React Router Code-Along (CodeGrade)

 (https://github.com/learn-co-curriculum/react-hooks-react-router-code-along-v6)  (https://github.com/learn-co-curriculum/react-hooks-react-router-code-along-v6/issues/new)

Learning Goals

- Add `react-router-dom` to an existing React application.
- Create multiple client-side routes.

Introduction

So far, we have been building our React applications without any navigation — everything in the app has lived at the same URL. Currently, we can make it look like we are changing the page by showing or hiding different components, but none of these changes are dependent on a change in the URL.

Why is this a problem? Well, web addresses are the backbone of the Internet. The web is just a series of links to other pages, after all.

Let's imagine that we have a React application hosted at `www.loveforsoils.com` (not a real website) dedicated to sharing knowledge about soil types (https://en.wikipedia.org/wiki/Soil_type). As developers of this website, we want to provide users with the option to see a list of our favorite soils. Currently, instead of sharing a link to that list, we can only provide a link to our "Love for soils" homepage. Users are then required to interact with our application to see the favorite soil list.

Because our personal opinion on the best soils is so important, we want to allow users to go straight to this list of our favorite soils by using the URL `www.loveforsoils.com/favorites`. Enter **React Router**: a routing library for **React** that allows us to link to specific URLs and conditionally render components depending on which URL is displayed.

As mentioned in the previous lesson, **React Router** enables *client-side routing* which allows us to render different portions of our webpage using the browser's History API (https://reactrouter.com/en/main/start/concepts#history-and-locations) instead of making requests to our

server for a new webpage. With client-side routing, our browser renders a new component, and our client-side JavaScript requests any data we want to display in that component. This is essential for routing in any React application, as we only have a single HTML file to serve — that's the nature of an SPA.

To demonstrate some of the key features of React Router, we have an exercise to code along with. We'll be making a *very* simple social media app — let's dive into it!

Code Along

Setting up our Main Route

To get started, clone down this repo and run `npm install`. Then run `npm run server` to start your `json-server` and `npm start` to open the application in your browser.

If you open up `src/index.js`, you will see that we are currently rendering our `Home` component, which will serve as the homepage of our application. `Home` is rendering a list of user cards displaying existing site users.

```
// index.js
import React from "react";
import ReactDOM from "react-dom/client";
import Home from "./pages/Home";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<Home />);


// Home.js
import { useState, useEffect } from "react"
import UserCard from "../components/UserCard";

function Home() {
  const [users, setUsers] = useState([])
```

```
  useEffect(() =>{
    fetch("http://localhost:4000/users")
      .then(r => r.json())
      .then(data => setUsers(data))
      .catch(error => console.error(error))
  }, [])

  const userList = users.map(user =>{
    return <UserCard key={user.id} user={user}/>
  });

  return (
    <>
      <header>
        {/* place NavBar here */}
      </header>
      <main>
        <h1>Home!</h1>
        {userList}
      </main>
    </>
  );
};

export default Home;
```

To start using React Router, we need to install `react-router-dom`:

```
$ npm install react-router-dom@6
```

> **Note**: make sure to include `@6` at the end of the install command. This walkthrough is designed for version 6 — other versions may have different syntax.

To start implementing routes, we first need to import `createBrowserRouter` and `RouterProvider` from `react-router-dom`:

```
// index.js
import React from "react";
import ReactDOM from "react-dom/client";
// Step 1. Import react-router functions
import { createBrowserRouter, RouterProvider } from "react-router-dom";
import Home from "./pages/Home";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<Home />);
```

`createBrowserRouter` is used to create the router for our application. We'll pass it an array of route objects as its argument. Each route object will have a routing path and a corresponding element to be rendered on that path.

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />
  }
]);
```

The `RouterProvider` provides the router created by `createBrowserRouter` to our application, so it can use React-Router's client-side routing.

```
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<RouterProvider router={router} />);
```

Let's try it! Copy the code below into `src/index.js` and run `npm start` again if you've closed down your application at any point (don't forget to run `npm run server` if you closed down your `json-server` too). Once it is running, point your URL to `http://localhost:3000/`. We should still see the home page, but now it's being rendered using React Router!

```
import React from "react";
import ReactDOM from "react-dom/client";
import { createBrowserRouter, RouterProvider } from "react-router-dom";
import Home from "./pages/Home";

const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />
  }
]);

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<RouterProvider router={router} />);
```

Adding Additional Routes

In the last two steps, we learned how to set up our router using `createBrowserRouter` and `RouterProvider` and add our very first route.

Next, we want to set up routing for `About` and `Login` pages.

First, we'll make two new components within our `pages` directory.

```
// About.js
function About() {
  return (
    <>
      <header>
        {/* Save space for NavBar */}
      </header>
      <main>
        <h1>This is my about component!</h1>
      </main>
```

```
      </>
    );
  };

  export default About;

  // Login.js
  function Login() {
    return (
      <>
        <header>
          {/* Save space for NavBar */}
        </header>
        <main>
          <h1>Login</h1>
          <form>
            <div>
              <label for="username">Username: </label>
              <input id="username" type="text" name="username" placeholder="Username" />
            </div>
            <br/>
            <div>
              <label for="password">Password: </label>
              <input id="password" type="password" name="password" placeholder="Password" />
            </div>
            <br/>
            <button type="submit">Submit</button>
          </form>
        </main>
      </>
    );
  };
```

```
export default Login;
```

Next, we'll import our new pages into our `index.js` file, and add them as routes within `createBrowserRouter`:

```
// ./src/index.js
import React from "react";
import ReactDOM from "react-dom/client";
import { createBrowserRouter, RouterProvider } from "react-router-dom";
import Home from "./pages/Home";
import About from "./pages/About";
import Login from "./pages/Login";

const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />
  },
  {
    path: "/about",
    element: <About />
  },
  {
    path: "/login",
    element: <Login />
  }
])

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<RouterProvider router={router} />);
```

If you go back to the browser you will see that it looks the same — our `Home` component is displaying as before. Now try manually typing in the URL locations for `/about` and `/login`. You should see these new components rendering!

Links and NavLinks

But what good are routes if users don't know how to navigate to them?

React Router provides two components that allow users to navigate through our page using client-side routing: `Link` and `NavLink`. They both have the same base level functionality:

- They render an `<a>` tag to the DOM.
- When the `<a>` tag is clicked, they change the URL and tell React Router to re-render the page, displaying the component that matches the new URL.
- Instead of taking an `href` attribute like normal `<a>` tags, `Link` and `NavLink` take a `to` prop that points to the endpoint the link should take the user to.

`NavLink` acts as a superset of `Link`, adding a default **active** class **when it matches the current URL**. `NavLink` works well for creating a navigation bar, since it allows us to add styling to indicate which link is currently selected. `Link` is a good option for creating standard hyperlinks. For this example, we will be using `NavLink`; we will see examples of using `Link` later on.

For example, this `NavLink` would display "About" and would navigate users to our `/about` page when clicked:

```
<NavLink to="/about">About</NavLink>
```

Let's create a new `NavBar` component in the `components` folder to add these `NavLink`s to our application.

NavBar.js:

```
import { NavLink } from "react-router-dom";
import "./NavBar.css";

/* define the NavBar component */
function NavBar() {
  return (
    <nav>
      <NavLink
```

```
          to="/"
          /* add styling to Navlink */
          className="nav-link"
        >
          Home
        </NavLink>
        <NavLink
          to="/about"
          className="nav-link"
        >
          About
        </NavLink>
        <NavLink
          to="/login"
          className="nav-link"
        >
          Login
        </NavLink>
      </nav>
    );
  };

  export default NavBar;
```

Let's also create a `NavBar.css` file adjacent to our `NavBar` file to add in some styling:

```
.nav-link{
    display: inline-block;
    box-sizing: border-box;
    width: 60px;
    padding: 5px;
    margin: 0 6px 6px;
    background: blue;
```

```
    text-decoration: none;
    text-align: center;
    font-size: min(10vw, 15px) ;
    color: white;
}

.active {
    color: red;
}
```

You can then place your NavBar component in each of your page components to enable easy navigation between different pages in your application!

Ex:

```
// Home.js
import { useState, useEffect } from "react"
import UserCard from "../components/UserCard";
import NavBar from "../components/NavBar";

function Home() {
  const [users, setUsers] = useState([])

  useEffect(() =>{
    fetch("http://localhost:4000/users")
      .then(r => r.json())
      .then(data => setUsers(data))
      .catch(error => console.error(error))
  }, [])

  const userList = users.map(user =>{
    return <UserCard key={user.id} user={user}/>
  });
```

```
  return (
    <>
      <header>
        <NavBar />
      </header>
      <main>
        <h1>Home!</h1>
        {userList}
      </main>
    </>
  );
};

export default Home;
```

You'll also want to add `NavBar` to your `About` and `Login` components.

(Does this seem inefficient? Not very DRY? Don't worry — we'll look at a more efficient way to include a `NavBar` throughout your app in future lessons!)

Load up the browser again and you should see beautiful blue `NavLink` s that load up the desired components.

Dynamic Routes and URL Params

Nice! We're making good progress! But in our social media app, we'll probably want to see our user profiles. Let's create a new file in `pages` called `UserProfile.js` .

Here's a basic setup for that component:

```
// UserProfile.js
import NavBar from "../components/NavBar";

function UserProfile() {
```

```
    return(
      <>
        <header>
          <NavBar />
        </header>
        <main>
          <h1>User Profile</h1>
        </main>
      </>
    );
  };

  export default UserProfile;
```

We can then add this new component to our router:

```
  // index.js
  // ...other import statements
  import UserProfile from "./pages/UserProfile";

  const router = createBrowserRouter([
    {
      path: "/",
      element: <Home />
    },
    {
      path: "/about",
      element: <About />
    },
    {
      path: "/login",
      element: <Login />
    },
```

```
    {
      path: "/profile",
      element: <UserProfile />
    }
  ]);

  // ...render statements
```

Ideally, we want to navigate people to this page when they click on one of our user cards displaying on our homepage.

Let's update our `UserCard` component to use a `Link` from `react-router-dom`:

```
// UserCard.js
import { Link } from "react-router-dom";

function UserCard({user}) {
  return (
    <article>
        <h2>{user.name}</h2>
        <Link to="/profile">View profile</Link>
    </article>
  );
}

export default UserCard;
```

Let's test it out! Click on one of those links — you should be navigated to our `UserProfile` page.

Hang on — we're navigating successfully, but we're not showing any information about a particular user. That won't work!

We still want to use our `UserProfile` page to display information about a user. But we want the *user information* we're displaying to change.

This is where **Dynamic Routes** and **URL Parameters** come in — we can actually use URL routes to pass data!

Let's go back and update our routes to start using dynamic routing and URL parameters:

```
// index.js
// ...import statements

const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />
  },
  {
    path: "/about",
    element: <About />
  },
  {
    path: "/login",
    element: <Login />
  },
  {
    path: "/profile/:id",
    element: <UserProfile />
  }
]);

// ...render statements
```

Notice that we added `/:id` to the end of our `path` for our `UserProfile` route. This notation creates a `URL parameter` — a segment of our URL that can change and that contains data we can use in our components.

By including a URL parameter (or multiple parameters) in a route, we make that route *dynamic* — this single route can actually have many different URLs! For example, the `/profile/1`, `/profile/2`, and `/profile/3` URLs will all lead to the same page. That page will just display different information depending on which URL is used!

Let's update our `UserCard` component to start making use of our dynamic route:

```
// UserCard.js
import {Link} from "react-router-dom";

function UserCard({user}) {
  return (
    <article>
        <h2>{user.name}</h2>
        <p>
          <Link to={`/profile/${user.id}`}>View profile</Link>
        </p>
    </article>
  );
};

export default UserCard;
```

We've used string interpolation to update the `to` prop of our `Link` component to include the `id` of the user corresponding to the link that was clicked. Now, when we click on one of these links, it will take us to the URL `/profile/<some-user-id>` , which will correspond with the `/profile/:id` route we set up in our router.

Try it out! You should still see the `UserProfile` component being rendered as it was before, but the URL should show the `id` of whichever user you clicked on.

Great! But our `UserProfile` component still isn't displaying specific user information.

That's where the last piece of the puzzle comes into play: the `useParams` hook. `useParams` allows us to access the data we've stored in our URL parameters and use it within our components.

Let's start by importing the hook in our UserProfile.js file. Go ahead and import `useEffect` and `useState` from `react` as well - we'll be using both in a moment. We can then invoke the hook inside the component to access the parameters we included in our URL route and save those parameters to a variable. Let's also add a console.log so we can take a look at our new params variable:

```
// UserProfile.js
// ...other import statements
import { useEffect, useState } from "react";
import { useParams } from "react-router-dom";

function UserProfile() {
  const params = useParams();
  console.log(params);
// ...remaining code
```

Go ahead and update UserProfile.js as shown above, then click the link for George Orwell and open the Dev Tools. You should see an object logged to the console that looks like this:

```
{ id: '1' }
```

Note that the key is the parameter we defined in our route, and the value is what appears in the URL.

We can now use the data contained in our params object to access the specific piece of data we want to display! We can interpolate the `id` of the user into a `fetch` request URL and `fetch` that user's specific information from our backend:

```
const [user, setUser] = useState({});
const params = useParams();
const userId = params.id;

useEffect(() =>{
  fetch(`http://localhost:4000/users/${userId}`)
  .then(r => r.json())
  .then(data => setUser(data))
  .catch(error => console.error(error))
}, [userId]);
```

We can now update the JSX to display our specific users name:

```
<h1>{user.name}</h1>
```

We'll also want to add some conditional rendering to make sure our app doesn't error out while it's waiting for our user to be fetched:

```
if(!user.name){
  return <h1>Loading...</h1>;
};
```

Here's what our full UserProfile component looks like now:

```
// UserProfile.js
import { useEffect, useState } from "react";
import { useParams } from "react-router-dom";
import NavBar from "../components/NavBar";

function UserProfile() {
  const [user, setUser] = useState({});
  const params = useParams();
  const userId = params.id;

  useEffect(() =>{
    fetch(`http://localhost:4000/users/${userId}`)
    .then(r => r.json())
    .then(data => setUser(data))
    .catch(error => console.error(error));
  }, [userId]);

  if(!user.name){
    return <h1>Loading...</h1>;
  };
```

```
    return(
      <>
        <header>
          <NavBar />
        </header>
        <main>
          <h1>{user.name}</h1>
        </main>
      </>
    );
  };

  export default UserProfile;
```

With our component updated, we should now see the correct user displaying when we navigate to a specific user's profile page! Nice!

Error Handling

Ok great, we've got most of the basic functionality for client-side routing down! But what if somebody enters a route that doesn't exist? Try entering `http://localhost:3000/florp` into your browser. Yikes! That's an ugly looking error page!

Let's create one more page in our application: ErrorPage.js.

Create this new component within our `pages` folder, then add the following code:

```
import NavBar from "../components/NavBar";
import { useRouteError } from "react-router-dom";

function ErrorPage() {
  const error = useRouteError();
  console.error(error);

  return (
```

```
      <>
        <header>
          <NavBar />
        </header>
        <main>
          <h1>Whoops! Something went wrong!</h1>
        </main>
      </>
    );
}

export default ErrorPage;
```

Note that we're importing the `useRouteError` hook in addition to our `NavBar` component. The `useRouteError` hook allows us to interact with the error itself, including the error status and its message. You can read more about it in the `useRouteError` **documentation** **(https://reactrouter.com/en/main/hooks/use-route-error)** .

Now that we have that, we can add this `ErrorPage` to each of our routes using the `errorElement` field within our route objects:

```
// index.js
// ... other import statements
import ErrorPage from "./pages/ErrorPage";

const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />,
    errorElement: <ErrorPage />
  },
  {
    path: "/about",
    element: <About />,
    errorElement: <ErrorPage />
```

```
    },
    {
      path: "/login",
      element: <Login />,
      errorElement: <ErrorPage />
    },
    {
      path: "/profile/:id",
      element: <UserProfile />,
      errorElement: <ErrorPage />
    }
]);

// ...render statements
```

The `errorElement` can handle more than just bad URLs — it will redirect your app toward the provided Error component should any error occur within your main UI component! For that reason, we'll want to make sure each of our routes has an appropriate `errorElement`.

> **Note**: Applications that are created using `create-react-app` have a built-in React Error Overlay used in development mode. If your page generates an error during development, you will still see the React Error Overlay over your browser page, even with the errorElement included. You can see the errorElement by closing the Error Overlay.

Separation of Concerns

By this point, everything should be functional. (If it's not, try carefully reviewing any errors you're receiving and double checking your code against the example code.) But, we could make an *organizational* improvement.

Take a look at our `index.js` file. It's getting pretty long and messy! Instead of including all of this routing logic within our `index.js` file, let's extract some of it out into a separate file, `routes.js`. This file has already been created for you, but you can create it yourself in future projects.

Let's move our array of route objects into this `routes.js` file, and save it in a variable called `routes`. We can then make our `routes` variable the default export for the file. Don't forget to import all of the necessary components as well!

```
// routes.js
import Home from "./pages/Home";
import About from "./pages/About";
import Login from "./pages/Login";
import UserProfile from "./pages/UserProfile";
import ErrorPage from "./pages/ErrorPage";

const routes = [
  {
    path: "/",
    element: <Home />,
    errorElement: <ErrorPage />
  },
  {
    path: "/about",
    element: <About />,
    errorElement: <ErrorPage />
  },
  {
    path: "/login",
    element: <Login />,
    errorElement: <ErrorPage />
  },
  {
    path: "/profile/:id",
    element: <UserProfile />,
    errorElement: <ErrorPage />
  }
];
```

```
export default routes;
```

Now we just need to make sure we import our `routes` variable back into our `index.js` file and pass it into `createBrowserRouter`:

```
// index.js
import React from "react";
import ReactDOM from "react-dom/client";
import { createBrowserRouter, RouterProvider } from "react-router-dom";
import routes from "./routes.js";

const router = createBrowserRouter(routes);

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<RouterProvider router={router} />);
```

There! Much cleaner! This will also make it easier to run tests on your routes, as your routing configuration has been separated from the rendering logic of your app. This is one of many reasons that separation of concerns is so widely used and so helpful.

Conclusion

You've now seen all the core functionality of React Router required for client-side routing. We've met the requirements so that our app can:

- Conditionally render a different component based on the URL (using `createBrowserRouter` and `RouterProvider`).
- Change the URL using JavaScript, without making a GET request and reloading the HTML document (using the `<Link>` or `<NavLink>` components).

In the coming lessons, we'll explore more of the advanced functionality provided by React Router. If you have the time, you should definitely look at the **React Router docs (https://reactrouter.com/en/main)** — especially the examples — to dive deeper into React Router's many features and get a better sense of how to use it in an application.

Resources

- **React Router docs (https://reactrouter.com/en/main)**

This tool needs to be loaded in a new browser window

Load React Router Code-Along (CodeGrade) in a new window