Lab 8: Routing in React

Currently, we have one screen and one path that our users can visit. When users visit Graphbook, they can log in and see their news feed and chats. Another requirement for a social network is that users have their own profile pages. We will also introduce client-side routing for our React application.

This lab will cover the following topics:

- Installing React Router
- Implementing routes
- Securing routes
- Manual navigation

Lab Solution

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

```
cd ~/Desktop/react-graphql-course/labs/Lab08
```

Installing React Router

In the past, there were a lot of of React Router, with various implementations and features. As we mentioned previously, we are going to install and configure version 4 for this course. If you search for other tutorials on this topic, make sure that you follow the instructions for this version. Otherwise, you might miss some of the changes that React Router has gone through.

To install React Router, simply run [npm] again, as follows:

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

The first step that we will take introduces a simple router to get our current application working, including different paths for all of the screens. The routes that we are going to add are as follows:

- Our posts feed, chats, and the top bar, including the search box, should be accessible under the [/app] route of our application. The path is self-explanatory, but you could also use the [/] root as the main path.
- The login and signup forms should have a separate path, which will be accessible under the root [/] path.
- As we do not have any further screens, we also have to handle a situation in which none of the preceding routes match. In that case, we could display a so-called 404 page, but instead, we are going to redirect to the root path directly.

There is one thing that we have to prepare before continuing. For development, we are using the webpack development server, as this is what we configured in the Lab 1. 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: 'midori',
  historyApiFallback: true,
},
```

The [historyApiFallback] field tells the [devServer] to serve the [index.html] file, not only for the root path, [http://localhost:3000/], but also when it would typically receive a 404 error (such as for paths like [http://localhost:3000/app]). That happens when the path does not match a file or folder that is normal when implementing routing.

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: '/',
},
```

The [publicPath] property tells webpack to prefix the bundle URL to an absolute path, instead of a relative path. When this property is not included, the browser cannot download the bundle when visiting the sub-directories of our application, as we are implementing client-side routing. Let's begin with the first path, and bind the central part of our application, including the news feed, to the [/app] path.

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:

```
import React, { Component } from 'react';
import Feed from './Feed';
import Chats from './Chats';
import Bar from './components/bar';
import CurrentUserQuery from './components/queries/currentUser';

export default class Main extends Component {
  render() {
    return (
      <CurrentUserQuery>
        <Bar changeLoginState={this.props.changeLoginState}/>
        <Feed />
        <Chats />
      </CurrentUserQuery>
    );
  }
}
```

As you might have noticed, the preceding code is pretty much the same as the logged in condition inside the [App.js] file. The only change is that the [changeLoginState] function is taken from the properties, and is not directly a method of the component itself. That is because we split this part out of the [App.js] and put it into a separate file. This improves reusability for other components that we are going to implement.

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>
  )
}
```

If you compare the preceding method with the old one, you can see that we have inserted a [Router] component, instead of directly rendering either the posts feed or the login form. The original components of the [App.js] file are now in the previously created [Main.js] file. Here, we pass the [loggedIn] state variable and the [changeLoginState] function to the [Router] component. Remove the dependencies at the top, such as the [Chats] and [Feed] components, because we won't use them anymore thanks to the new [Main] component. Add the following line to the dependencies of our [App.js] file:

```
import Router from './router';
```

To get this code working, we have to implement our custom [Router] component first. Generally, it is easy to get the routing running with React Router, and you are not required to separate the routing functionality into a separate file, but, that makes it more readable. To do this, create a new [router.js] file in the [client] folder, next to the [App.js] file, with the following content:

```
import React, { Component } from 'react';
import LoginRegisterForm from './components/loginregister';
import Main from './Main';
import { BrowserRouter as Router, Route, Redirect, Switch } from 'react-router-dom';

export default class Routing extends Component {
  render() {
    return (
      <Router>
        <Switch>
          <Route path="/app" component={() => <Main changeLoginState=
          {this.props.changeLoginState}/>}/>
        </Switch>
      </Router>
    )
  }
}
```

At the top, we import all of the dependencies. They include the new [Main] component and the [react-router] package. The following is a quick explanation of all of the components that we are importing from the React Router package:

- [BrowserRouter] (or [Router], for short, as we called it here) is the component that keeps the URL in the address bar in sync with the UI; it handles all of the routing logic.
- The [Switch] component forces the first matching [Route] or [Redirect] to be rendered. We need it to stop rerendering the UI if the user is already in the location to which a redirect is trying to navigate. I generally recommend that you use the [Switch] component, as it catches unforeseeable routing errors.
- [Route] is the component that tries to match the given path to the URL of the browser. If this is the case, the [component] property is rendered. You can see in the preceding code snippet that we are not setting the [Main] component directly as a parameter; instead, we return it from a stateless function. That is required because the [component] property of a [Route] only accepts functions, and not a component object. This solution allows us to pass the [changeLoginState] function to the [Main] component.
- [Redirect] navigates the browser to a given location. The component receives a property called [to], filled by a path starting with a [/]. We are going to use this component in the next section.

The problem with the preceding code is that we are only listening for one route, which is [/app]. If you are not logged in, there will be many errors that are not covered. The best thing to do would be to redirect the user to the root path, where they can log in.

Secured routes

Secured routes represent a to specific paths that are only the is authenticated, or has the correct authorization.

The recommended solution to implement secure routes in React Router version 4 is to write a small, stateless function that conditionally renders either a [Redirect] component or the component specified on the route that requires an authenticated user. We extract the [component] property of the route into the [Component] variable, which is a renderable React object. 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: '/',
        }} />
  )} />
)
```

We call the stateless function [PrivateRoute]. It returns a standard [Route] component, which receives all of the properties initially given to the [PrivateRoute] function. To pass all properties, we use a destructuring assignment with the [...rest] syntax. Using the syntax inside of curly braces on a React component passes all fields of the [rest] object as properties to the component. The [Route] component is only rendered if the given path is matched.

Furthermore, the rendered component is dependent on the user's [loggedIn] state variable, which we have to pass. If the user is logged in, we render the [Component] without any problems. Otherwise, we redirect the user to the root path of our application using the [Redirect] component.

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}/>
```

Notice that we pass the [loggedIn] property by taking the value from the properties of the [Router] itself. It initially receives the [loggedIn] property from the [App] component that we edited previously. The great thing is that the [loggedIn] variable can be updated from the parent [App] component at any time. That means that the [Redirect] component is rendered and the user is automatically navigated to the login form (if the user logs out, for example). We do not have to write separate logic to implement this functionality.

However, we have now created a new problem. We redirect from [/app] to [/] if the user is not logged in, but we do not have any routes set up for the initial ['/'] path. It makes sense for this path to either show the login form or to redirect the user to [/app] if the user is logged in. The pattern for the new component is the same as the preceding code for the [PrivateRoute] component, but in the opposite direction. 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',
        }} />
  )} />
)
```

The preceding condition is inverted to render the original component. If the user is not logged in, the login form is rendered. Otherwise, they will be redirected to the posts feed.

Add the new path to the router, as follows:

```
<LoginRoute exact path="/" component={() => <LoginRegisterForm changeLoginState=
{this.props.changeLoginState}/>} loggedIn={this.props.loggedIn}/>
```

The code looks the same as that of the [PrivateRoute] component, except that we now have a new property, called [exact]. If we pass this property to a route, the browser's location has to match one hundred percent. 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

For the root path, we set [exact] to true, because otherwise the path matches with any browser's location where a [/] is included, as you can see in the preceding table.

ProTip

There are many more configuration options that React Router offers, such as enforcing trailing slashes, case sensitivity, and much more. You can find all of the options and examples in the official documentation at https://reacttraining.com/react-router/web/api/.

Catch-all routes in React Router

Currently, we have two paths set up, which are [/app] and [/]. If a user visits a non-existent path, such as [/test], they will see an empty screen. The solution is to implement a route that matches any path. For simplicity, we redirect the user to the root of our application, but you could easily replace the redirection with a typical 404 page.

Add the following code to the [router.js] file:

```
class NotFound extends Component {
  render() {
    return (
      <Redirect to="/"/> );
  }
}
```

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} />
```

As you can see, we are rendering a simple [Route] in the preceding code. What makes the route special is that we are not passing a [path] property with it. By default, the [path] is completely ignored and the component is rendered every time, except if there is a match with a previous component. That is why we added the route to the bottom of the [Router]. When no route matches, we redirect the user to the login screen in the root path, or, if the user is already logged in, we redirect them to a different screen using the routing logic of the root path. Our [LoginRoute] component handles this last case.

You can test all changes when starting the front end with `npm run client` and the back end with `npm run server`. We have now moved the current state of our application from a standard, single-route application to an application that differentiates the login form and the news feed based on the location of the browser.

Advanced routing with React Router

The primary goal of this lab 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. We are going to create our profile page in the same way, and use the username as the custom path.

We have to implement the following features:

1. We add a new parameterized route for the user profile. The path starts with [/user/] and follows a username.
2. We change the user profile page to send all GraphQL queries, including the [username] route parameter, inside of the [variables] field of the GraphQL request.
3. We edit the [postsFeed] query to filter all posts by the [username] parameter provided.
4. We implement a new GraphQL query on the back end to request a user by their username, in order to show information about the user.
5. When all of the queries are finished, we render a new user profile header component and the posts feed.
6. Finally, we enable navigation between each page without reloading the complete page, but only the changed parts.

Let's start by implementing routing for the profile page in the next section.

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}/>
```

The code contains two new elements, as follows:

- The path that we entered is [/user/:username]. As you can see, the username is prefixed with a colon, telling React Router to pass the value of it to the underlying component being rendered.
- The component that we rendered previously was a stateless function that returned either the [LoginRegisterForm] or the [Main] component. Neither of these received any parameters or properties from React Router. Now, however, it is required that all properties of React Router are transferred to the child component. That includes the username parameter that we just introduced. We use the same destructuring assignment with the [props] object to pass all properties to the [User] component.

Those are all of the changes that we need to accept parameterized paths in React Router. We read out the value inside of the new user page component. Before implementing it, we import the dependency at the top of [router.js] to get the preceding route working:

```
import User from './User';
```

Create the preceding [User.js] file next to the [Main.js] file. Like the [Main] component, we are collecting all of the components that we render on this page. You should stay with this layout, as you can directly see which main parts each page consists of. 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>
    );
  }
}
```

Like before, we use the [CurrentUserQuery] component as a wrapper for the [Bar] component and the [Chats] component. If a user visits the profile of a friend, they see the common application bar at the top. They can access their chats on the right-hand side, like in Facebook. It is one of the many situations in which React and the reusability of components come in handy.

We removed the [Feed] component and replaced it with a new [UserProfile] component. Importantly, the [UserProfile] receives the [username] property. Its value is taken from the properties of the [User] component. These properties were passed over by React Router. If you have a parameter, such as a [username], in the routing path, the value is stored in the [match.params.username] property of the child component. The [match] object generally contains all matching information of React Router.

From this point on, you can implement any custom logic that you want with this value. We will now continue with implementing the profile page.

Follow these steps to build the user's profile page:

1. Create a new folder, called [user], inside the [components] folder.
2. Create a new file, called `index.js`, inside the [user] folder.
3. 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';
```

The first three lines should look familiar. The last two imported files, however, do not exist at the moment, but we are going to change that shortly. The first new file is [UserHeader], which takes care of rendering the avatar image, the name, and information about the user. Logically, we request the data that we will display in this header through a new Apollo query, called [UserQuery].

4. Insert the code for the [UserProfile] component that we are building at the moment beneath the dependencies, as follows:

```
export default class UserProfile extends Component {
  render() {
    const query_variables = { page: 0, limit: 10, username:
    this.props.username };
    return (
      <div className="user">
        <div className="inner">
          <UserQuery variables={{username: this.props.username}}>
            <UserHeader/>
          </UserQuery>
        </div>
        <div className="container">
          <PostsQuery variables={query_variables}>
            <FeedList/>
          </PostsQuery>
        </div>
      </div>
    )
  }
}
```

The [UserProfile] class is not complex. We are running two Apollo queries simultaneously. Both have the [variables] property set. The [PostQuery] receives the regular pagination fields, [page] and [limit], but also the username, which initially came from React Router. This property is also handed over to the [UserQuery], inside of a [variables] object.

5. We should now edit and create the Apollo queries, before programming the profile header component. Open the [postsFeed.js] file from the [queries] folder.

To use the username as input to the GraphQL query we first have to change the query string from the [GET_POSTS] variable. 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) {
```

Add a new line to the [getVariables] method, above the [return] statement:

```
if(typeof variables.username !== typeof undefined) {
  query_variables.username = variables.username;
}
```

If the custom query component receives a [username] property, it is included in the GraphQL request. It is used to filter posts by the specific user that we are viewing.

6. Create a new [userQuery.js] file in the [queries] folder to create the missing query class.
7. Import all of the dependencies and parse the new query schema with [graphl-tag], as follows:

```
import React, { Component } from 'react';
import { Query } from 'react-apollo';
import Loading from '../loading';
import Error from '../error';
import gql from 'graphql-tag';
```

```
const GET_USER = gql`
  query user($username: String!) {
    user(username: $username) {
      id
      email
      username
      avatar
    }
  }
`;
```

The preceding query is nearly the same as the [currentUser] query. We are going to implement the corresponding [user] query later, in our GraphQL API.

8. The component itself is as simple as the ones that we created before. Insert the following code:

```
export default class UserQuery extends Component {
  getVariables() {
    const { variables } = this.props;
    var query_variables = {};
    if(typeof variables.username !== typeof undefined) {
      query_variables.username = variables.username;
    }
    return query_variables;
  }
  render() {
    const { children } = this.props;
    const variables = this.getVariables();
    return(
      <Query query={GET_USER} variables={variables}>
        {({ loading, error, data }) => {
          if (loading) return <Loading />;
          if (error) return <Error><p>{error.message}</p></Error>;
          const { user } = data;
          return React.Children.map(children, function(child){
            return React.cloneElement(child, { user });
          })
        }}
      </Query>
    )
  }
}
```

We set the [query] property and the parameters that are collected by the [getVariables] method to the GraphQL [Query] component. The rest is the same as any other query component that we have written. All child components receive a new property, called [user], which holds all the information about the user, such as their name, their email, and their avatar image. You can extend that later on, but always remember to not publish data that should be private.

9. 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:

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

export default class UserProfileHeader extends Component {
  render() {
    const { avatar, email, username } = this.props.user;
    return (
      <div className="profileHeader">
        <div className="avatar">
          <img src={avatar}/>
        </div>
        <div className="information">
          <p>
            {username}
          </p>
          <p>
            {email}
          </p>
          <p>You can provide further information here and build
          your really personal header component for your users.</p>
        </div>
      </div>
    )
  }
}
```

If you need help getting the CSS styling right, take a look at the official repository for this course. The preceding code only renders the user's data; you could also implement features such as a chat button, which would give the user the option to start messaging with other people. Currently, we have not implemented this feature anywhere, but it is not necessary to explain the principles of React and GraphQL.

We have finished the new front end components, but the [UserProfile] component is still not working. The queries that we are using here either do not accept the username parameter or have not yet been implemented.

The next section will cover which parts of the back end have to be adjusted.

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:

1. We have to add the [username] parameter to the schema of the [postsFeed] query and adjust the resolver function.
2. We have to create the schema and the resolver function for the new [UserQuery] component.

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.

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) {
```

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 };
}
```

We have already covered the basic Sequelize API and how to query associated models by using the [include] parameter in Lab 3. An important point is how we filter posts associated with a user by their username:

1. In the preceding code, we fill the [include] field of the [query] object with the Sequelize model that we want to join. This allows us to filter the associated [Chats] model in the next step.
2. Then, we create a normal [where] object, in which we write the filter condition. If you want to filter the posts by an associated table of users, you can wrap the model and field names that you want to filter by with dollar signs. In our case, we wrap [User.username] with dollar signs, which tells Sequelize to query the [User] model's table and filter by the value of the [username] column.

No adjustments are required for the pagination part. The GraphQL query is now ready. The great thing about the small changes that we have made is that we have just one API function that accepts several parameters, either to display posts on a single user profile, or to display a list of posts like a news feed.

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
```

This query only accepts a [username], but this time it is a required parameter in the new query. Otherwise, the query would make no sense, since we only use it when visiting a user's profile through their username. In the [resolvers.js] file, we will now implement the resolver function using Sequelize:

```
user(root, { username }, context) {
  return User.findOne({
    where: {
      username: username
    }
  });
},
```

In the preceding code, we use the [findOne] method of the [User] model by Sequelize, and search for exactly one user with the username that we provided in the parameter.

We also want to display the email of the user on the user's profile page. Add the email as a valid field on the [User] type in your GraphQL schema with the following line of code:

```
email: String
```

Now that the back end code and the user page are ready, we have to allow the user to navigate to this new page. The next section will cover user navigation using React Router.

Programmatic navigation in React Router

We created a new site with the user profile, but now we have to offer the user a link to get there. The transition between the news feed and the login and registration forms is automated by React Router, but not the transition from the news feed to a profile page. The user decides whether they want to view the profile of the user. React Router has multiple ways to handle navigation. 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';
```

The [Link] component is a tiny wrapper around a regular HTML [a] tag. Apparently, in standard PHP applications or websites, there is no complex logic behind hyperlinks; you click on them, and a new page is loaded from scratch. With React Router or most single-page application JS frameworks, you can add more logic behind hyperlinks. Importantly, instead of completely reloading the pages when navigating between different routes, this now gets handled by React Router. There won't be complete page reloads when navigating; instead, only the required parts are exchanged, and the GraphQL queries are run. This method saves the user expensive bandwidth, because it means that we can avoid downloading all of the HTML, CSS, and image files again.

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>
```

In the rendered HTML, the [img] and [div] tags are surrounded by a common [a] tag, but are handled inside React Router. The [Link] component receives a [to] property, which is the destination of the navigation. 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;
}
```

If you test the changes now, clicking on the username or avatar image, you should notice that the content of the page dynamically changes, but does not entirely reload. A further task would be to copy this approach to the user search list in the application bar and the chats. Currently, the users are displayed, but there is no option to visit their profile pages by clicking on them.

Now, let's take a look at another way to navigate with React Router. If the user has reached a profile page, we want them to navigate back by clicking on a button in the application bar. 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);
```

The [withRouter] HoC gives the [Home] component access to the [history] object of React Router. That is great, because it means that we do not need to pass this object from the top of our React tree down to the [Home] component.

Furthermore, we use the [history] object to navigate the user to the news feed. In the [render] method, we return a button, which, when clicked, runs the [history.push] function. This function adds the new path to the history of the browser and navigates the user to the ['/app'] main page. The good thing is that it works in the same way as the [Link] component, and does not reload the entire website.

There are a few things to do in order to get the button working, as follows:

1. Import the component into the `index.js` file of the [bar] folder, as follows:

```
import Home from './home';
```

2. Then, replace the [Logout] button with the following lines of code:

```
<div className="buttons">
  <Home/>
  <Logout changeLoginState={this.props.changeLoginState}/>
</div>
```

3. I have wrapped the two buttons in a separate [div] tag, so that it is easier to align them correctly. 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;
}
```

Now that we have everything together, the user can visit the profile page and navigate back again. Our final result looks as follows:

We have a big profile header for the user and their posts at the bottom of the window. At the top, you can see the top bar with the currently logged in user.

Remembering the redirect location

Open the [router.js] file. With all of the routing components provided by React Router, we always get access to the properties inside of them. We will make use of this and save the last location that we were redirected from.

In the [PrivateRoute] component, swap out the [Redirect] with the following code:

```
<Redirect to={{
  pathname: '/',
  state: { from: props.location }
}} />
```

Here, I have added the [state] field. The value that it receives comes from the parent [Route] component, which holds the last matched path in the [props.location] field generated by React Router. The path can be a user's profile page or the news feed, since both rely on the [PrivateRoute] component where authentication is required. When the preceding redirect is triggered, you receive the [from] field inside of the router's state.

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',
}} />
```

Here, I have introduced a small condition for the [pathname]. If the [location.state] property is defined, we can rely on the [from] field. Previously, we stored the redirect path in the [PrivateRoute] component. If the [location.state] property does not exist, the user was not visiting a direct hyperlink, but just wanted to log in normally. They will be navigated to the news feed with the [/app] path.

Your application should now be able to handle all routing scenarios, and this should allow your users to view your site comfortably.

Summary

In this lab, 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.

We will continue with this progression by implementing server-side rendering, which needs many adjustments on both the front end and the back end.