Patterns in React Router

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

```
class App extends Component {
  render() {
    return (
      <App>
        <Route path="/food/tacos"
               render={() => <Food name="tacos" />} />
        <Route path="/food/salad"
               render={() => <Food name="salad" />} />
        <Route path="/food/sushi"
               render={() => <Food name="sushi" />} />
        <Route path="/food/pasta"
               render={() => <Food name="pasta" />} />
      </App>
    );
  }
}
```

😖

What’s the Problem?

```
<App>
  <Route path="/food/tacos"
         render={() => <Food name="tacos" />} />
  <Route path="/food/salad"
         render={() => <Food name="salad" />} />
  <Route path="/food/sushi"
         render={() => <Food name="sushi" />} />
  <Route path="/food/pasta"
         render={() => <Food name="pasta" />} />
</App>
```

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

A Better Way

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

class App extends Component {
  render() {
    return (
      <div className="App">
        <Nav />
        <Route path="/food/:name"
              render={routeProps => <Food {...routeProps} />} />
      </div>
    );
  }
}

export default App;
```

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

Accessing URL Parameters

The ***match*** route prop stores info on URL and associated URL parameters.

- It's an object with these keys:
  - ***path***: same as ***path*** prop passed to ***Route***
  - ***url***: specific URL found in the URL bar
  - ***isExact***: is match between ***url*** & ***path*** exact?
  - ***params***: object of all of the url parameters

example

```
{
  path: "/food/:name",
  url: "/food/tacos",
  isExact: true,
  params: {
    name: "tacos"
  }
}
```

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"
       render={routeProps => <Food {...routeProps} />} />
```

In this case, ***props.match.params*** would be an object with two keys: ***foodName*** and ***drinkName***.

> **Note: Cleaning up our route code**

> If the function you pass in to render starts to become too long, you can always define the function somewhere else to clean up your routing code. For example, both of these code blocks will do the same thing:
>
> *Option 1*
>
> ```
> <Route path="/titanic" render={routeProps => (
>   <Film location={routeProps.location} director="James Cameron"/> )} />
> ```
>
> *Option 2*
>
> ```
> let titanicComponent = routeProps => (
>   <Film location={routeProps.location} director="James Cameron"/> );
>
> // later...
>
> <Route path="/titanic" render={titanicComponent} />
> ```

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

```
class Routes extends Component {
  render() {
    return (
      <div>
        <Route path="/about"
               render={() => <About />} />
        <Route path="/contact"
               render={routeProps => <Contact {...routeProps} />} />
        <Route path="/blog/:slug"
               render={routeProps => <BlogPost {...routeProps} />} />
        <Route path="/blog"
               render={() => <BlogHome />} />
        <Route path="/"
               render={() => <Home />} />
      </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

```
class Routes extends Component {
  render() {
    return (
      <Switch>
        <Route exact path="/about"
               render={() => <About />} />
        <Route exact path="/contact"
               render={routeProps => <Contact {...routeProps} />} />
        <Route exact path="/blog/:slug"
               render={routeProps => <BlogPost {...routeProps} />} />
        <Route exact path="/blog"
               render={() => <BlogHome />} />
        <Route exact path="/"
               render={() => <Home />} />
      </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="/" render={() => <Home />} />
  <Route path="/about" render={() => <About />} />
  <Route path="/contact" render={() => <Contact />} />
  <Route path="/blog/:slug" render={routeProps => <BlogPost {...routeProps} />} />
  <Route path="/blog" render={() => <BlogHome />} />
</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

```
class Routes extends Component {
  render() {
    return (
      <Switch>
        <Route exact path="/about"
               render={() => <About />} />
        <Route exact path="/contact"
               render={routeProps => <Contact {...routeProps} />} />
        <Route exact path="/blog/:slug"
               render={routeProps => <BlogPost {...routeProps} />} />
        <Route exact path="/blog"
               render={() => <BlogHome />} />
        <Route exact path="/"
               render={() => <Home />} />
        <Route render={() => <NotFound />} />
      </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 ***history*** route prop
    - Useful for "you finished this, now go here"

The *Redirect* Component: An Example

```
class Routes extends Component {
  render() {
    return (
      <Switch>
        <Route exact path="/about"
               render={() => <About />} />
        <Route exact path="/contact"
               render={routeProps => <Contact {...routeProps} />} />
        <Route exact path="/blog/:slug"
               render={routeProps => <BlogPost {...routeProps} />} />
        <Route exact path="/blog"
               render={() => <BlogHome />} />
        <Route exact path="/"
```

```
              render={() => <Home />} />
        <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="/"
       render={() => <Home />} />
<Redirect to="/" />
```

and this one line?

```
<Route exact path="/"
       render={() => <Home />} />
```

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* prop

- ***history*** route prop is a wrapper over the browser's ***history*** API
- On ***history*** prop is ***.push(url)***, which add 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* prop: An Example

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

```
class Contact extends Component {
  render() {
    return (
      <div>
        <h1>This is the contact page.</h1>
        <p>To get in touch, enter email.</p>
        <form onSubmit={this.handleSubmit}>
          <input
            type="email"
            name="email"
            value={this.state.email}
            onChange={this.handleChange} />
          <button>Submit</button>
        </form>
      </div>
    );
  }
}
```

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

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

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 <https://developer.mozilla.org/en-US/docs/Web/API/History_API>

Testing React Router

Testing Components with React Router

Components that were rendered by react-router are harder to test than regular components.

There are two issues with components rendered by React Router:

- components sometimes depend on Router Props that we'll have to mock, i.e. the `this.props.match.params` object
- components require the context of a parent router during the test

Mocking Router Props

All we need is a simple POJO that *looks like* our Router props.

demo/url-params-example/src/Food.test.js

```
// mock the route "match" prop
const routeMatch = { params: { name: 'grapefruit' } };

it('renders without crashing', function() {
  shallow(<Food match={routeMatch} />);
});

it('matches snapshot', function() {
  let wrapper = shallow(<Food match={routeMatch} />);
  let serialized = toJson(wrapper);
  expect(serialized).toMatchSnapshot();
});
```

🎉

Mocking Router Context

Consider our ***Nav*** component:

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

```
class Nav extends Component {
  render() {
    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
```

At first glance, the Nav tests normally:

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

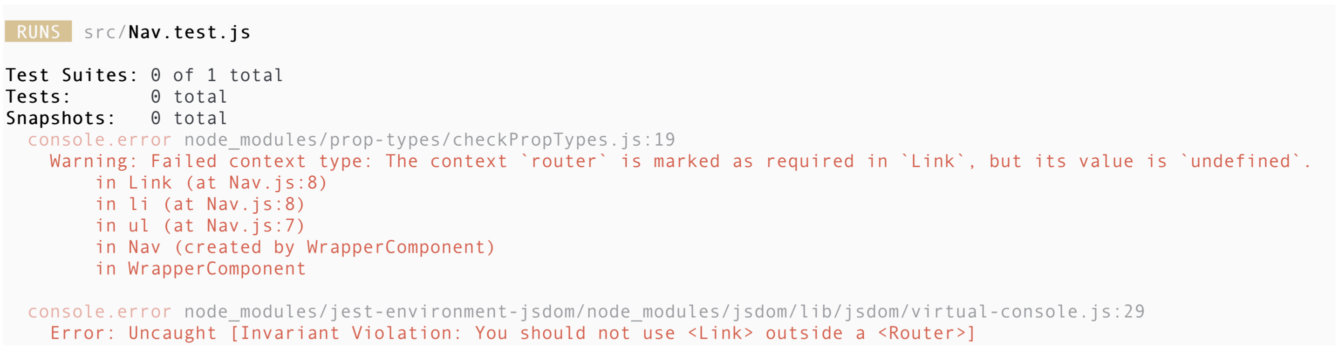
```
// basic tests
it('shallow renders without crashing', function() {
  shallow(<Nav />);
});

it('matches snapshot', function() {
  let wrapper = shallow(<Nav />);
  let serialized = toJson(wrapper);
  expect(serialized).toMatchSnapshot();
});
```

Router Context Errors

Problems arise when we fully `mount` the component:

```
it('mounts without crashing', function() {
  mount(<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>]
```

<_images/test_fail.png>

MemoryRouter

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

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

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

```
// full mount
it('mounts without crashing', function() {
  mount(
    <MemoryRouter>
      <Nav />
    </MemoryRouter>
  );
});
```

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.

Favor ***render*** over ***component***

- There are a different ways to tell ***Route*** "what to render?"
  - ***render=function***, ***component=component***, or ***children***
- ***render*** is flexible and easy to understand

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

BrowserRouter inside of ***index.js***

- You need your ***<BrowserRouter>*** above all of your routes
  - You *could* put it in ***App.js*** or somewhere else high-up in site
- Put it inside your ***index.js***, wrapping your ***<App />***
  - Do this consistently, and you'll never have to look for it

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