Patterns in React Router

Springboard

Download Demo Code

Goals

- Describe how router URL parameters work
- Understand ***Switch*** and when to use/not use it
- Compare different ways to redirect in React Router
- Learn how to test React Router components

URL Parameters

An Anti-Pattern

```
function App() {
  return (
    <App>
      <Route path="/food/tacos">
        <Food name="tacos" />
      </Route>
      <Route path="/food/salad">
        <Food name="salad" />
      </Route>
      <Route path="/food/sushi">
        <Food name="sushi" />
      </Route>
    </App>
  );
}
```

What's the Problem?

```
<App>
  <Route path="/food/tacos">
    <Food name="tacos" />
  </Route>
  <Route path="/food/salad">
    <Food name="salad" />
  </Route>
  <Route path="/food/sushi">
    <Food name="sushi" />
  </Route>
</App>
```

- Lots of duplication
- What if we want to add more foods?
- Solution: Let's use URL parameters!

A Better Way

```
import React from "react";
import Nav from "./Nav";
import { Route, BrowserRouter } from "react-router-dom";
import Food from "./Food";

function App() {
  return (
    <div className="App">
      <BrowserRouter>
        <Nav />
        <Route exact path="/food/:name">
          <Food />
        </Route>
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Like with Express, we indicate a URL parameter with a colon `:`

Accessing URL Parameters

The ***useParams*** hook stores info on the URL parameters.

```
<Route exact path="/food/:name">
  <Food />
</Route>
```

- given the above code and a url of ***/food/sushi***
  - the useParams hook returns an object
  - the key of the object in this example will be name
  - the value will be sushi

useParams in <Food />

```
import React, { useState, useEffect } from "react";
import { useParams } from "react-router-dom";
import axios from "axios";

const GIPHY_URL = "http://api.giphy.com/v1";
function Food() {
  const { name } = useParams();
  const [src, setSrc] = useState(null);

  useEffect(function loadGiphyImgToSrc() {
    async function fetchGif(searchTerm) {
      let res = await axios.get(`${GIPHY_URL}/gifs/search`, {
        params: { q: searchTerm, api_key: "dc6zaTOxFJmzC" }
      });
      setSrc(res.data.data[0].images.original.url);
    }
    fetchGif(name);
  }, [name]);

  let img = src ? <img src={src} alt={name} /> : null;
  return (
    <div>
      <h1>Here's a pic of {name}.</h1>
      {img}
    </div>
  );
}
```

Multiple URL Parameters

In that example, we only used one URL parameter.

It's possible to have multiple parameters in a single route.

For example, to have food and beverage pairings in route:

```
<Route path="/food/:foodName/drink/:drinkName">
  <Food />
</Route>
```

Here, ***useParams()*** will return an object with two keys: ***foodName*** and ***drinkName***.

The *Switch* Component

Organizing Your Routes

- By default, ***Routes*** match paths *inclusively*.
- If multiple ***Route*** components match, each match will be rendered.
- Using ***exact*** helps, but it's easy to make mistakes with routing logic.

Inclusive Routing: An Example

```
function Routes() {
  return (
    <div>
      <Route exact path="/about"><About /></Route>
      <Route exact path="/contact"><Contact /></Route>
      <Route exact path="/blog/:slug"><Post /></Route>
      <Route exact path="/blog"><BlogHome /></Route>
      <Route exact path="/"><Home /></Route>
    </div>
  );
}
```

- How many routes will match */about*?
- How many routes will match */blog*?
- How many routes will match */blog/unicorns-ftw*?

Exclusive Routing with *Switch*

- Often easier to understand routing when it is *exclusive* (find first match) instead of *inclusive* (find all).
- For exclusive routing: wrap all of ***Route*** components in ***Switch*** component.
- ***Switch*** finds first child ***Route*** that matches and renders only that.

Exclusive Routing: An Example

```
function Routes() {
  return (
    <Switch>
      <Route exact path="/about"><About /></Route>
      <Route exact path="/contact"><Contact /></Route>
      <Route exact path="/blog/:slug"><Post /></Route>
      <Route exact path="/blog"><BlogHome /></Route>
      <Route exact path="/"><Home /></Route>
    </Switch>
  );
}
```

Note: Ordering your routes

Remember that when you wrap your ***Route*** components inside of a ***Switch***, only the first matching ***Route*** will be rendered. This means that the ordering of your routes can be incredibly important. For example, if you put the route with the path of "/" at the top, you'll only ever see the home page:

```
// everything will match the first Route!

<Switch>
  <Route path="/"><Home /></Route>
  <Route path="/about"><About /></Route>
  <Route path="/contact"><Contact /></Route>
  <Route path="/blog/:slug"><BlogPost /></Route>
  <Route path="/blog"><BlogHome /></Route>
</Switch>
```

Similarly, if you put the route to ***/blog*** above the route to ***/blog/:slug***, you'll never hit the route that renders the ***BlogPost*** component.

To remedy these issues, you can either be careful with your routing (paths that will match more things towards the bottom), or you can use the ***exact*** prop inside of some of your ***Route*** components.

Including a 404

```
function Routes() {
  return (
    <Switch>
      <Route exact path="/about"><About /></Route>
      <Route exact path="/contact"><Contact /></Route>
      <Route exact path="/blog/:slug"><Post /></Route>
      <Route exact path="/blog"><BlogHome /></Route>
      <Route exact path="/"><Home /></Route>
      <Route><NotFound /></Route>
    </Switch>
  );
}
```

Note the use of ***exact*** above the catch-all!

Redirects

Client-side Redirects

- With React Router we can mimic the behavior of server-side redirects.
- Useful after certain user actions (e.g. submitting a form)
- Can be used in lieu of having a catch-all 404 component.

How to Redirect

- In React Router, there are two ways to redirect:
  - Using the ***<Redirect>*** component
    - Useful for "you shouldn't have gotten here, go here instead"
  - Calling ***.push*** method on the ***history*** object
    - Useful for "you finished this, now go here"

The *Redirect* Component: An Example

```
function Routes() {
  return (
    <Switch>
      <Route exact path="/about"><About /></Route>
      <Route exact path="/contact"><Contact /></Route>
      <Route exact path="/blog/:slug"><Post /></Route>
      <Route exact path="/blog"><BlogHome /></Route>
      <Route exact path="/"><Home /></Route>
      <Redirect to="/" />
    </Switch>
  );
}
```

Note: Why Bother Redirecting?

At this point, you may be wondering why it's worth redirecting inside of a ***Switch*** statement. After all, what's the difference between closing your switch with these two lines:

```
<Route exact path="/"><Home /></Route>
<Redirect to="/" />
```

and this one line?

```
<Route exact path="/"><Home /></Route>
```

It turns out there's an important distinction. When you use ***Redirect***, a client-side redirect will actually occur, meaning that the URL will change to the value of the ***to*** prop on the redirect. So in the first example, if you went to *localhost:3000/blargh*, the redirect would clean up the URL to just *localhost:3000/*.

In the second example, the lack of an ***exact*** prop means you'd still see the ***Home*** component, but the URL wouldn't get cleaned up. Instead, you'd still see *localhost:3000/blargh* in the URL bar.

The *history* Object

- ***history*** object is a wrapper over the browser's ***history*** API
- You have access to the history object using the ***useHistory*** hook
- The ***history*** object has ***.push(url)***, which adds URL to the session history.
  - So, unlike ***<Redirect>***, hitting back button will return here
- After pushing this new URL, React Router will update the view accordingly.

The *history* Object: An Example

demo/switch-and-redirects/src/Contact.js

```
function Contact() {
  const [email, setEmail] = useState("");
  const history = useHistory();

  function handleChange(evt) {
    setEmail(evt.target.value);
  }

  function storeEmail() {
    console.log("jk, no email storage");
  }

  function handleSubmit(evt) {
    evt.preventDefault();
    storeEmail(email);
    // imperatively redirect to homepage
    history.push("/");
  }
```

demo/switch-and-redirects/src/Contact.js

```
function Contact() {
  return (
    <div>
      <h1>This is the contact page.</h1>
      <p>Enter email to get in touch</p>
      <form onSubmit={handleSubmit}>
        <input
          type="email"
          name="email"
          value={email}
          onChange={handleChange} />
        <button>Submit</button>
      </form>
    </div>
  );
}
```

Note: History API

- All client-side routing libraries use the browser's history API
- History API allows us to manipulate browser history via JS
- Common API methods on ***window.history***:
  - ***.back***: go back one page in history
  - ***.forward***: go forward one page in history
  - ***.go***: go to an arbitrary page in history
  - ***.pushState***: add new entry in history & update URL *without* reloading page.
  - ***.replaceState*** - without adding new entry in history, update URL *without* reloading page.

The function signatures for ***back***, ***forward***, and ***go*** are all relatively straightforward. The first two functions don't take any parameters; the third accepts one parameter, indicating how far back or forward in the session history you'd like to travel.

On the other hand, If you read about the history API on MDN, you'll see that the signature for ***pushState*** and ***replaceState*** are a little… weird. Both accept three parameters: a ***state*** object, a ***title*** and a ***url***. Let's describe these in reverse order:

- ***url*** is simply the new URL you want to put into the URL bar.
- ***title*** parameter is, strangely, ignored by every browser. You can supply a string here if you want, but it does not matter.
- ***state*** parameter is an object that you can potentially access later if the user navigates back to this point in the session history by clicking back or forward. Practically speaking, this isn't something you need to worry about for now.

While these function signatures can definitely be confusing, most times you can ignore the first and second parameters: the most important is the third.

For more on the history API in general, check out MDN on History API

React Router Tips

React Router Tips (Overview)

- Favor ***Route*** child components over other options
- Keep routes up high in the component hierarchy
- Use ***Switch***!
- Avoid nested routes
- Use ***history.push*** in response to user actions (e.g. submitting a form)

Testing React Router

Testing Components with React Router

Components rendered by router are harder to test than regular components:

- components may depend on router hooks we'll have to mock:
  - eg, ***useParams*** hook
- components require the context of a parent router during the test

Mocking Router Context

Consider our ***Nav*** component:

demo/switch-and-redirects/src/Nav.js

```
function Nav() {
  return (
    <ul>
      <li><Link to="/">Home</Link></li>
      <li><Link to="/about">About Us</Link></li>
      <li><Link to="/contact">Contact</Link></li>
      <li><Link to="/blog">Blog</Link></li>
      <li><Link to="/blargh">Broken Link</Link></li>
    </ul>
  );
}
// end
```

Router Context Errors

Problems arise when we `render` the component:

```
it('renders without crashing', function() {
  render(<Nav />);
});
```

```
 RUNS  src/Nav.test.js

Test Suites: 0 of 1 total
Tests:       0 total
Snapshots:   0 total
  console.error node_modules/prop-types/checkPropTypes.js:19
    Warning: Failed context type: The context `router` is marked as required in `Link`, but its value is `undefined`.
        in Link (at Nav.js:8)
        in li (at Nav.js:8)
        in ul (at Nav.js:7)
        in Nav (created by WrapperComponent)
        in WrapperComponent

  console.error node_modules/jest-environment-jsdom/node_modules/jsdom/lib/jsdom/virtual-console.js:29
    Error: Uncaught [Invariant Violation: You should not use <Link> outside a <Router>]
```

MemoryRouter

To avoid *You should not use <Link> outside a <Router>* error, use a mock router, ***MemoryRouter***, which is designed for tests:

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

demo/switch-and-redirects/src/Nav.test.js

```
// full render
it('mounts without crashing', function() {
  const { getByText } = render(
    <MemoryRouter>
      <Nav />
    </MemoryRouter>
  );

  const blogLink = getByText(/Blog/i);
  expect(blogLink).toBeInTheDocument();
});
```

Patterns in React Router

- One of the hardest things about working with React Router is that there aren't strong community standards about the best way to do things with it.
- Here are patterns *we recommend* you adopt for React Router.
- Some of these have been mentioned already—here they are in one place.

Consider a single *Routes.js* file

- Don't spread ***<Route>*** components across multiple files
- You can put all ***<Route>***s directly in your ***App***
- When you have many, it may be overwhelming
  - Having a place for all routing info may be preferable
  - May be easier to debug
  - Make a file, ***Routes.js***, with a ***Routes*** component

Favor exclusive routing with *Switch*

- Routing is easier to understand when matching is exclusive.
  - Use ***Switch*** to wrap your routes.
- Only omit ***Switch*** if you *want* multiple routes to match
  - This is an advanced and unusual pattern.

Avoid nested routes

- Components rendered by a ***Route*** can themselves render ***Route*** components.
- An example of nested routing, and is generally confusing and error prone.
- Unless you need it, don't nest your routes!
  - You'll often end up with spaghetti code and a headache.

Use *history.push* for declarative redirection

- When you know "I want to redirect right now", use ***history.push***
- Save ***Redirect*** for using inside your ***Switch*** for 404-like cases

Looking Ahead

Coming Up

- Redux!
- Integrating React, React Router, and Redux together.
- Full-stack React for fun and profit!