Nested Routing Code-Along (CodeGrade)

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

Learning Goals

- Create nested routes using `children` and `Outlet`.
- DRY up code with nested routing.
- Pass data to nested route components using `useOutletContext`.

Introduction

In this code-along, we're going to keep working with our Social Media application we made in the previous code-along. However, we want to make some updates!

First of all, we don't want to have to include our `NavBar` component in every page level component — that wasn't very DRY! We also included the same `ErrorPage` on every one of our components — we'll fix that too.

Second of all, we don't want to navigate to a brand new web page to view a specific user. Instead, we want that user to display on the same page as the list of users! But we do still want each user to have their own URL so that we can share links to specific users if we want to.

All of this can be done using **Nested Routing**. Nested Routing allows us to re-render specific *pieces* of a webpage when a user navigates to a new route, rather than re-rendering the entire page. This can be great for developers, as it allows easy reuse of certain components, and also for users, as it can make navigating a website smoother and easier.

To add Nested Routing to our application, we'll need to use a few other things from `react-router-dom`: the `children` attribute on our route objects, the `Outlet` component, and the `useOutletContext` hook. Let's dive into each of those in turn!

Adding a Parent Component

In our last code-along, you might have noticed that we didn't have an `App` component in our list of components. In fact, there was no single parent component to our whole application! Instead, we just had a bunch of parallel components, each of which was rendering on its own route.

While this parallel approach definitely works, and might be the right decision depending on the app you're building, it has some drawbacks. As mentioned above, we had some code that wasn't very DRY — we used the `NavBar` component in every one of our page views, and gave each of our routes the same exact `errorElement`.

Moreover, the only way we could have declared global state for our application would have been through creating our own `contextProvider` with the `useContext` hook. While this is, once again, a perfectly reasonable approach, it can be nice to have a parent component that can instantiate and pass down global application state when your app first loads.

> **Note**: We could have also used a more advanced feature of `react-router` called `loaders`, which allow you to request data for a page as it loads. This is an incredibly powerful and useful feature of `react-router`, but it takes a fair bit of overhead to implement. To read more about loaders, **check out the documentation (https://reactrouter.com/en/main/route/loader)**.

There are many different ways to solve these problems, and the best solution will often depend on what you're trying to build. As a beginner, it's best to learn a variety of design patterns, so you can intelligently apply the right one to your own unique situation!

Okay, enough theorizing — let's get to actually creating this parent App component. We will pick up where we left off with the last code-along, but note that we've already added the `App.js` file for you.

If you haven't already, go ahead and fork and clone the repo for this code-along. Then run `npm install` to install the dependencies, `npm run server` to start your `json-server`, and `npm start` to open the application in the browser.

Rendering Nested Routes as "children"

`react-router-dom` gives us a variety of options we can include in our route objects; so far, we've covered `path`, `element`, and `errorElement`. Another option, `children`, is how we can tell a route that it has *nested routes*.

Go ahead and update our `routes.js` file to include the following code. This will render each of our page-level components as a nested route of our `/` path and our `App` component:

```
// routes.js
import App from "./App";
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: <App />,
        errorElement: <ErrorPage />,
        children: [
             {
                path: "/",
                element: <Home />
            },
            {
                path: "/about",
                element: <About />
            },
            {
                path: "/login",
                element: <Login />
            },
            {
                path: "/profile/:id",
                element: <UserProfile />
            }
```

```
        ]
    }
];

export default routes;
```

Let's walk through all of the changes we've made in the `routes.js` file.

First, we imported the `App` component and added it as the parent component in our routes array.

Second, by entering our different route objects as an array associated with our `App` route's `children` key, we've set them up to render *inside* of our `App` component. That means that if we navigate to any of these *nested routes* — such as `/login`, for example — our `App` component will render with our `Login` component as a child component.

Note that it's okay for our `Home` component to have the same path as our parent `App` component. All child route paths must *start with* their parent's route path, and one of them (but only one) can *exactly match* its parent's route path.

Third, now that all of our routes are children of `App`, we can just include our `errorElement` on `App` — any errors that occur in one of our nested routes will "bubble up" to the parent route, which will render our `ErrorPage`. Much DRYer!

Alternatively, you can render unique `errorElement`s for each route, if you want to create different error handling pages for different routes.

There's one more simplification we can make, this time to the `App.js` file. Since `App` now renders no matter what URL we visit, we can just include our `NavBar` component directly within our `App`, rather than dropping it into every page-level component:

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

function App(){
    return(
        <>
            <header>
                <NavBar />
            </header>
```

```
            </>
        );
    };
```

Much easier! And, if we create a new page for our website, we don't have to remember to include the `NavBar` component within that new page.

Remember to remove the `header` containing the `NavBar` from the `Home` component after adding this code to `App`. We have already removed it from the other pages for you.

Using `react-router-dom`'s Outlet Component

If you've opened the code up in your browser, you might have noticed that our app still isn't working correctly — visiting the child routes doesn't actually render the pages we want.

That's because there is still one tool we need to implement from `react-router-dom` in order to get our nested routes up and running: the `Outlet` component.

An `Outlet` component is included within a component that has nested routes. It basically serves as a signal to that parent component that it will render various different components as its children, depending on what route a user visits. The `Outlet` component works in conjunction with the `router` to determine which component should be rendered based on the current route.

Including it in a component is pretty straightforward:

```
// App.js
import { Outlet } from "react-router-dom";
import NavBar from "./components/NavBar";

function App(){
    return(
        <>
            <header>
                <NavBar />
```

```
            </header>
            <Outlet />
        </>
    );
};
```

And boom! We have nested routing!

Practice

Ok, let's try setting up another nested route. As mentioned in our introduction, we want to view a specific user profile while still viewing the list of all our available users. We can implement this feature by making our `UserProfile` component a *nested route* within our `Home` component.

Let's update our `routes.js` file to make that change!

```
// routes.js
// ...import statements

const routes = [
    {
        path: "/",
        element: <App />,
        errorElement: <ErrorPage />,
        children: [
            {
                path: "/",
                element: <Home />,
                children: [
                    {
                        path: "/profile/:id",
                        element: <UserProfile />
                    }
                ]
```

```
            },
            {
                path: "/about",
                element: <About />
            },
            {
                path: "/login",
                element: <Login />
            }
        ]
    }
];

// ...export statement
```

We'll need to also make sure we update our `Home` component to use the `Outlet` component from `react-router-dom`.

```
// Home.js
import { useState, useEffect } from "react";
import { Outlet } from "react-router-dom";
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 (
            <main>
                <h1>Home!</h1>
                <Outlet />
                {userList}
            </main>
        );
    };

    export default Home;
```

Try navigating to one of our user profile routes. You should see that profile component rendering at the top of the page, above our list of users! (It won't look like much, at present, since we're only rendering a user's name. In a real app, you'll like be displaying more information and will make things look a lot snazzier using CSS.)

Passing Data via useOutletContext

What if we need to pass data from a parent component to a nested route? We're invoking the `Outlet` component within the parent component instead of any of our child components, so we can't pass props in our usual way.

The answer is to create a Context Provider using React's `useContext` hook. Fortunately `react-router-dom` already has this feature built in to `Outlet` components via the `useOutletContext` hook!

We can pass data to our `Outlet` component via a `context` prop, then access it in whatever child component needs the data using the `useOutletContext` hook.

```
<Outlet context={data}/>


const data = useOutletContext();
```

If you want to pass multiple pieces of data, you can pass either an array or an object to the context prop, then destructure it when you invoke the `useOutletContext` hook:

```
<Outlet context={{firstProp: firstData, secondProp: secondData}}/>


const {firstProp, secondProp} = useOutletContext();
```

Let's change our code such that our `users` data is being fetched within our `App` component. We'll then want to pass `users` down via our `Outlet` component's `context` prop, so that we can access it within our nested routes.

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

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

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

    return(
        <>
            <header>
                <NavBar />
            </header>
            <Outlet context={users}/>
        </>
```

```
    );
};
```

Now, within our `Home` component we can use the `useOutletContext` hook to access that piece of data:

```
// Home.js
import { Outlet, useOutletContext } from "react-router-dom";
import UserCard from "../components/UserCard";

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

  return (
      <main>
        <h1>Home!</h1>
        <Outlet />
        {userList}
      </main>
  );
};

export default Home;
```

We should see our list of users rendering just as it was before!

Accessing Outlet Context Within Child Components

Like with any context provider, we can actually access data that we pass to our `Outlet` component's `context` prop within deeply nested components.

Take our `UserCard` component, for example. We don't need our whole array of user data in this component, but for the sake of demonstration we're going to update this component to include the following code:

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

function UserCard({user}) {
    const users = useOutletContext();
    console.log(users);

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

export default UserCard;
```

We should be seeing our array of four users being logged to our browser console.

Instead of passing props from `Home` to `UserCard`, we can just use the `useOutletContext` hook to directly access the data that was originally passed to our `Outlet` component in `App`. This is a very helpful feature if you ever need to pass data to a deeply nested component, and is a great reason to use Context Providers in general with React's `useContext` hook.

However, there is a small hitch that we run into if we have deeply nested routes.

`useOutletContext` and Deeply Nested Routes

If we look at our `routes` in our `routes.js` file, we'll see that we have a deeply nested route:

```
// routes.js
// ...import statements
```

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

// ...export statement
```

Our `Home` route is nested within our `App` route, and our `UserProfile` route is nested within our `Home` route.

If we provide a piece of data to the `Outlet` component within our `App`, and we want to access it within our `UserProfile` component, we'll have to pass that data to the `Outlet` component within our `Home` component first.

Essentially, `useOutletContext` only looks at the *immediate parent* `Outlet` for data. So, if we have one `Outlet` nested within another `Outlet`, we'll need to make sure we pass data to that inner `Outlet` as well:

```
// Home.js
import { Outlet, useOutletContext } from "react-router-dom";
import UserCard from "../components/UserCard";

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

  return (
      <main>
        <h1>Home!</h1>
        <Outlet context={users}/>
        {userList}
      </main>
  );
};

export default Home;
```

Now we can successfully access that data within our `UserProfile` component.

```
// UserProfile.js
import { useParams, useOutletContext } from "react-router-dom";

function UserProfile() {
  const params = useParams();
  const users = useOutletContext();
```

```
  const user = users.find(user => user.id === parseInt(params.id));

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

  return(
      <aside>
        <h1>{user.name}</h1>
      </aside>
  );
};

export default UserProfile;
```

We could have still used a `useEffect` and a `fetch` to load specific user data as we did in the previous code along, but in this case we're finding our specific user using our array of user state passed by `useOutletContex` and the `.find` method, for the sake of demonstration.

> **Note**: We're using an `aside` here instead of `main` because `UserProfile` is now being rendered as a child of `Home`, and `Home` already has a `main` element. HTML best practices dictate that there should be only one `main` element per page view. And, since `UserProfile` only appears on a nested route, we're displaying it in an `aside`, as it will appear alongside the list of users we're rendering.

Conclusion

To review, we learned how to set up Nested Routes using `react-router-dom`, which will allow us to only re-render specific portions of our webpage and include a global parent component for our whole app.

It's not a *requirement* to use Nested Routing in an application, but it's an incredibly powerful tool to have at our disposal.

In the next section, we'll look at two other powerful tools, the `useNavigate` hook and the `Navigate` component, to learn how to add programmatic navigation to our applications.

Resources

- **Nested Routes (https://reactrouter.com/en/main/start/tutorial#nested-routes)**
- **Outlet (https://reactrouter.com/en/main/components/outlet)**
- **useOutletContext (https://reactrouter.com/en/main/hooks/use-outlet-context)**

This tool needs to be loaded in a new browser window

Load Nested Routing Code-Along (CodeGrade) in a new window