COURSEWARE

React Routing

Contents

Overview

In this module, we will look at routing.

Single Page Applications

Many modern applications are based on a single page - content within the page changes depending on user input.

Single page apps increase the speed of web applications.

Only the content that needs to be changed is affected, rather than the whole page having to reload.

`React-Router` is standard routing library to allow this behaviour
From the 'docs':

"React Router is a powerful routing library built on top of React that helps you add new screens and flows to your application incredibly quickly, all while keeping the URL in sync with what's being displayed on the page."

Static and Dynamic Routing

Most frameworks and libraries use static routing.
This involves declaring roots as part of initialisation of app before rendering.

Client side routers need routes to be declared up-front (such as in Angular).

From React Router v4, **dynamic routing** is used - routing takes place as app is rendering.

The whole application is wrapped in a Router, with the links defined within components.

The route is used to define which component should be rendered for the path.

Important note - the route is just a component!

Routers and Routing

The `react-router-dom` package has the <Router> element, with 5 variations on the common low-level interface.

Typically, the high-level <router> element is used in an app, but there are other variations.

- <BrowserRouter> - Uses HTML5 history API to keep UI in sync with URL
- <HashRouter> - Uses hash portion of URL to keep UI in sync
- <MemoryRouter> - Keeps history of URL in memory (useful in testing and React Native)
- <StaticRouter> - Never changes location – useful in server-side rendering

The basic responsibility of a **route** is to render a UI when a location matches route's `path`.

The syntax for a route path is similar to the below;

```
<Route path render_method options />
```

We firstly name the **path**, which specifies the URL.

We then need to consider a render method for the route;

- **component** – specified component will be created and rendered
- **render** – allows for convenient inline rendering and wrapping
- **children** – used to render whether the path matches the location or not – useful for animations

Other options to include here:
`exact` – match only if path matches location.pathname exactly
`strict` – Boolean to represent if match should be made if trailing slash is present
`object` – match path to passed location object's history (current by default)

Route Rendering

component

This is the simplest way to show a Component on a route, and is good for components that don't need any props passed to them

```
<Switch>
    <Route path="/" exact component={App} />
    <Route path="/subContent1" component={SubComponent1} />
    <Route path="/subContent2" component={SubComponent2} />
</Switch>
```

A <Switch> looks through children elements in <Route> and renders the first path match.

If there are multiple routes that you only wish to render one at a time, use a <Switch>!

If not used, interesting and unexpected rendering can occur.

render

If props need to be passed to a component to be rendered on a Route, we should use `render`.

This takes a callback that returns the desired component, and can be mixed with other Routes that have components.

Important to note that there's no performance difference to rendering with `component`.

```
<Switch>
    <Route path="/" render={ props => <App {...props} newProp={newProp} /> } />
    <Route path="/subContent1" component={SubComponent1} />
</Switch>
```

children

This is used to render whether the path matches the location or not.

It works like `render` except it gets called whether there is a match or not.

`children` receives all same route props as component and render methods, and if the route fails to match a URL, match is null.

We can then allow dynamic adjustment of UI based on whether the route matches.

```
<>
    <Route path="/" children={({match, props}) =>
        <div className={match ? "active" : ""}>
            <SomeComponent {...props} /> }
        </div>
    }/>
</>
```

Linking to Routes

We also provide <Link> components to create hyperlinks in the appropriate components.

```
<Link to="/">Home</Link>
```

The `to` attribute can be either a string or an object;

- The object has 4 properties: pathname, search, hash and state
- Other attributes include replace, innerRef and others

Parameterised Routes and Links

Links to parameterised paths can be defined by hard coding or by supplying an expression, perhaps based on some property of an object:

```
<Link to="/content/subContent1"> SubContent 1</Link>

<Link to={`/content/${subContent.id}`}> SubContent </Link>
```

To define a route to a parameterised property, colon notation is used followed by the name of the parameter to use:

```
<Route path="/content/:subContentId" component={SubContent} />
```

React Router v5.1 introduced a number of **hooks** to use with routes.

useParams()

One such hook was the `useParams` hook – imported from `react-router-dom`.

To use route parameters, simply deconstruct the result of the `useParams` function;

```
import { useParams } from `react-router-dom`;

const SubContent = () => {
    const { subContentId } = useParams();
    return (
        <h1>{subContentId}</h1>
    );
};
```

Remember: Deconstructed names must match those in the object being deconstructed.

useHistory()

The `useHistory()` hook gives you access to the `history` instance that you may use to navigate.

```
import {useHistory} from `react-router-dom`;

const HomeButton = () => {
    const history = useHistory();

    function handleClick(){
        history.push("/");
    }

    return(
        <button type="button" onClick={handleClick}> Take me home </button>
    )
}

export default HomeButton;
```

Tutorial

1. Create three pages and place some content inside of them
    1. Home
    2. About
    3. Shop

    ```
    // Home.jsx

    const Home = () => {
      return( <h1> Welcome to the home page </h1> );
    }
    export default Home;
    ```

    ```
    // About.jsx

    const About = () => {
      return( <h1> This is the about Page </h1> );
    }
    export default About;
    ```

    ```
    // Shop.jsx

    const Shop = () => {
      return( <h1> Here is where you can browse our products. </h1>);
    }
    export default Shop;
    ```

2. Create another component called `Nav` and place links inside routing to each of the pages you created previously.

    ```
    import {Link} from 'react-router-dom';

    const Nav = () => {
        <Link to="/">Home</Link>
        <Link to="/about">About Us</Link>
        <Link to="/shop">Shop Now</Link>
    }
    export default Nav;
    ```

3. Place the `<Nav/>` Component in `App.js`.

4. In `App.js` place three `Routes` inside the `<Switch>` Component, linking to each component respectively.

```
function App() {
    return (
        <Router>
        <Nav/>
            <Switch>
            <Route exact path="/">
                <Home/>
            </Route>
            <Route path="/about">
                <About/>
            </Route>
            <Route path="/shop">
                <Shop/>
            </Route>
            </Switch>
        </Router>
    );
}
```

5. Ensure components are correctly imported within `App.js`

```
import Home from './Components/Home';
import About from './Components/About';
import Shop from './Components/Shop';
```

6. Run with `npm start` and observe the output.

Exercises

1. Create a page which contains:
    1. Home page
    2. Users Page
    3. Contact Page
    4. 404 page
2. Create a switch and route for each of the pages in `App.js`
3. Create a URL parameter for `users/:id` - Render their id on the page
4. Implement nested routing within the User's component

▶ Solution