:

```
return (
  <div className="user">
    <img src={user.avatar} onClick={this.showModal}>
    <UploadAvatarMutation>
      <AvatarUpload isOpen={this.state.isOpen}>
        showModal={this.showModal}>
      </AvatarUpload>
    </UploadAvatarMutation>
    <span>{user.username}</span>
  </div>
);
```

React image cropping and uploading

- Start the server and client with the matching npm run commands.
- Reload your browser and try out the new feature.
- You can see an example of this in the following screenshot:



Summary

- In this lesson, we started by creating an AWS account and an S3 bucket for uploading static images from our back end.
- Modern social networks consist of many images, videos, and other types of files.
- We introduced the Apollo Client, which allows us to upload any type of file.

"Complete Lab 7"

8: Routing in React



Routing in React

We will introduce client-side routing for our React application.

This lesson will cover the following topics:

- Installing React Router
- Implementing routes
- Securing routes
- Manual navigation

Setting up React Router

- Routing is essential to most web applications.
- You cannot cover all of the features of your application in just one page.
- It would be overloaded, and your user would find it difficult to understand.
- Sharing links to pictures, profiles, or posts is also very important for a social network such as Graphbook.
- One advantageous feature, for example, is being able to send links to specific profiles.

Installing React Router

- In the past, there were a lot of Reacts React Router, with various implementations and features.
- To install React Router, simply run npm again, as follows:

```
npm install --save react-router-dom
```

Installing React Router

- To get the routing working out of the box, we will add two parameters to the webpack.client.config.js file.
- The devServer field should look as follows:

```
devServer: {  
  port: 3000,  
  open: true,  
  historyApiFallback: true,  
},
```

Installing React Router

- The output field at the top of the config file must have a publicPath property, as follows:

```
output: {  
  path: path.join(__dirname, buildDirectory),  
  filename: 'bundle.js',  
  publicPath: '/',  
},
```

Implementing your first route

- Before implementing the routing, we will clean up the App.js file.
- Create a Main.js file next to the App.js file in the client folder.
- Insert the following code:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/8_1.txt

- Now, open and replace the render method of the App component to reflect those changes, as follows:

```
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}>
        </div>
    )
}
```

Installing React Router

- add the following line to the dependencies of our App.js file:

```
import Router from './router';
```

Installing React Router

- To do this, create a new router.js file in the client folder, next to the App.js file, with the following content:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/8_2.txt

Secured routes

- Secured routes represent a to specific paths that are only the is authenticated, or has the correct authorization.
- Insert the following code into the router.js file:

```
const PrivateRoute = ({ component: Component, ...rest }) => (
  <Route {...rest} render={(props) => (
    rest.loggedIn === true
      ? <Component {...props} />
      : <Redirect to={{
        pathname: '/',
      }} />
  )} />
)
```

Secured routes

- Use the new PrivateRoute component in the render method of the Router and replace the old Route, as follows:

```
<PrivateRoute path="/app" component={() => <Main  
changeLoginState=  
{this.props.changeLoginState}>}  
loggedIn={this.props.loggedIn}>
```

Secured routes

- Add the new LoginRoute component to the router.js file, as follows:

```
const LoginRoute = ({ component: Component, ...rest }) => (
  <Route {...rest} render={({props}) => (
    rest.loggedIn === false
      ? <Component {...props} />
      : <Redirect to={{
        pathname: '/app',
      }} />
    )} />
)
```

Secured routes

- Add the new path to the router, as follows:

```
<LoginRoute exact path="/" component={() =>
  <LoginRegisterForm
    changeLoginState={this.props.changeLoginState}/>
    loggedIn={this.props.loggedIn}/>
```

Secured routes

- The following table shows a quick example, taken from the official React Router documentation:

Router path	Browser path	exact	matches
/one	/one/two	true	no
/one	/one/two	false	yes

Catch-all routes in React Router

- Add the following code to the router.js file:

```
class NotFound extends Component {  
  render() {  
    return (  
      <Redirect to="/" />  
    );  
  }  
}
```

Catch-all routes in React Router

- The NotFound component is minimal.
- It just redirects the user to the root path.
- Add the next Route component to the Switch in the Router.
- Ensure that it is the last one on the list:

```
<Route component={NotFound} />
```

Advanced routing with React Router

- The primary goal of this lesson is to build a profile page for your users.
- We need a separate page to show all of the content that a single user has entered or created.
- The content would not fit next to the posts feed.
- When looking at Facebook, we can see that every user has their own address, under which we can find the profile page of a specific user.

Parameters in routes

- We have prepared most of the work required to add a new user route.
- Open up the router.js file again.
- Add the new route, as follows:

```
<PrivateRoute path="/user/:username"
component={props => <User {...props}
changeLoginState={this.props.changeLoginState}/>}
loggedIn={this.props.loggedIn}/>
```

Parameters in routes

- Before implementing it, we import the dependency at the top of router.js to get the preceding route working:

```
import User from './User';
```

- The User.js file should look as follows:

```
import React, { Component } from 'react';
import UserProfile from './components/user';
import Chats from './Chats';
import Bar from './components/bar';
import CurrentUserQuery from './components/queries/currentUser';
export default class User extends Component {
  render() {
    return (
      <CurrentUserQuery>
        <Bar changeLoginState={this.props.changeLoginState}/>
        <UserProfile username={this.props.match.params.username}>
        <Chats />
      </CurrentUserQuery>
    );
  }
}
```

Parameters in routes

- Follow these steps to build the user's profile page:
- Create a new folder, called user, inside the components folder.
- Create a new file, called index.js, inside the user folder.
- Import the dependencies at the top of the file, as follows:

```
import React, { Component } from 'react';
import PostsQuery from '../queries/postsFeed';
import FeedList from '../post/feedlist';
import UserHeader from './header';
import UserQuery from '../queries/userQuery';
```

Parameters in routes

- Insert the code for the UserProfile component that we are building at the moment beneath the dependencies, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/8_3.txt

Parameters in routes

- We should now edit and create the Apollo queries, before programming the profile header component.
- Open the postsFeed.js file from the queries folder.
- Change the first two lines to match the following code:

```
query postsFeed($page: Int, $limit: Int, $username: String) {  
  postsFeed(page: $page, limit: $limit, username: $username) {
```

Parameters in routes

- Add a new line to the getVariables method, above the return statement:

```
if(typeof variables.username !== typeof undefined) {  
    query_variables.username = variables.username;  
}
```

Parameters in routes

- Create a new userQuery.js file in the queries folder to create the missing query class.
- Import all of the dependencies and parse the new query schema with graphl-tag, as follows:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/8_4.txt

Parameters in routes

- The component itself is as simple as the ones that we created before.
- Insert the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/8_5.txt

Parameters in routes

- The last step is to implement the `UserProfileHeader` component.
- This component renders the `user` property, with all its values. It is just simple HTML markup.
- Copy the following code into the `header.js` file, in the `user` folder:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/8_6.txt

Querying the user profile

- With the new profile page, we have to update our back end accordingly.
- Let's take a look at what needs to be done, as follows:
- We have to add the username parameter to the schema of the postsFeed query and adjust the resolver function.
- We have to create the schema and the resolver function for the new UserQuery component.

Querying the user profile

- We will begin with the postsFeed query.
- Edit the postsFeed query in the RootQuery type of the schema.js file to match the following code:
`postsFeed(page: Int, limit: Int, username: String): PostFeed @auth`
- Here, I have added the username as an optional parameter.

Querying the user profile

- Now, head over to the resolvers.js file, and take a look at the corresponding resolver function.
- Replace the signature of the function to extract the username from the variables, as follows:

```
postsFeed(root, { page, limit, username }, context) {
```

Querying the user profile

- To make use of the new parameter, add the following lines of code above the return statement:

```
if(typeof username !== typeof undefined) {  
    query.include = [{model: User}];  
    query.where = { '$User.username$': username };  
}
```

Querying the user profile

- Let's move on and implement the new user query.
- Add the following line to the RootQuery in your GraphQL schema:

`user(username: String!): User @auth`

Querying the user profile

- In the resolvers.js file, we will now implement the resolver function using Sequelize:

```
user(root, { username }, context) {
  return User.findOne({
    where: {
      username: username
    }
  });
},
},
```

Querying the user profile

- Add the email as a valid field on the User type in your GraphQL schema with the following line of code:

email: String

Programmatic navigation in React Router

- We are going to extend the news feed to handle clicks on the username or the avatar image, in order to navigate to the user's profile page.
- Open the header.js file in the post components folder.
- Import the Link component provided by React Router, as follows:

```
import { Link } from 'react-router-dom';
```

Programmatic navigation in React Router

- To test this, wrap the username and the avatar image in the Link component, as follows:

```
<Link to={'/user/'+post.user.username}>
  <img src={post.user.avatar} />
  <div>
    <h2>{post.user.username}</h2>
  </div>
</Link>
```

Programmatic navigation in React Router

- You have to copy one new CSS rule, because the Link component has changed the markup:

```
.post .header a > * {  
  display: inline-block;  
  vertical-align: middle;  
}
```

- First of all, we will create a new home.js file in the bar folder, and we will enter the following code:

```
import React, { Component } from 'react';
import { withRouter } from 'react-router';
class Home extends Component {
  goHome = () => {
    this.props.history.push('/app');
  }
  render() {
    return (
      <button className="goHome" onClick={this.goHome}>Home</button>
    );
  }
}
export default withRouter(Home);
```

Programmatic navigation in React Router

- There are a few things to do in order to get the button working, as follows:
- Import the component into the index.js file of the bar folder, as follows:

```
import Home from './home';
```

Programmatic navigation in React Router

- Then, replace the Logout button with the following lines of code:

```
<div className="buttons">
  <Home/>
  <Logout
    changeLoginState={this.props.changeLoginState}>
</div>
```

- You can replace the old CSS for the logout button and add the following:

```
.topbar .buttons {  
    position: absolute;  
    right: 5px;  
    top: 5px;  
    height: calc(100% - 10px);  
}  
.topbar .buttons > * {  
    height: 100%;  
    margin-right: 5px;  
    border: none;  
    border-radius: 5px;  
}
```

Programmatic navigation in React Router



Remembering the redirect location

- In the PrivateRoute component, swap out the Redirect with the following code:

```
<Redirect to={{  
  pathname: '/',  
  state: { from: props.location }  
}} />
```

Remembering the redirect location

- We want to use this variable when the user is logging in.
- Replace the Redirect component in the LoginRoute component with the following lines:

```
<Redirect to={{
  pathname: (typeof props.location.state !== typeof
undefined) ?
  props.location.state.from.pathname : '/app',
}} />
```

Summary

- In this lesson, we transitioned from our one-screen application to a multi-page setup.
- React Router, our main library for routing purposes, now has three paths, under which we display different parts of Graphbook.
- Furthermore, we now have a catch-all route, in which we can redirect the user to a valid page.

"Complete Lab 8"

9: Implementing Server-Side Rendering



Implementing Server-Side Rendering

This lesson 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

Introduction to server-side rendering

- First, you have to understand the differences between using a server-side rendered and a client-side rendered application.
- There are numerous things to bear in mind when transforming a pure client rendered application to support server-side rendering.
- The current user flow begins with requesting a standard index.html.

SSR in Express.js

- The first step is to implement basic server-side rendering on the back end.
- 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
```

SSR in Express.js

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

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

SSR in Express.js

- Follow these steps to get your development environment ready for SSR:
- The first step is to import the two new webpack modules: webpack-dev-middleware and webpack-hot-middleware.
- Put the following code underneath the setup for the Express.js helmet, in order to only use the new packages in development:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/9_1.txt

SSR in Express.js

- 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.
- 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.

SSR in Express.js

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/9_2.txt

- 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>
    );
  }
}
```

SSR in Express.js

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/9_3.txt

SSR in Express.js

- Open the router.js inside the client folder and follow these steps:
- Delete the import statement for the react-router-dom package.
- Insert the following code to import the package properly:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/9_4.txt

SSR in Express.js

- 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}>
```

SSR in Express.js

- 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();
});
```

SSR in Express.js

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/9_5.txt

SSR in Express.js

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

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

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

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

SSR in Express.js

- 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/';
```

SSR in Express.js

- 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}>);
```

SSR in Express.js

- 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';
```

SSR in Express.js

- 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);
```

SSR in Express.js

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/9_6.txt

SSR in Express.js

- 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)}`);
```

SSR in Express.js

- 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();
```

SSR in Express.js

- Pass this head variable to the template function as a second parameter, as follows:

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

SSR in Express.js

- 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()}
```

SSR in Express.js

- 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:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/9_7.txt

Authentication with SSR

- You should have noticed that we have removed most of the authentication logic from the server-side React code.
- The authentication has to be transitioned to cookies, which are sent with every request.

Authentication with SSR

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

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

Authentication with SSR

- 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;
```

Authentication with SSR

- 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();  
  }  
);
```

Authentication with SSR

- 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;
}
```

Authentication with SSR

- Pass the loggedIn variable to the Graphbook component, as follows:

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

Authentication with SSR

- Now, we can access the loggedIn property inside index.js from the ssr folder.
- Extract the loggedIn state 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}/>
```

Authentication with SSR

- 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  
  }  
}
```

Authentication with SSR

- Insert the following code directly above the return statement:

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

Authentication with SSR

- 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  
)
```

Authentication with SSR

- Create a new logout.js inside the mutations folder, in order to create the new LogoutMutation class.
- The content should look as follows:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/9_8.txt

Authentication with SSR

- 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';`

Authentication with SSR

- 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>`

Authentication with SSR

- 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();
  });
}
```

Authentication with SSR

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

logout: Response @auth

Authentication with SSR

- 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
  };
},
```

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.

Running Apollo queries with SSR

- 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);`

- 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)
};
```

Running Apollo queries with SSR

- 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';
```

Running Apollo queries with SSR

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/9_9.txt

Running Apollo queries with SSR

- 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();
```

Running Apollo queries with SSR

- 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)}');`
- Inside the template.js file, extend the function declaration and append the state variable as a third parameter, after the head variable.

Running Apollo queries with SSR

- 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:

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

Running Apollo queries with SSR

- 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__)`

Running Apollo queries with SSR

- 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)
```

Summary

- In this lesson, 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.

"Complete Lab 9"

10: Real-Time Subscriptions



Real-Time Subscriptions

This lesson covers the following topics:

- Using GraphQL with WebSockets
- Implementing Apollo Subscriptions
- JWT authentication with Subscriptions

GraphQL and WebSockets

- The problem with regular HTTP connections, however, is that they are one-time requests.
- They can only respond with the data that exists at the time of the request.
- If the database receives a change concerning the posts or the chats, the user won't know about this until they execute another request.
- The user interface shows outdated data in this case.
- To solve this issue, you can refetch all requests in a specific interval, but this is a bad solution because there's no time range that makes polling efficient.

Apollo Subscriptions

- The first step is to install all the required packages to get GraphQL subscriptions working.
- Install them using npm:

```
npm install --save graphql-subscriptions subscriptions-  
transport-ws apollo-link-ws
```

Subscriptions on the Apollo Server

- Open the index.js file of the server.
- Import a new Node.js interface at the top of the file:

```
import { createServer } from 'http';
```

Subscriptions on the Apollo Server

- Add the following line of code beneath the initialization of Express.js inside the app variable:

```
const server = createServer(app);
```

Subscriptions on the Apollo Server

- To get our server listening again, edit the initialization routine of the services.
- The for loop should now look as follows:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_1.txt

Subscriptions on the Apollo Server

- To add the new service into the preceding `serviceNames` object, edit the `index.js` services file with the following content:

```
import graphql from './graphql';
import subscriptions from './subscriptions';
export default utils => ({
  graphql: graphql(utils),
  subscriptions: subscriptions(utils),
});
```

Subscriptions on the Apollo Server

- If you have created the new subscriptions index.js file, import all the dependencies at the top of the file:

```
import { makeExecutableSchema } from 'graphql-tools';
import Resolvers from'..../graphql/resolvers';
import Schema from'..../graphql/schema';
import auth from '..../graphql/auth';
import jwt from 'jsonwebtoken';
const { JWT_SECRET } = process.env;
import { SubscriptionServer } from 'subscriptions-transport-ws';
import { execute, subscribe } from 'graphql';
```

Subscriptions on the Apollo Server

- We begin the implementation of the new service by exporting a function with the `export default` statement and creating the `executableSchema`:

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

Subscriptions on the Apollo Server

- Insert the following code under executableSchema:

```
new SubscriptionServer({  
  execute,  
  subscribe,  
  schema: executableSchema,  
},  
{  
  server,  
  path: '/subscriptions',  
});
```

Subscriptions on the Apollo Server

- Open the schema.js file to define our first subscription.
- Add a new type called RootSubscription next to the RootQuery and RootMutation types, including the new subscription, called messageAdded:

```
type RootSubscription {  
  messageAdded: Message  
}
```

Subscriptions on the Apollo Server

- Change the schema as follows:

```
schema {  
  query: RootQuery  
  mutation: RootMutation  
  subscription: RootSubscription  
}
```

Subscriptions on the Apollo Server

- Open the resolvers.js file and perform the following steps:
- Import all dependencies that allow us to set up our GraphQL API with a PubSub system:

```
import { PubSub, withFilter } from 'graphql-subscriptions';
const pubsub = new PubSub();
```

Subscriptions on the Apollo Server

- Add it to the resolvers:

```
RootSubscription: {  
  messageAdded: {  
    subscribe: () =>  
      pubsub.asyncIterator(['messageAdded']),  
  },  
},
```

Subscriptions on the Apollo Server

- When subscribing to the messageAdded subscription, there needs to be another method that publicizes the newly created message to all clients.
- The best location is the addMessage mutation where the new message is created.
- Replace the addMessage resolver function with the following code:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_2.txt

Subscriptions on the Apollo Client

- To configure the Apollo Client correctly, follow these steps:
- Open the index.js file from the apollo folder.
- Import the following dependencies:

```
import { WebSocketLink } from 'apollo-link-ws';
import { SubscriptionClient } from 'subscriptions-transport-
ws';
import { getMainDefinition } from 'apollo-utilities';
import { ApolloLink, split } from 'apollo-link';
```

Subscriptions on the Apollo Client

- We are going to create both links for the split function.
- Detect the protocol and port where we send all GraphQL subscriptions and requests.
- Add the following code beneath the imports:

```
const protocol = (location.protocol != 'https:') ? 'ws://':  
'wss://';  
const port = location.port ? ':'+location.port: ";
```

Subscriptions on the Apollo Client

- Remove the `createUploadLink` function call from the `ApolloLink.from` function and add it before the `ApolloClient` class:

```
const httpLink = createUploadLink({  
  uri: location.protocol + '//' + location.hostname + port +  
    '/graphql',  
  credentials: 'same-origin',  
});
```

Subscriptions on the Apollo Client

- Add the WebSocket link that's used for the subscriptions next to httpLink.
- It's the second one we pass to the split function:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_3.txt

Subscriptions on the Apollo Client

- The syntax to combine the two links should look as follows:

```
const link = split(  
  ({ query }) => {  
    const { kind, operation } = getMainDefinition(query);  
    return kind === 'OperationDefinition' && operation ===  
      'subscription';  
  },  
  wsLink,  
  httpLink,  
);
```

Subscriptions on the Apollo Client

- Insert the preceding link variable directly before the onError link.
- The createUploadLink function shouldn't be inside the Apollo.from function.

Subscriptions on the Apollo Client

- We begin with the main file of our chats feature, which is the Chats.js file in the client folder.
- I've reworked the render method so that all the markup that initially came directly from this file is now entirely rendered by other child components.
- You can see all the changes in the following code:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_4.txt

Subscriptions on the Apollo Client

- To continue, first create a separate ChatsQuery component.
- We send the request for all chats like before.
- The render method of the ChatsQuery component should look as follows:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_5.txt

Subscriptions on the Apollo Client

- We have three necessary dependencies that you should import at the top of the list.js file where the ChatsList class is saved:

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

Subscriptions on the Apollo Client

- Parse the GraphQL subscription string.
- The chats query was executed previously, and ChatsList receives the response.
- The new messageAdded subscription has to look as follows:
- Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_6.txt

Subscriptions on the Apollo Client

- Create the new ChatsList class like a standard React component.
- You can copy the shorten function from Chats.js and remove it from there.
- It should look like this:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_7.txt

Subscriptions on the Apollo Client

- The usernamesToString function changes a bit, and I have also added a new getAvatar function.
- When we first created the chats functionality, there was no authentication system.
- We are now going to rewrite this and use the information we have at our disposal to display the correct data.
- Copy these functions into our new class:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_8.txt

Subscriptions on the Apollo Client

- The render method returns the same markup we had in the Chats component.
- It's now way more readable as it's in a separate file.
- The code should look as follows:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_9.txt

Subscriptions on the Apollo Client

- To export your ChatsList class correctly, use the withApollo HoC:

```
export default withApollo(ChatsList)
```

Subscriptions on the Apollo Client

- Here's the crucial part.
- At the moment, the ChatsList component is mounted, so we subscribe to the messageAdded channel.
- Only then is the messageAdded subscription used to receive new data or, to be exact, new chat messages.
- To start subscribing to the GraphQL subscription, we have to add a new method to the componentDidMount method:

```
componentDidMount() {  
  this.subscribeToNewMessages();  
}
```

Subscriptions on the Apollo Client

- We have to add the corresponding `subscribeToNewMessages` method as well.
- We're going to explain every bit of this function in a moment.
- Insert the following code into the `ChatsList` class:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_10.txt

Subscriptions on the Apollo Client

- Insert the following code right before the final return statement inside the updateQuery function:

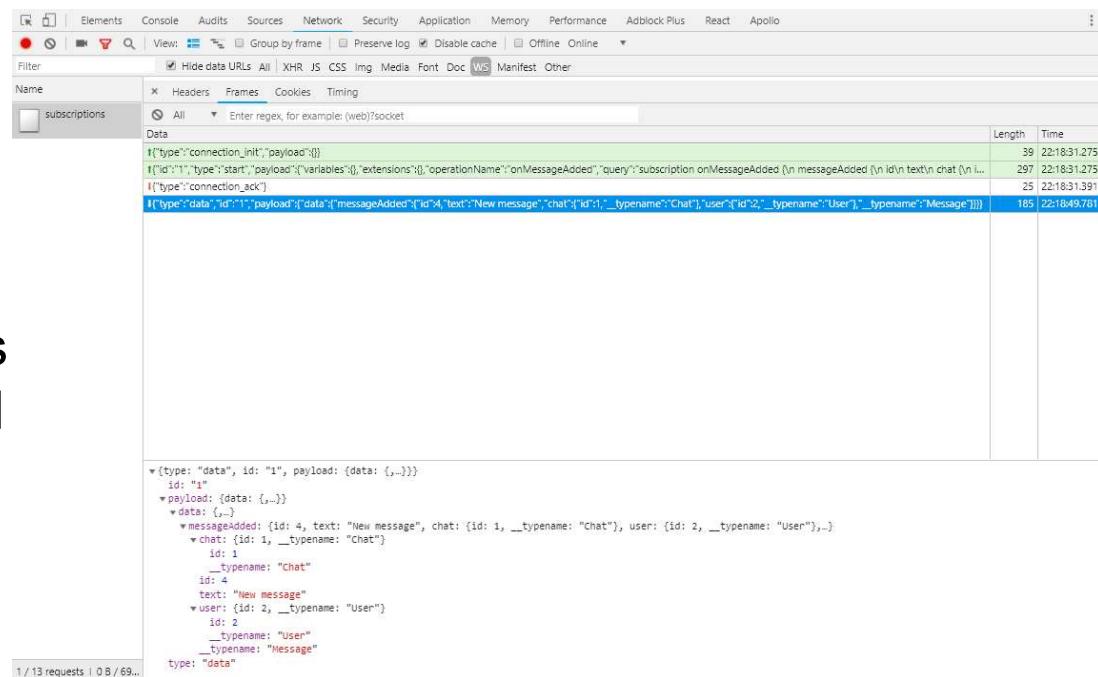
```
try {  
  const data = self.props.client.store.cache.readQuery({ query:  
    GET_CHAT, variables: { chatId:  
      subscriptionData.data.messageAdded.chat.id } });  
  if(user.id !== subscriptionData.data.messageAdded.user.id) {  
    data.chat.messages.push(subscriptionData.data.messageAdded);  
    self.props.client.store.cache.writeQuery({ query: GET_CHAT,  
      variables: { chatId: subscriptionData.data.messageAdded.chat.id },  
      data });  
  }  
} catch(e) {}
```

Authentication with Apollo Subscriptions

- How is it possible for the user to receive new messages when they aren't authenticated on the back end for the WebSocket transport protocol?
- The best way to figure this out is to have a look at your browser's developer tools.

Authentication with Apollo Subscriptions

- Try this scenario with the Developer Tools open.
- You should see the same WebSocket frames for all browsers. It should look like the following screenshot:



Authentication with Apollo Subscriptions

- In index.js, from the subscriptions folder of the server, add the following code to the first parameter of the SubscriptionServer initialization.
- It accepts an onConnect parameter as a function, which is executed whenever a client tries to connect to the subscriptions endpoint.
- Add the code just before the schema parameter:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_11.txt

Authentication with Apollo Subscriptions

- We have now identified the user that has connected to our back end with the preceding code, but we're still sending every frame to all users.
- This is a problem with the resolver functions because they don't use the context yet.
- Replace the messageAdded subscription with the following code in the resolvers.js file:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_12.txt

Notifications with Apollo Subscriptions

- Follow these steps to get your first Subscription component running:
- Create a subscriptions folder inside the client's components folder.
- You can save all subscriptions that you implement using Apollo's Subscription component inside this folder.
- Insert a messageAdded.js file into the folder and paste in the following code:
- Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_13.txt

Notifications with Apollo Subscriptions

- Because we want to show notifications to the user when a new message is received, we install a package that takes care of showing pop-up notifications. Install it using npm:

```
npm install --save react-toastify
```

Notifications with Apollo Subscriptions

- To set up react-toastify, add a ToastContainer component to a global point of the application where all notifications are rendered.
- Import the dependency at the top of it:
`import { ToastContainer } from 'react-toastify';`

Notifications with Apollo Subscriptions

- Inside the render method, the first thing to render should be ToastContainer.
- Add it like in the following code:

```
<div className="wrapper">  
  <ToastContainer/>
```

Notifications with Apollo Subscriptions

- To handle the subscription data, we need a child component that gets the data as a property.
- To do this, create a notification.js file inside the chats component folder.
- The file should look as follows:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_14.txt

Notifications with Apollo Subscriptions

- Import ChatNotification and the MessageAddedSubscription component inside the Chats.js file:

```
import MessageAddedSubscription from './components/subscriptions/  
/messageAdded';  
import ChatNotification from './components/chat/notification';
```

Notifications with Apollo Subscriptions

- Include both components in the render method of the Chats class from the Chats.js file.
- The final method looks like this:
Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/10_15.txt

Notifications with Apollo Subscriptions

- Add a small CSS rule and import the CSS rules of the react-toastify package.
- Import the CSS in the App.js file:
`import 'react-toastify/dist/ReactToastify.css';`

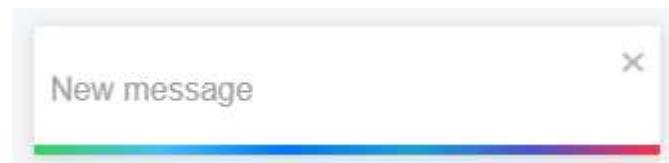
Notifications with Apollo Subscriptions

- Then, add these few lines to the custom style.css file:

```
.Toastify__toast-container--top-left {  
  top: 4em !important;  
}
```

Notifications with Apollo Subscriptions

- You can see an example of a notification in the following screenshot:



Summary

- This lesson aimed to offer the user a real-time user interface that allows them to chat comfortably with other users.
- We also looked at how to make this UI extendable.
- You learned how to set up subscriptions with any Apollo or GraphQL back end for all entities.

"Complete Lab 10"

11: Writing Tests



Writing Tests

This lesson covers the following topics:

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

Writing Tests

- The problem we're facing is that we have to ensure the quality of our software without increasing the amount of manual testing.
- It isn't possible to recheck every feature of our software when new updates are released.
- To solve this problem, we're going to use Mocha, which is a JavaScript testing framework.

Writing Tests

- 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
```

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"
```

Our first Mocha test

- 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"]  
});
```

Our first Mocha test

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/11_1.txt

Our first Mocha test

- 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 Mocha test

- 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)
```

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.

Starting the back end with Mocha

- 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')();
```

Starting the back end with Mocha

- 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();  
  });  
});
```

Starting the back end with Mocha

- 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;
```

Starting the back end with Mocha

- 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);  
});
```

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.

Starting the back end with Mocha

- 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:
- Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/11_2.txt

Starting the back end with Mocha

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/11_3.txt

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.

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.

Starting the back end with Mocha

- 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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/11_4.txt

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:

Refer to https://github.com/fenago/react-graphql-course/blob/master/snippets/11_5.txt

Testing authenticated requests

- 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:

```
Graphbook application test
  ✓ renders and serves the index page
  404
    ✓ redirects the user when not matching path is found
  authentication
    ✓ redirects the user when not logged in
    ✓ allows the user to sign up
    ✓ allows the user to query all chats

  5 passing (3s)
```

Testing React with Enzyme

- 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
```

Testing React with Enzyme

- 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']);
```

Testing React with Enzyme

- 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;
```

Testing React with Enzyme

- 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()
      });
    });
});
```

Testing React with Enzyme

- 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=  
        {"\\"/>/<\\">  
    )  
  }  
}
```

Testing React with Enzyme

- We use the mount function of Enzyme to render the Graphbook class to the DOM:
- ```
const wrapper = mount(<Graphbook />);
```
- 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');
```

# Testing React with Enzyme

- 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');
```

# Testing React with Enzyme

- We check whether the form was changed correctly:

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

# Testing React with Enzyme

- 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:

Refer to [https://github.com/fenago/react-graphql-course/blob/master/snippets/11\\_6.txt](https://github.com/fenago/react-graphql-course/blob/master/snippets/11_6.txt)

# Summary

- In this lesson, 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.

# "Complete Lab 11"

# 12: Continuous Deployment with CircleCI and Heroku



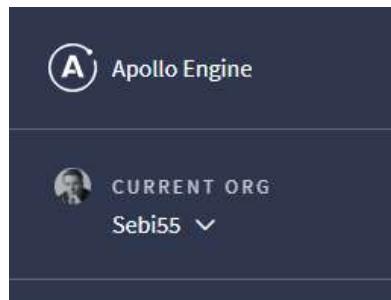
# Continuous Deployment with CircleCI and Heroku

This lesson covers the following topics:

- Setting up Apollo Engine
- Schema analysis
- Performance analytics
- Error tracking

# Setting up Apollo Engine

- Apollo Engine provides many great features, which we'll explore in this lesson.
- Before moving on, however, you need to sign up for an Apollo Engine account.
- Apollo Engine is a commercial product produced by MDG, the Meteor Development Group, the company behind Apollo.



## Service List

NEW SERVICE

Service List

Org settings

- Docs
- Contact support
- Status Report
- Log out



# Setting up Apollo Engine

- You can do this using the command provided by Apollo Engine.
- Copy it directly from the website and execute it.
- For me, this command looked as follows:

```
npx apollo service:push --
endpoint="http://localhost:8000/graphql" --
key="YOUR_KEY"
```

# Setting up Apollo Engine

- It can be added inside the index.js file of the graphql folder:

```
const server = new ApolloServer({
 schema: executableSchema,
 introspection: true,
```

# Setting up Apollo Engine

- Open index.js in the server's graphql folder and add the following object to the ApolloServer initialization:

```
engine: {
 apiKey: ENGINE_KEY
}
```

- The ENGINE\_KEY variable should be extracted from the environment variables at the top of the file.
- We also need to extract JWT\_SECRET with the following line:

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

# Analyzing schemas with Apollo Engine

The screenshot shows the Apollo Engine Registry interface. At the top, there's a header with "Explorer" and a timestamp "Last hour". A search bar says "Search schema...". Below the header, tabs for "Registry" (which is selected) and "Deprecations" are visible. A summary box displays "Version f470de", "Committed on November 27 at 6:34pm", and performance metrics: "1.6 requests/min", "17 types", and "38 fields". A "Version History" link is also present. The main content area is titled "Root types" and contains a table with the following data:

| RootMutation         |                                    |                         |             |
|----------------------|------------------------------------|-------------------------|-------------|
| Field                | Arguments                          | Used by                 | Description |
| addPost: Post        | { post: PostInput! }               | In the last hour        |             |
| updatePost: Post     | { post: PostInput!, postId: Int! } |                         |             |
| deletePost: Response | ( postId: Int! )                   |                         |             |
| addChat: Chat        | { chat: ChatInput! }               |                         |             |
| addMessage: Message  | { message: MessageInput! }         |                         |             |
| login: Auth          | { email: String!, password: ... }  | 1 clients, 1 operations |             |
| logout: Response     |                                    |                         |             |

# Analyzing schemas with Apollo Engine

- I have created an example for you in the following screenshot:

Schema History

The screenshot shows the Apollo Engine Schema History interface. On the left, under "Schema versions", there are two entries:

- 27 November 2018 (today) - A commit by Sebastian Grebe (0da332) at 11:58 pm. It shows 1 type added (+1), 0 types removed (-0), and 0 fields added (+0). A green arrow icon indicates a forward diff.
- initial publish (6:34 pm) - A commit by Sebastian Grebe (0da332) at 6:34 pm. This is the base version.

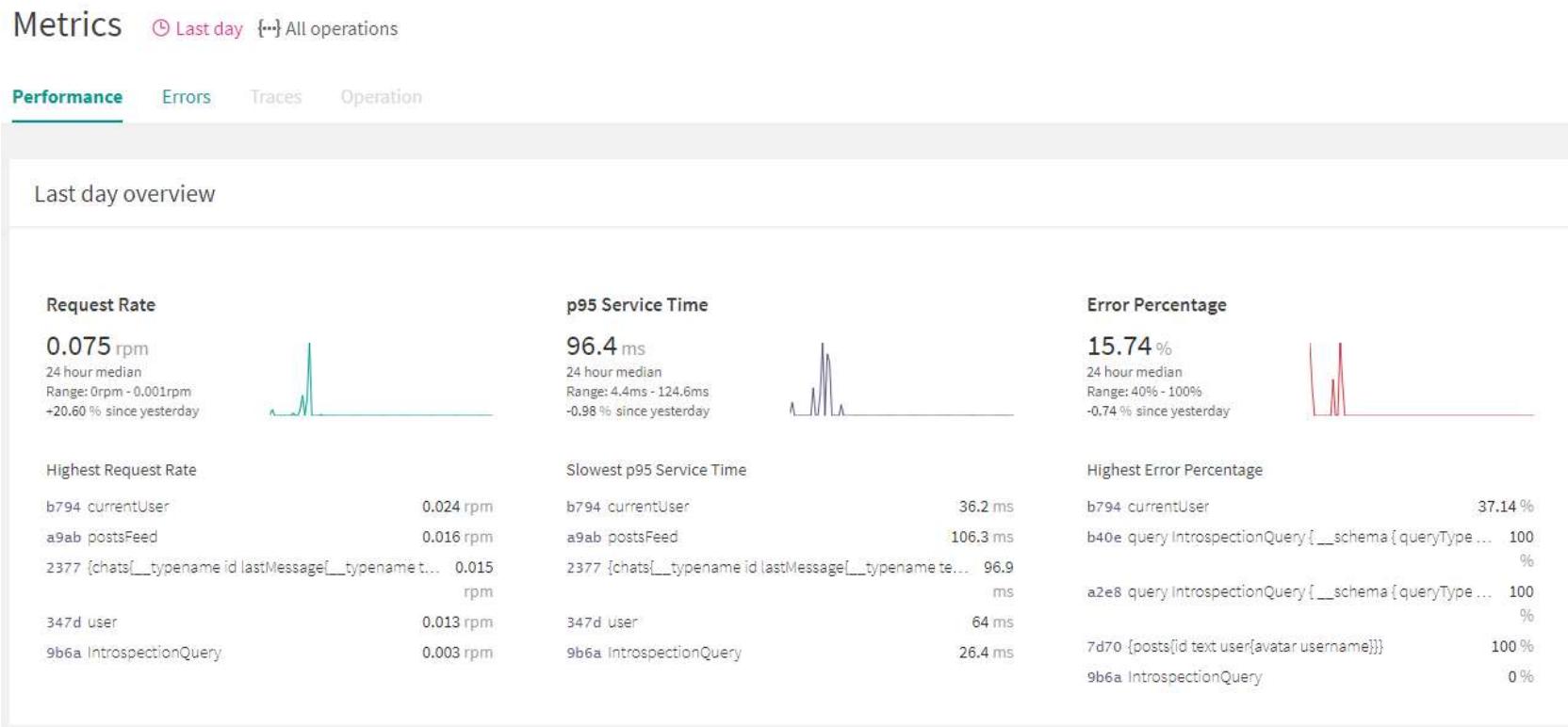
On the right, under "Schema diff", it shows the changes from the initial publish to the latest version:

| Field              | Description                     |
|--------------------|---------------------------------|
| + example: String! | Demonstration.example was added |

# Performance metrics with Apollo Engine

- When your application is live and heavily used, you can't check the status of every feature yourself; it would lead to an impossible amount of work.
- Apollo Engine can tell you how your GraphQL API is performing by collecting statistics with each request that's received.

- At the top of the Metrics page, you have four tabs. The first tab will look as follows:



# Performance metrics with Apollo Engine

- Now, switch to the Traces tab at the top.
- The first chart on this page looks as follows:



## Execution

| Resolvers          | Timing | TTL |
|--------------------|--------|-----|
| postsFeed          | 93.0ms |     |
| postsFeed:PostFeed | <1ms   |     |
| posts:[Post]       | 38.3ms |     |
| posts.0            | 29.0ms |     |
| posts.1            | 27.9ms |     |
| posts.2            | 27.4ms |     |
| posts.3            | 34.5ms |     |
| posts.4            | 34.1ms |     |
| posts.5            | 33.6ms |     |
| posts.6            | 36.8ms |     |
| posts.7            | 36.3ms |     |
| posts.8            | 41.1ms |     |
| posts.9            |        |     |
| id:Int             | <1ms   |     |
| text:String        | <1ms   |     |
| user:User          | 42.2ms |     |
| avatar:String      | <1ms   |     |
| username:String    | <1ms   |     |

# Error tracking with Apollo Engine

- We've already looked at how to inspect single operations using Apollo Engine.
- Under the Clients tab, you will find a separate view that covers all client types and their requests:

The screenshot shows two main sections of the Apollo Engine interface. On the left, there's a summary card titled 'Activity in the last day' with metrics for 'Unidentified clients', 'All versions' (13.73% errors), and '7 operations'. A prominent green button labeled '100% requests' with the number '102' indicates no errors. On the right, a detailed table titled 'Operations' lists various GraphQL operations with their request counts and error percentages.

| ID   | Operation Name                               | Requests | Errors |
|------|----------------------------------------------|----------|--------|
| b794 | currentUser                                  | 35       | 37.14% |
| a9ab | postsFeed                                    | 23       |        |
| 2377 | {chats{__typename id lastMessage{__type...}} | 22       |        |
| 347d | user                                         | 18       |        |
| 9b6a | IntrospectionQuery                           | 2        |        |
| 286e | login                                        | 1        |        |
| b40e | query IntrospectionQuery { __schema { q...   | 1        | 100%   |

# Error tracking with Apollo Engine

- In this example, we'll use HTTP header fields to track the client type.
- There will be two header fields: `apollo-client-name` and `apollo-client-version`. We'll use these to set custom values to filter requests later in the Clients page.
- Open the `index.js` file from the `graphql` folder, Add the following function to the `engine` property of the `ApolloServer` initialization:

Refer to [https://github.com/fenago/react-graphql-course/blob/master/snippets/12\\_1.txt](https://github.com/fenago/react-graphql-course/blob/master/snippets/12_1.txt)

# Error tracking with Apollo Engine

To get both of our clients – the front end and back end – set up, we have to add these fields. Perform the following steps:

- Open the index.js file of the client's apollo folder file.
- Add a new InfoLink to the file to set the two new header fields:

Refer to [https://github.com/fenago/react-graphql-course/blob/master/snippets/12\\_2.txt](https://github.com/fenago/react-graphql-course/blob/master/snippets/12_2.txt)

- Add InfoLink in front of AuthLink in the ApolloLink.from function.
- On the back end, we need to edit the apollo.js file in the ssr folder:

```
const InfoLink = (operation, next) => {
 operation.setContext(context => ({
 ...context,
 headers: {
 ...context.headers,
 'apollo-client-name': 'Apollo Backend Client',
 'apollo-client-version': '1'
 },
 }));
 return next(operation);
};
```

## Activity in the last hour

### Apollo Backend Client

All versions  
4 operations

84%  
656 requests >

| Versions | Requests | % Errors | > |
|----------|----------|----------|---|
| 1        | 656      | -        | > |

### Unknown Client

All versions  
5 operations

11%  
83 requests >

| Versions    | Requests | % Errors | > |
|-------------|----------|----------|---|
| Unversioned | 83       | -        | > |

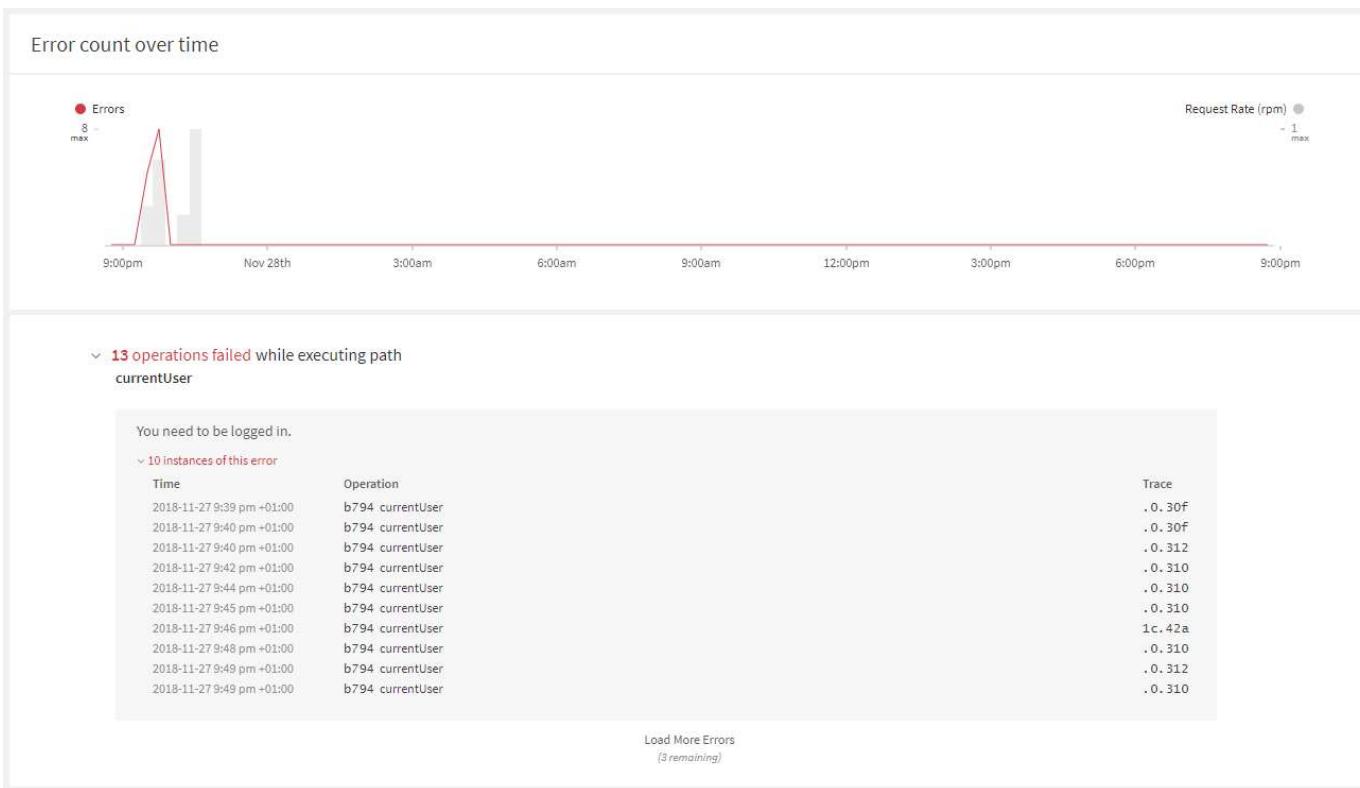
### Apollo Frontend Client

All versions  
1 operations

< 1%  
1 requests >

| Versions | Requests | % Errors | > |
|----------|----------|----------|---|
| 1        | 1        | -        | > |

- When you visit the Error tab, you will be presented with a screen that looks like the following screenshot:



# Caching with Apollo Server and the Client

- The client sends a hash instead of the full query string.
- Apollo Server tries to find this hash inside its cache.
- If the server finds the corresponding query string to the hash, it'll execute it and respond with its result.
- If the server doesn't find the hash inside its cache, it'll ask the client to send the hash along with the actual query string.
- The back end will then save this hash with the query string for all future requests and respond to the client's request.

# Caching with Apollo Server and the Client

- There are two server-side changes that we have to do.
- One is in the initialization of Apollo Server.
- Extend the graphql index.js by adding the following two parameters to the ApolloServer options:

```
cacheControl: {
 defaultMaxAge: 5,
 stripFormattedExtensions: false,
 calculateCacheControlHeaders: true,
},
```

# Caching with Apollo Server and the Client

- The second change is to enable cache control in the GraphQL schema.
- Just copy the following code into the schema.js file:

```
enum CacheControlScope {
 PUBLIC
 PRIVATE
}
directive @cacheControl (
 maxAge: Int
 scope: CacheControlScope
) on FIELD_DEFINITION | OBJECT | INTERFACE
```

# Caching with Apollo Server and the Client

- If you now execute any query, the response will include an extensions object that looks like the following example:

```
▼ cacheControl: {version: 1, hints: [{path: ["postsFeed"], maxAge: 5}, {path: ["postsFeed", "posts"], maxAge: 5},...]}
 ▼ hints: [{path: ["postsFeed"], maxAge: 5}, {path: ["postsFeed", "posts"], maxAge: 5},...]
 ▶ 0: {path: ["postsFeed"], maxAge: 5}
 ▶ 1: {path: ["postsFeed", "posts"], maxAge: 5}
 ▶ 2: {path: ["postsFeed", "posts", 0, "user"], maxAge: 120}
 ▶ 3: {path: ["postsFeed", "posts", 1, "user"], maxAge: 120}
 ▶ 4: {path: ["postsFeed", "posts", 2, "user"], maxAge: 120}
 ▶ 5: {path: ["postsFeed", "posts", 3, "user"], maxAge: 120}
 ▶ 6: {path: ["postsFeed", "posts", 4, "user"], maxAge: 120}
 ▶ 7: {path: ["postsFeed", "posts", 5, "user"], maxAge: 120}
 ▶ 8: {path: ["postsFeed", "posts", 6, "user"], maxAge: 120}
 ▶ 9: {path: ["postsFeed", "posts", 7, "user"], maxAge: 120}
 ▶ 10: {path: ["postsFeed", "posts", 8, "user"], maxAge: 120}
 ▶ 11: {path: ["postsFeed", "posts", 9, "user"], maxAge: 120}
 ▶ 12: {path: ["postsFeed", "posts", 0, "user", "avatar"], maxAge: 240}
 ▶ 13: {path: ["postsFeed", "posts", 1, "user", "avatar"], maxAge: 240}
 ▶ 14: {path: ["postsFeed", "posts", 3, "user", "avatar"], maxAge: 240}
 ▶ 15: {path: ["postsFeed", "posts", 4, "user", "avatar"], maxAge: 240}
 ▶ 16: {path: ["postsFeed", "posts", 2, "user", "avatar"], maxAge: 240}
 ▶ 17: {path: ["postsFeed", "posts", 5, "user", "avatar"], maxAge: 240}
 ▶ 18: {path: ["postsFeed", "posts", 6, "user", "avatar"], maxAge: 240}
 ▶ 19: {path: ["postsFeed", "posts", 7, "user", "avatar"], maxAge: 240}
 ▶ 20: {path: ["postsFeed", "posts", 8, "user", "avatar"], maxAge: 240}
 ▶ 21: {path: ["postsFeed", "posts", 9, "user", "avatar"], maxAge: 240}
version: 1
```

# Caching with Apollo Server and the Client

- If you open your schema.js file, you can change your User type to reflect the preceding screenshot, as follows:

```
type User @cacheControl(maxAge: 120) {
 id: Int
 avatar: String @cacheControl(maxAge: 240)
 username: String
 email: String
}
```

# Caching with Apollo Server and the Client

- To persist our queries, we have to change some aspects of our SSR code.
- Before we do this, however, we need to install a package using npm:

```
npm install --save apollo-link-persisted-queries
```

- This package provides a special Apollo Client link to use persisted queries.
- Import it at the top of both the `apollo.js` file in the `ssr` folder and the `index.js` in the `apollo` folder of the client:

```
import { createPersistedQueryLink } from 'apollo-link-persisted-queries';
```

# Caching with Apollo Server and the Client

- The following snippet shows how this can be implemented for the client-side code:

```
const httpLink =
createPersistedQueryLink().concat(createUploadLink({
 uri: location.protocol + '//' + location.hostname + port +
 '/graphql',
 credentials: 'same-origin',
}));
```

# Caching with Apollo Server and the Client

- The SSR-related code looks pretty similar, but the function is directly executed within the Apollo.from function instead.
- In the apollo.js file from the apollo folder, replace the initialization of HttpLink with the following code:

```
createPersistedQueryLink().concat(new HttpLink({
 uri: 'http://localhost:8000/graphql',
 credentials: 'same-origin',
 fetch
}));
```

# Caching with Apollo Server and the Client

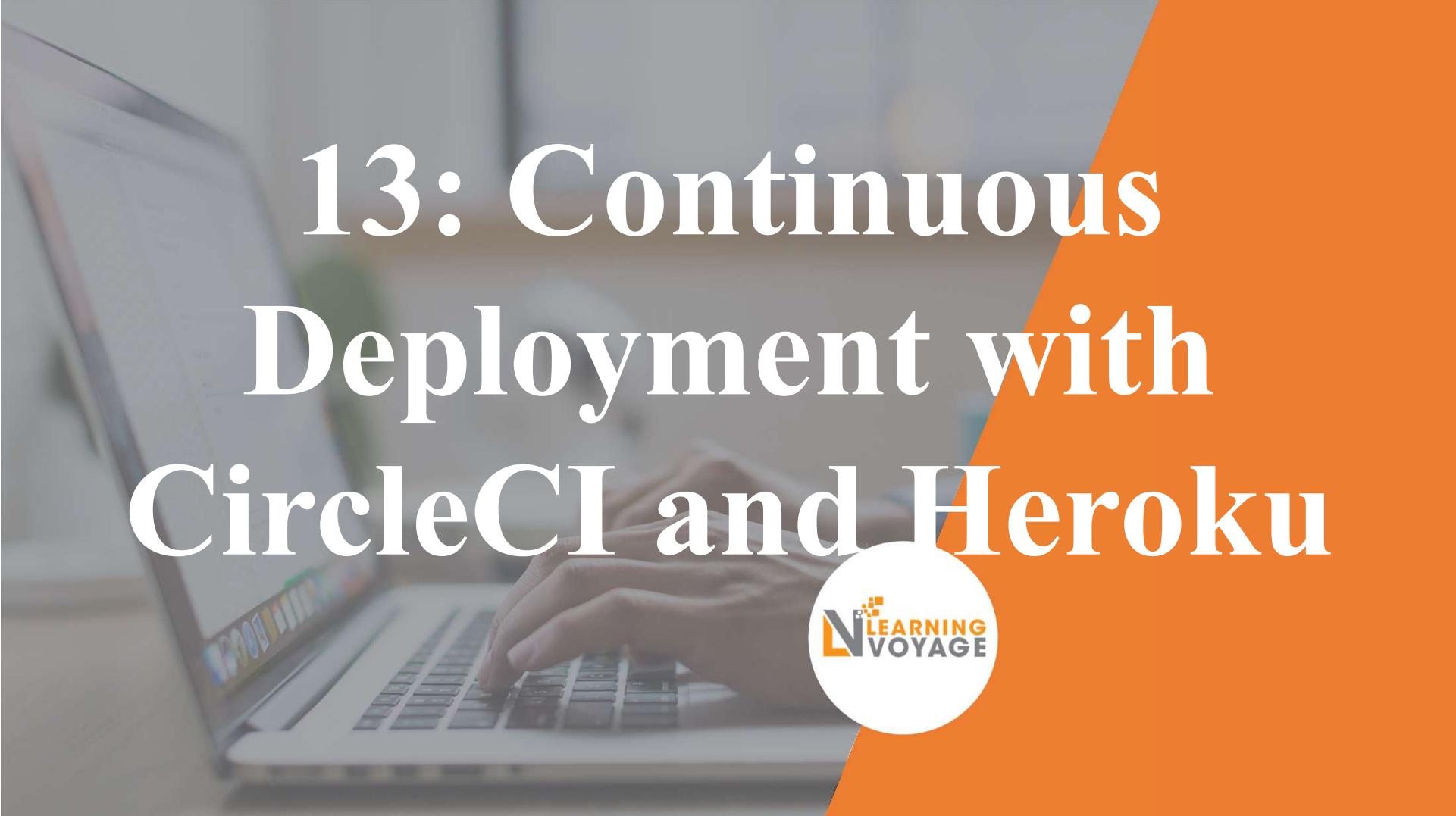
- A GraphQL request will now include a hash instead of the regular query string on the first try.
- You can see an example in the following screenshot:

```
▼ {operationName: "postsFeed", variables: {page: 0, limit: 10}, extensions: {,...}}
 ▼extensions: {,...}
 ▼persistedQuery: {version: 1, sha256Hash: "03ce590cb5e466a07ca4620a2d6067a8e66c94019fabd678c3e25ced41fa6ea6"}
 sha256Hash: "03ce590db5e466a07ca4620a2d6067a8e66c94019fabd678c3e25ced41fa6ea6"
 version: 1
 operationName: "postsFeed"
 ▶ variables: {page: 0, limit: 10}
```

# Summary

- In this lesson, we learned how to sign up to and set up Apollo Engine.
- You should now understand all the features that Apollo Engine provides and how to make use of collected data.
- We also looked at how to set up cacheControl and Automatic Persisted Queries to improve the performance of your application.

# "Complete Lab 12"



# 13: Continuous Deployment with CircleCI and Heroku



# Continuous Deployment with CircleCI and Heroku

This lesson covers the following topics:

- Production-ready bundling
- What is Docker?
- What is continuous integration/deployment?
- Configuring Docker
- Setting up continuous deployment with CircleCI
- Deploying our application to Heroku

# Preparing the final production build

- Currently, we use our application in a development environment while working on it. It is not highly optimized for performance or low bandwidth usage.
- We include developer functionalities with the code so that we can debug it properly.
- We also only generate one bundle, which is distributed at all times.
- No matter which page the user visits, the code for our entire application is sent to the user or browser.

# Code-splitting with React Loadable and webpack

- We will begin by installing a few packages that we need to implement this technique.
- Install them using npm, as follows:

```
npm install --save-dev @babel/plugin-syntax-dynamic-import babel-plugin-dynamic-import-node webpack-node-externals @babel/plugin-transform-runtime
npm install --save react-loadable
```

- Aside from React and React Router, you can replace all import statements at the top of the file with the following code:

```
import loadable from 'react-loadable';
import Loading from './components/loading';
const User = loadable({
 loader: () => import('./User'),
 loading: Loading,
});
const Main = loadable({
 loader: () => import('./Main'),
 loading: Loading,
});
const LoginRegisterForm = loadable({
 loader: () => import('./components/loginregister'),
 loading: Loading,
});
```

# Code-splitting with React Loadable and webpack

Edit the output property of the webpack configuration to include the chunkFilename field, as follows:

```
output: {
 path: path.join(__dirname, outputDirectory),
 filename: "bundle.js",
 publicPath: '/',
 chunkFilename: '[name].[chunkhash].js'
},
```

# Code-splitting with React Loadable and webpack

- To make use of React Loadable, we rely on the `ReactLoadablePlugin` that it provides.
- Import the following plugin at the top of the file:

```
const { ReactLoadablePlugin } = require('react-loadable/webpack');
```

# Code-splitting with React Loadable and webpack

- Since we are using SSR with our application, we can remove the part where we insert our bundle and other files in the HTML template by using the HtmlWebpackPlugin.
- We are going to replace it with the preceding ReactLoadablePlugin.
- Insert the following code, instead of the HtmlWebpackPlugin:

```
new ReactLoadablePlugin({
 filename: './dist/react-loadable.json',
}),
```

# Code-splitting with SSR

- When rendering our application on the server, we have to tell the client which bundles to download on the initial page load.
- Open the server's index.js file to implement this logic.
- Import the react-loadable dependencies at the top of the file, as follows:

```
import Loadable, { Capture } from 'react-loadable';
import { getBundles } from 'react-loadable/webpack';
```

# Code-splitting with SSR

- To let our back end know which bundles exist, we load the previously generated JSON file with the following code.
- Insert it directly underneath the import statements:

```
if(process.env.NODE_ENV !== 'development') {
 var stats = require('../dist/react-loadable.json');
}
```

# Code-splitting with SSR

- Replace the server.listen method call in the services for loop with the following code:

```
Loadable.preloadAll().then(() => {
 server.listen(process.env.PORT? process.env.PORT:8000,
() => {
 console.log('Listening on port '+ (process.env.PORT?
 process.env.PORT:8000)+ '!');
 services[name](server);
});
});
```

# Code-splitting with SSR

- To reuse the server-side rendered code and declare which modules are being used, we have to edit our .babelrc file.
- Add the following lines of code to the plugins section of the .babelrc file:

```
"@babel/plugin-syntax-dynamic-import",
"react-loadable/babel"
```

# Code-splitting with SSR

- In our app.get catch-all Express route, where all SSR requests are processed, we have to add the Capture component of React Loadable to our App component.
- Replace the current App variable with the following code lines:

```
const modules = [];
const App = (<Capture report={moduleName =>
modules.push(moduleName)}><Graphbook client={client}
loggedIn={loggedIn} location={req.url}
context={context}>/</Capture>);
```

- The final renderToStringWithData function call should look as follows:

```
renderToStringWithData(App).then((content) => {
 if (context.url) {
 res.redirect(301, context.url);
 } else {
 var bundles;
 if(process.env.NODE_ENV !== 'development') {
 bundles = getBundles(stats, Array.from(new Set(modules)));
 } else {
 bundles = [];
 }
 const initialState = client.extract();
 const head = Helmet.renderStatic();
 res.status(200);
 res.send(`<!doctype html>\n${template(content, head, initialState,
 bundles)}</body>`);
 res.end();
 }
});
```

# Code-splitting with SSR

- To do so, we pass the final bundles array to our template function.
- We have to adjust our template.js file from the server's ssr folder to accept and render the bundles variable.
- First, change the template function's signature to match the following line of code:

```
export default function htmlTemplate(content, head, state, bundles) {
```

# Code-splitting with SSR

- We just added the bundles as the fourth parameter, Next, we have to include all of the bundles in the HTML.
- As it is a simple array of objects, we can use the JavaScript map function to process all bundles.
- Insert the following line of code above the script tag, with the bundle.js file as the src attribute:

```
 ${bundles.map(bundle => `<script
src="${bundle.publicPath}"></script>`).join('\n')}
```

# Code-splitting with SSR

- To bundle our back end, we are going to set up a new webpack configuration file.
- Create a `webpack.server.build.config.js` file next to the other webpack files with the following content:

Refer to [https://github.com/fenago/react-graphql-course/blob/master/snippets/13\\_1.txt](https://github.com/fenago/react-graphql-course/blob/master/snippets/13_1.txt)

# Code-splitting with SSR

- Add the following two lines to the scripts field of the package.json file:

```
"build": "npm run client:build && npm run server:build",
"server:build": "webpack --config
webpack.server.build.config.js"
```

# Code-splitting with SSR

- The old npm run server command would start the server-side code in the unbundled version, which is not what we want.
- Insert the following line into the package.json file:

`"server:production": "node dist/server/bundle.js"`

# Code-splitting with SSR

- Replace our npm run server command with the following line, in the package.json file:

```
"server": "nodemon --exec babel-node --plugins require-context-hook,dynamic-import-node --watch src/server src/server/index.js",
```

# Code-splitting with SSR

- For our test, the only thing that we have to change is the babel-hook.js file to let Babel transpile everything correctly.
- Add the following plugins to the babel-hook.js file:

"react-loadable/babel", "dynamic-import-node"

# Code-splitting with SSR

- Our test runs in the production environment, because only then can we verify that all features that are enabled in the live environment work correctly.
- Because we have just introduced React Loadable, which generates a JSON file when building the client-side code, we have to run a full build when we are testing our application.
- Edit the test command of the package.json file to reflect this change, as follows:

```
"test": "npm run build && mocha --exit test/ --require babel-hook --
require @babel/polyfill --recursive"
```

# Setting up Docker

- Publishing an application is a critical step that requires a lot of work and care.
- Many things can go wrong when releasing a new version.
- We have already made sure that we can test our application before it goes live.
- After deployment, we will have Apollo Engine, which will inform us about anything that goes well and anything that goes wrong.

# What is Docker?

- One major trending piece of software is called Docker.
- It was released in 2013, and its aim is at isolating the application within a container by offering its own runtime environment, without having access to the server itself.
- The aim of a container is to isolate the application from the operating system of the server.

# Installing Docker

- Update your system's package manager, as follows:  
`sudo apt-get update`
- Install all of the dependencies for Docker, as follows:  
`sudo apt-get install apt-transport-https ca-certificates  
curl gnupg2 software-properties-common`

# Installing Docker

- Verify and add the GNU Privacy Guard (GPG) key for the Docker repository, as follows:

```
curl -fsSL https://download.docker.com/linux/debian/gpg
| sudo apt-key add -
```

# Installing Docker

- Now that the GPG key has been imported, we can add the repository to the package manager, as follows:

```
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/debian $(lsb_release -
cs) stable"
```

# Installing Docker

- After adding the new repository, you must update the package manager's index again, as follows:

`sudo apt-get update`

- Lastly, we can install the Docker package to our system, as follows:

`sudo apt-get install docker-ce`

# Dockerizing your application

- Many companies have adopted Docker and replaced their old infrastructure setup, thereby largely reducing system administration.
- Still, there is some work to do before deploying your application straight to production.

# Writing your first Dockerfile

- The conventional approach to generating a Docker image of your application is to create a Dockerfile in the root of your project, But what does the Dockerfile stand for?
- A Dockerfile is a series of commands that are run through the Docker CLI.

# Writing your first Dockerfile

- Docker Hub is the official container image registry, providing many minimalistic images.
- Just insert the following line inside of our new Dockerfile, in the root of our project:

**FROM node:10**

# Writing your first Dockerfile

- The next step for our Dockerfile is to create a new folder, in which the application will be stored and run.
- Add the following line to the Dockerfile:

`WORKDIR /usr/src/app`

# Writing your first Dockerfile

- As the third line of our Dockerfile, add the following code:

**COPY ..**

# Writing your first Dockerfile

- Create a `.dockerignore` file in the root of the project folder, next to the Dockerfile, and enter the following content:

```
node_modules
package-lock.json
```

# Writing your first Dockerfile

- Install the npm packages inside of the Docker image that we are creating at the moment.
- Add the following line of code to the Dockerfile:

**RUN npm install**

# Writing your first Dockerfile

- Insert the following line at the end of the Dockerfile to make the port accessible from outside of the container:

**EXPOSE 8000**

# Writing your first Dockerfile

- Finally, we have to tell Docker what our container should do once it has booted.
- In our case, we want to start our back end (including SSR, of course). Since this should be a simple example, we will start the development server.
- Add the last line of the Dockerfile, as follows:

```
CMD ["npm", "run", "server"]
```

# Building and running Docker containers

- The MySQL host must allow for remote connections from inside the container.
- Execute the following command to build the Docker image on your local machine:

```
docker build -t sgrebbe/graphbook .
```

# Building and running Docker containers

- When the command has finished generating the image, it should be available locally.
- To prove this, you can use the Docker CLI by running the following command:  
**docker images**
- The output from Docker should look as follows:

| REPOSITORY       | TAG    | IMAGE ID     | CREATED        | SIZE   |
|------------------|--------|--------------|----------------|--------|
| sgrebe/graphbook | latest | fe30bceb0268 | 27 minutes ago | 1.22GB |
| node             | 10     | 75a3a4428e1d | 3 days ago     | 894MB  |

# Building and running Docker containers

- Now, we should be able to start our Docker container with this new image.
- The following command will launch your Docker container:

```
docker run -p 8000:8000 -d --env-file .env
sgrebe/graphbook
```

# Building and running Docker containers

- Let's create the .env file in the root directory of our project.
- Insert the following content, replacing all placeholders with the correct value for every environment variable:

```
ENGINE_KEY=YOUR_APLLO_ENGINE_API_KEY
NODE_ENV=development
JWT_SECRET=YOUR_JWT_SECRET
AWS_ACCESS_KEY_ID=YOUR_AWS_KEY_ID
AWS_SECRET_ACCESS_KEY=YOUR_AWS_SECRET_ACCESS_KEY
```

# Building and running Docker containers

- Consequently, Docker also provides a command to test this, as follows:

`docker ps`

- The docker ps command gives you a list of all running containers.
- You should find the Graphbook container in there, too, The output should appear as follows:

| CONTAINER ID | IMAGE            | COMMAND          | CREATED       | STATUS       | PORTS                  | NAMES        |
|--------------|------------------|------------------|---------------|--------------|------------------------|--------------|
| 08499322a998 | sgrebe/graphbook | "npm run server" | 4 seconds ago | Up 3 seconds | 0.0.0.0:8000->8000/tcp | dreamy_knuth |

# Building and running Docker containers

- In development, it makes sense to have access to the command-line printouts that our application generates.
- When running the container in the detached mode, you have to use the Docker CLI to see the printouts, using the following command.
- Replace the id at the end of the command with the id of your container:

`docker logs 08499322a998`

# Building and running Docker containers

- We are running the container in the detached mode, you will not be able to stop it by just using Ctrl + C, like before.
- Instead, you have to use the Docker CLI again.
- To stop the container again, run the following command:

```
docker rm 08499322a998
```

# Building and running Docker containers

- When working and developing with Docker frequently, you will probably generate many images to test and verify the deployment of your application.
- These take up a lot of space on your local machine.
- To remove the images, you can execute the following command:

```
docker rmi fe30bceb0268
```

# Multi-stage Docker production builds

- Our current Docker image, which we are creating from the Dockerfile, is already useful.
- We want our application to be transpiled and running in production mode, because many things are not optimized for the public when running in development mode.

# Multi-stage Docker production builds

- Our new file starts with the FROM command again. We are going to have multiple FROM statements, because we are preparing a multi-stage build.
- The first one should look as follows:

FROM node:10 AS build

# Multi-stage Docker production builds

- Next, we initialize the working directory, like we did in our first Dockerfile, as follows:

`WORKDIR /usr/src/app`

- It is essential to only copy the files that we really need.
- It hugely improves the performance if you reduce the amount of data/files that need to be processed:

`COPY .babelrc ./`

`COPY package*.json ./`

`COPY webpack.server.build.config.js ./`

`COPY webpack.client.build.config.js ./`

`COPY src transpired`

`COPY assets transpired`

# Multi-stage Docker production builds

- Like in our first Dockerfile, we must install all npm packages; otherwise, our application won't work.
- We do this with the following line of code:

**RUN npm install**

# Multi-stage Docker production builds

- After we have copied all of the files and installed all of the packages, we can start the production build.
- Before doing so, it would make sense to run our automated test.
- Add the test script to the Dockerfile, as follows:  
Refer to [https://github.com/fenago/react-graphql-course/blob/master/snippets/13\\_2.txt](https://github.com/fenago/react-graphql-course/blob/master/snippets/13_2.txt)

# Multi-stage Docker production builds

- After all packages have been installed successfully, we can start the build process.
- We added the build script in the first section of this lesson.
- Add the following line to execute the script that will generate the production bundles in the Docker image:

**RUN npm run build**

# Multi-stage Docker production builds

- To get a clean Docker image that only contains the dist folder and the files that we need to run the application, we will introduce a new build stage that will generate the final image.
- The new stage is started with a second FROM statement, as follows:

FROM node:10

# Multi-stage Docker production builds

- Again, we need to specify the working directory for the second stage, as the path is not copied from the first build stage:

WORKDIR /usr/src/app

# Multi-stage Docker production builds

- Because we have given our first build stage a name, we can access the filesystem of this stage through that name.
- To copy the files from the first stage, we can add a parameter to the COPY statement.
- Add the following commands to the Dockerfile:

```
COPY --from=build /usr/src/app/package.json dB:
```

```
migrate
```

```
COPY --from=build /usr/src/app/dist dB: migrate
```

# Multi-stage Docker production builds

- Notice that we only copy the package.json file and the dist folder.
- Our npm dependencies are not included in the application build inside of the dist folder.
- As a result, we need to install the npm packages in the second build stage, too:

`RUN npm install --only=production`

# Multi-stage Docker production builds

- The last two things to do here are to expose the container port to the public and to execute the CMD command, which will let the image run a command of our package.json file when the container has booted:

**EXPOSE 8000**

**CMD [ "npm", "run", "server:production" ]**

# Amazon Relational Database Service

The screenshot shows the Amazon RDS Dashboard in the EU (Frankfurt) region. The left sidebar includes links for Dashboard, Instances, Clusters, Performance Insights, Snapshots, Automated backups, Reserved instances, Subnet groups, Parameter groups, Option groups, Events, Event subscriptions, and Recommendations (1).

**Resources**

You are using the following Amazon RDS resources in the EU (Frankfurt) region (used/quota)

| DB Instances (1/40)                       | Parameter groups (1) |
|-------------------------------------------|----------------------|
| Allocated storage (20.00 GB/100.00 TB)    | Default (1)          |
| Click here to increase DB instances limit | Custom (0/100)       |

| Reserved instances (0/40)  | Option groups (1)            |
|----------------------------|------------------------------|
| Snapshots (30)             | Default (1)                  |
| Manual (0/100)             | Custom (0/20)                |
| Automated (6)              | Subnet groups (1/50)         |
| Recent events (4)          | Supported platforms VPC      |
| Event subscriptions (0/20) | Default network vpc-58231f33 |

**Create database**

Amazon Relational Database Service (RDS) makes it easy to set up, operate, and scale a relational database in the cloud.

[Restore from S3](#) [Create database](#)

Note: your DB instances will launch in the EU (Frankfurt) region

**Service health**

[View service health dashboard](#)

| Current status                                 | Details                       |
|------------------------------------------------|-------------------------------|
| Amazon Relational Database Service (Frankfurt) | Service is operating normally |

**Additional information**

Getting started with RDS  
Overview and features  
Documentation  
Articles and tutorials  
Data import guide for MySQL  
Data import guide for Oracle  
Data import guide for SQL Server  
New RDS feature announcements  
Pricing  
Forums

**Database Preview Environment**

Get early access to new DB engine versions, before they're generally available. The RDS database preview environment lets you work with upcoming beta, release candidate, and early production versions of PostgreSQL engines. Preview environment instances are fully functional, so you can easily test new features and functionality with your applications. [Info](#)

[Preview PostgreSQL in US EAST \(Ohio\)](#)

- You will be asked for the database specification details.
- The first part of the screen will look as follows:

**Instance specifications**

Estimate your monthly costs for the DB Instance using the [AWS Simple Monthly Calculator](#)

DB engine: MySQL Community Edition

License model: [Info](#) general-public-license

DB engine version: [Info](#) MySQL 5.6.41

**Free tier**  
 The Amazon RDS Free Tier provides a single db.t2.micro instance as well as up to 20 GiB of storage, allowing new AWS customers to gain hands-on experience with Amazon RDS. Learn more about the RDS Free Tier and the instance restrictions [here](#).  
 Only enable options eligible for RDS Free Usage Tier [Info](#)

DB instance class: [Info](#) db.t2.micro — 1 vCPU, 1 GiB RAM

Multi-AZ deployment: [Info](#)  
 Create replica in different zone  
 Creates a replica in a different Availability Zone (AZ) to provide data redundancy, eliminate I/O freezes, and minimize latency spikes during system backups.  
 No

Storage type: [Info](#) General Purpose (SSD)

Allocated storage: 20 GiB  
(Minimum: 20 GiB, Maximum: 20 GiB) Higher allocated storage may improve IOPS performance.

# Amazon Relational Database Service

- You will see a list of security groups and a small view with some tabs at the top, as follows:

The screenshot shows the AWS RDS Management Console interface. At the top, there is a search bar with the placeholder "search : rds-launch-wizard" and a "Create Security Group" button. Below the search bar is a table header with columns: Name, Group ID, Group Name, VPC ID, and Description. A single row is visible in the table, representing a security group named "rds-launch-wizard".

Below the table, a message states: "Created from the RDS Management Console: 2018/12/04 22:25:32".

Underneath the table, a section titled "Security Group: sg-0f25078d222af7bc0" is displayed. It includes tabs for "Description", "Inbound" (which is selected), "Outbound", and "Tags".

In the "Inbound" tab, there is an "Edit" button. Below it is a table with columns: Type, Protocol, Port Range, Source, and Description. One rule is listed: "MySQL/Aurora" (Type), "TCP" (Protocol), "3306" (Port Range), "0.0.0.0/0" (Source), and an empty "Description" field.

# Amazon Relational Database Service

- The credentials that you have specified for the database must be included in the .env file for running our Docker container, as follows:

```
username=YOUR_USERNAME
password=YOUR_PASSWORD
database=YOUR_DATABASE
host=YOUR_HOST
```

# Configuring Continuous Integration

- Many people (especially developers) will have heard of continuous integration (CI) or continuous deployment (CD).
- However, most of them cannot explain their meanings and the differences between the two terms. So, what is continuous integration and deployment, in reality?
- When it comes to going live with your application, it might seem easy to upload some files to a server and then start the application through a simple command in the shell, via SSH.

# Configuring Continuous Integration

Projects » Add Projects » Sebi55/Hands-on-Full-Stack-Web-Development-with-GraphQL-and-React

## Set Up Project

CircleCI helps you ship better code, faster. To kick things off, you'll need to add a `config.yml` file to your project, and start building. After that, we'll start a new build for you each time someone pushes a new commit.

Select from the following options to generate a sample `.yml` for your project.

### Operating System

Linux

macOS

### Language

Clojure

Elixir

Go

Gradle (Java)

Maven (Java)

Node

PHP

Python

Ruby

Scala

Other

### Next Steps

You're almost there! We're going to walk you through setting up a configuration file, committing it, and turning on our listener so that CircleCI can test your commits.

Want to skip ahead? Jump right into our documentation, set up a `.yml` file, and kick off your build with the button below.



If you start building before you've added a configuration file, your project will not run. To build your project, add a `.circleci/config.yml` file. [Add a project on CircleCI 2.0](#).

1. Create a folder named `.circleci` and add a file `config.yml` (so that the filepath is in `.circleci/config.yml`).
2. Populate the `config.yml` with the contents of the sample `.yml` (shown below).
3. Update the sample `.yml` to reflect your project's configuration.
4. Push this change up to GitHub.
5. Start building! This will launch your project on CircleCI and make our webhooks listen for updates to your work.

[Copy To Clipboard](#)

[Start building](#)

# Configuring Continuous Integration

- Insert the following code into the config.yml file:

```
version: 2
jobs:
 build:
 docker:
 - image: circleci/node:10
 steps:
 - checkout
 - setup_remote_docker:
 docker_layer_caching: true
 - run:
 command: echo "This is working"
```

# Configuring Continuous Integration

Jobs » Sebi55 » HandsOn » master » 37 (build)

2.0 C Rerun workflow ⚙

**SUCCESS** Finished: 32 min ago (00:05) Previous: 36 Parallelism: 1x out of 1x Queued: 00:00 waiting + 00:01 in queue Resources: 2CPU/4096MB Workflow: workflow Context: N/A Triggered by: Sebastian Grebe (pushed c14df52)

COMMITS (1)  
Sebastian Grebe -o c14df52 circle

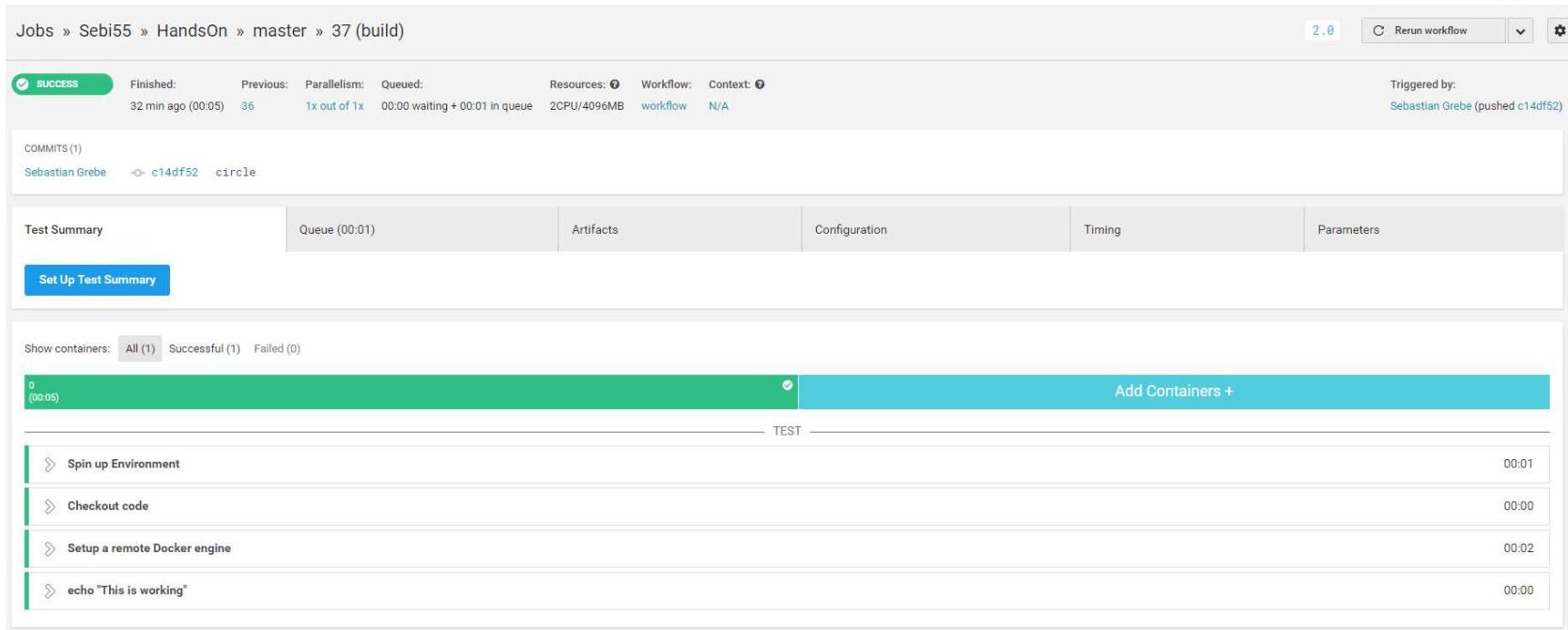
Test Summary Queue (00:01) Artifacts Configuration Timing Parameters

Set Up Test Summary

Show containers: All (1) Successful (1) Failed (0)

0 (00:05) Add Containers + TEST

Spin up Environment 0:01  
Checkout code 0:00  
Setup a remote Docker engine 0:02  
echo "This is working" 0:00

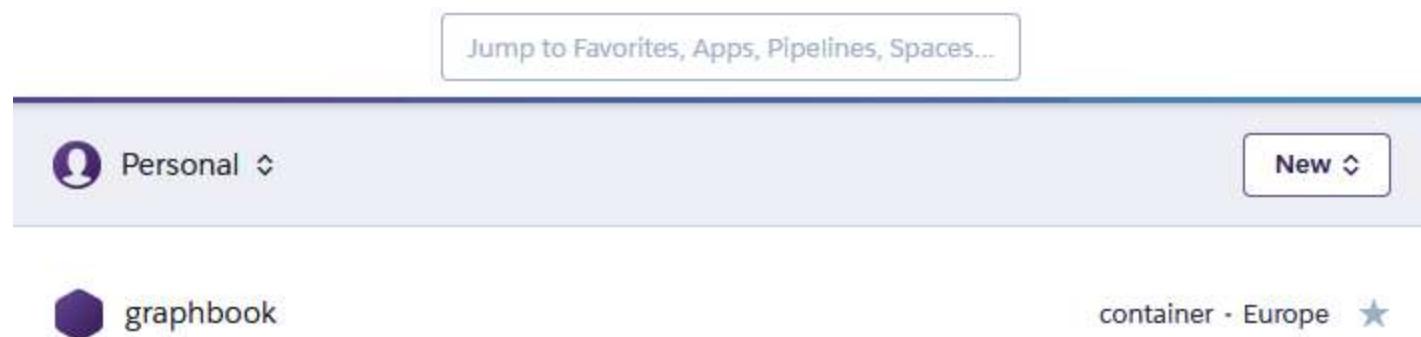


# Deploying applications to Heroku

- CircleCI executes our build steps each time we push a new commit.
- Now, we want to build our Docker image and deploy it automatically to a machine that will serve our application to the public.
- Our database and files are hosted on Amazon Web Services already, so we could also use AWS to serve our application.

# Deploying applications to Heroku

- After logging in, you will be redirected to the apps list for Heroku, as shown in the following screenshot:



# Deploying applications to Heroku

- You will be redirected to the app dashboard, as follows:

The screenshot shows the Heroku app dashboard for the 'graphbook' application. At the top, there's a navigation bar with 'Personal' and 'graphbook'. On the right are 'Open app' and 'More' buttons. Below the navigation is a menu bar with 'Overview', 'Resources', 'Deploy', 'Metrics', 'Activity', 'Access', and 'Settings'. The main content area has several sections: 'Installed add-ons' (\$0.00/month), 'Dyno formation' (\$0.00/month), 'Collaborator activity' (sebigrebe@googlemail.com), and 'Latest activity'. The 'Latest activity' section lists ten deployment events by 'sebigrebe@googlemail.com' from yesterday at 11:40 PM to today at 3:58 PM.

| Event                       | Date                  | Details |
|-----------------------------|-----------------------|---------|
| Deployed web (1a56e108be1b) | Today at 3:58 PM      | v17     |
| Deployed web (3e5ba20e0743) | Today at 1:18 AM      | v16     |
| Deployed web (ec3a7406ca3e) | Today at 12:37 AM     | v15     |
| Deployed web (4e95c643f56b) | Today at 12:03 AM     | v14     |
| Deployed web (b722857867e4) | Yesterday at 11:44 PM | v13     |
| Set  PORT  config var       | Yesterday at 11:41 PM | v12     |
| Set  host  config var       | Yesterday at 11:41 PM | v11     |
| Set  database  config var   | Yesterday at 11:41 PM | v10     |
| Set  password  config var   | Yesterday at 11:40 PM | v9      |

## Config Vars

Config vars change the way your app behaves. In addition to creating your own, some add-ons come with their own.

## Config Vars

[Hide Config Vars](#)

# Deploying applications to Heroku

- If you have Snap installed on your system, you can run the following command:

```
sudo snap install --classic heroku
```

- If this is not the case, manually install the Heroku CLI by using the following command:

```
curl https://cli-assets.heroku.com/install.sh | sh
```

# Deploying applications to Heroku

- Make sure that the installation has worked by verifying the version number, using the heroku command, as follows:

**heroku --version**

# Deploying applications to Heroku

- The Heroku CLI offers a login method. Otherwise, you cannot access your Heroku app and deploy images to it.
- Execute the following command:

`heroku login`

# Deploying applications to Heroku

- Heroku offers a private Docker image registry, like Docker Hub, which was specially made for use with Heroku.
- We will publish our image to this registry, because we can rely on the automatic deployment feature.
- You can deploy images from this repository to your Heroku app automatically.
- To authorize yourself at the registry, you can use the following command:

`heroku container:login`

# Deploying applications to Heroku

- Replace the name graphbook with the name of your app. Run the following command to build the Docker image:

```
docker build -t registry.heroku.com/graphbook/web .
```

# Deploying applications to Heroku

- In the previous tests in this lesson, we did not publish the generated images to any registry.
- Replace the graphbook name with your app's name.
- We will use the docker push command to upload our image to Heroku, as follows:

```
docker push registry.heroku.com/graphbook/web:latest
```

# Deploying applications to Heroku

- Still, nothing has gone live yet.
- There is only one command that we must run to make our application go live, as follows:

```
heroku container:release web --app graphbook
```

# Deploying applications to Heroku

- The beginning of our configuration should be the same as before. Insert it into our config.yml file, as follows:

```
version: 2
jobs:
 build:
 docker:
 - image: circleci/node:10
 steps:
 - checkout
 - setup_remote_docker:
 docker_layer_caching: true
```

# Deploying applications to Heroku

- Before building and deploying our application, we have to ensure that everything works as planned.
- We can use the tests we built in the previous lesson using Mocha.
- Add a second image to the docker section of the preceding code like so:
  - image: tkuchiki/delayed-mysql
  - environment:
    - MySQL\_ALLOW\_EMPTY\_PASSWORD: yes
    - MySQL\_ROOT\_PASSWORD: "
    - MySQL\_DATABASE: graphbook\_test

# Deploying applications to Heroku

- Instead of building and deploying the Docker image straight away, we first have to run our automated test.
- We have to install all dependencies from our package.json file directly within the CircleCI job's container.

Add the following lines to the configuration file:

```
- run:
 name: "Install dependencies"
 command: npm install
```

# Deploying applications to Heroku

- Our test relies on the fact that the back end and front end code is working.
- This includes the fact that our database is also correctly structured with the newest migrations applied.
- We can apply the migrations using Sequelize which we are going to install with the following lines of code::
  - run:

```
name: "Install Sequelize"
command: sudo npm install -g mysql2 sequelize sequelize-cli
```

# Deploying applications to Heroku

- Everything that we need to run our test is now prepared.
- All of the packages are installed, so, we just have to migrate the database and run the tests.
- To make sure that the database has been started though, we have to add one further command, which lets the CircleCI job wait until the database is started.
- Insert the following lines:
  - run:  
name: Wait for DB  
command: dockerize -wait tcp://127.0.0.1:3306 -timeout 120s

# Deploying applications to Heroku

- The next task of our CircleCI workflow is, of course, to apply all migrations to the test database. Add the following lines:

- run:

```
name: "Run migrations for test DB"
command: sequelize db:migrate --migrations-path
src/server/migrations --config src/server/config/index.js --env
production
environment:
 NODE_ENV: production
 password: "
 database: graphbook_test
 username: root
 host: localhost
```

- Now that the database has been updated, we can execute the test.
- Insert the following lines to run our npm run test script with the correct environment variables, as before:

- **run:**

```
name: "Run tests"
command: npm run test
environment:
 NODE_ENV: production
 password: "
 database: graphbook_test
 username: root
 host: localhost
 JWT_SECRET: 1234
```

# Deploying applications to Heroku

- Because we release our container image to our Heroku app, we also need the Heroku CLI installed inside of the deployment job that was started by CircleCI.
- Add the following lines to our config.yml file to install the Heroku CLI:
  - run:

```
name: "Install Heroku CLI"
command: curl https://cli-assets.heroku.com/install.sh | sh
```

# Deploying applications to Heroku

- We must log in to the Heroku Image Registry to push our Docker image after the image has been built.
- Add the following lines of code to our configuration file:
  - run:

```
name: "Login to Docker"
command: docker login -u $HEROKU_LOGIN -p
$HEROKU_API_KEY
registry.heroku.com
```

The screenshot shows the CircleCI web interface for a project named "Sebi55". The left sidebar contains navigation links for JOBS, WORKFLOWS, INSIGHTS, ADD PROJECTS, TEAM, TEST COMMANDS, SETTINGS, NOTIFICATIONS, and PERMISSIONS. The "Environment Variables" link under SETTINGS is highlighted. The main content area displays the "Environment Variables" page for the "Sebi55/HandsOn" job. The page title is "Environment Variables" and it includes a sub-header "Environment Variables for Sebi55/HandsOn". There are "Import Variables" and "Add Variable" buttons. A note states: "Add environment variables to the job. You can add sensitive data (e.g. API keys) here, rather than placing them in the repository." A table lists the current environment variables:

| Name           | Value    | Remove |
|----------------|----------|--------|
| HEROKU_API_KEY | xxxxe406 | X      |
| HEROKU_LOGIN   | xxxx.com | X      |
| database       | xxxxbook | X      |
| host           | xxxx.com | X      |
| password       | xxxx234% | X      |
| username       | xxxxbook | X      |

# Deploying applications to Heroku

- Now, we can start building our Docker image, like we did previously in our manual test.
- Add the following step to the config.yml file:
  - run:

```
name: "Build Docker Image"
command: docker build -t
registry.herokuapp.com/graphbook/web .
```

# Deploying applications to Heroku

- After building the image with the preceding command, we can push the image to the Heroku registry.
- Add the following lines to the configuration file:
  - run:

```
name: "Push Docker Image to Heroku registry"
command: docker push
registry.heroku.com/graphbook/web:latest
```

# Deploying applications to Heroku

- Next, we will migrate the changes to the database structures with the command that we covered in lesson 3, Connecting to The Database:
  - run:

```
name: "Run migrations for production DB"
command: sequelize db:migrate --migrations-path
src/server/migrations --config
src/server/config/index.js --env
production
```

# Deploying applications to Heroku

- Finally, we can deploy our new application, as follows:
  - run:

```
name: "Deploy image to Heroku App"
command: heroku container:release web --app
graphbook
```

- The resulting job should look like as follows:

Jobs » Sebi55 » HandsOn » master » 39 (build)

2.0 C Rerun workflow ⚙

| Test Summary                                       | Queue (00:01) | Artifacts | Configuration | Timing           | Parameters |
|----------------------------------------------------|---------------|-----------|---------------|------------------|------------|
| <a href="#">Set Up Test Summary</a>                |               |           |               |                  |            |
| Show containers: All (1) Successful (1) Failed (0) |               |           |               |                  |            |
| 0 (03:59)                                          |               |           |               | Add Containers + |            |
| TEST                                               |               |           |               |                  |            |
| ▷ Spin up Environment                              |               |           |               |                  | 00:01      |
| ▷ Checkout code                                    |               |           |               |                  | 00:00      |
| ▷ Setup a remote Docker engine                     |               |           |               |                  | 00:02      |
| ▷ Install Heroku CLI                               |               |           |               |                  | 00:01      |
| ▷ Login to Docker                                  |               |           |               |                  | 00:00      |
| ▷ Build Docker Image                               |               |           |               |                  | 03:15      |
| ▷ Push Docker Image to Heroku registry             |               |           |               |                  | 00:27      |
| ▷ Install Sequelize                                |               |           |               |                  | 00:06      |
| ▷ Run migrations for production DB                 |               |           |               |                  | 00:01      |
| ▷ Deploy image to Heroku App                       |               |           |               |                  | 00:01      |

# Deploying applications to Heroku

- If you want to know whether everything is working as expected, you can run the logs function of Heroku CLI on your local machine, as follows:

```
heroku logs --app graphbook
```

# Summary

- In this lesson, you learned how to dockerize your application using a normal Dockerfile and a multi-stage build.
- Furthermore, I have shown you how to set up an exemplary continuous deployment workflow using CircleCI and Heroku.

# "Complete Lab 13"