Home / Articles /

All About React Router 4

Author
Brad Westfall

14 Comments
Go to Comments

Published
Aug 7, 2017

Updated
Aug 9, 2017

I met Michael Jackson (https://twitter.com/mjackson) for the first time at React Rally 2016, soon after writing an article on React Router 3 (https://css-tricks.com/learning-react-router/) . Michael is one of the principal authors of React Router along with Ryan Florence (https://twitter.com/ryanflorence) . It was exciting to meet someone who built a tool I liked so much, but I was shocked when he said. "Let me show you our ideas React Router 4, it's *way* different!" Truthfully, I didn't understand the new direction and why it needed such big changes. Since the router is such a big part of an application's architecture, this would potentially change some patterns I've grown to love. The idea of these changes gave me anxiety. Considering community cohesiveness and being that React Router plays a huge role in so many React applications, I didn't know how the community would accept the changes.

A few months later, React Router 4 (https://reacttraining.com/react-router/) was released, and I could tell just from the Twitter buzz there was mixed feelings on the drastic re-write. It reminded me of the push-back the first version of React Router had for its progressive concepts. In some ways, earlier versions of React Router resembled our traditional mental model of what an application router "should be" by placing all the routes rules in one place. However, the use of nested JSX routes wasn't accepted by everyone. But just as JSX itself overcame its critics (at least most of them), many came around to believe that a nested JSX router was a pretty cool idea.

So, I learned React Router 4. Admittedly, it was a struggle the first day. The struggle was not with the API, but more so the patterns and strategy for using it. My mental model for using React Router 3 wasn't migrating well to v4. I would have to change how I thought about the relationship between the router and the layout components if I was going to be successful. Eventually, new patterns emerged that made sense to me and I became very happy with the router's new direction. React Router 4 allowed me to do everything I could do with v3, and more.

Also, at first, I was over-complicating the use of v4. Once I gained a new mental model for it, I realized that this new direction is amazing!

My intentions for this article aren't to rehash the already well-written documentation (https://reacttraining.com/react-router/) for React Router 4. I will cover the most common API concepts, but the real focus is on patterns and strategies that I've found to be successful.

Here are some JavaScript concepts you need to be familiar with for this article:

- React (Stateless) Functional Components (https://facebook.github.io/react/docs/components-and-props.html)
- ES2015 Arrow Functions (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) and their "implicit returns"
- ES2015 Destructuring (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)
- ES2015 Template Literals (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals)

If you're the type that prefers jumping right to a working demo, here you go:

A New API and A New Mental Model

Earlier versions of React Router centralized the routing rules into one place, keeping them separate from layout components. Sure, the router could be partitioned and organized into several files, but conceptually the router was a unit, and basically a glorified configuration file.

Perhaps the best way to see how v4 is different is to write a simple two-page app in each version and compare. The example app has just two routes for a home page and a user's page.

Here it is in v3:

JavaScript

```
import { Router, Route, IndexRoute } from 'react-router'

const PrimaryLayout = props => (
  <div className="primary-layout">
    <header>
      Our React Router 3 App
```

```
      </header>
      <main>
        {props.children}
      </main>
    </div>
  )

  const HomePage =() => <div>Home Page</div>
  const UsersPage = () => <div>Users Page</div>

  const App = () => (
    <Router history={browserHistory}>
      <Route path="/" component={PrimaryLayout}>
        <IndexRoute component={HomePage} />
        <Route path="/users" component={UsersPage} />
      </Route>
    </Router>
  )

  render(<App />, document.getElementById('root'))
```

Here are some key concepts in v3 that are not true in v4 anymore:

- The router is centralized to one place.
- Layout and page nesting is derived by the nesting of `<Route>` components.
- Layout and page components are completely naive that they are a part of a router.

React Router 4 does not advocate for a centralized router anymore. Instead, routing rules live within the layout and amongst the UI itself. As an example, here's the same application in v4:

JavaScript

```
  import { BrowserRouter, Route } from 'react-router-dom'

  const PrimaryLayout = () => (
    <div className="primary-layout">
      <header>
        Our React Router 4 App
```

```
      </header>
      <main>
        <Route path="/" exact component={HomePage} />
        <Route path="/users" component={UsersPage} />
      </main>
    </div>
  )

  const HomePage =() => <div>Home Page</div>
  const UsersPage = () => <div>Users Page</div>

  const App = () => (
    <BrowserRouter>
      <PrimaryLayout />
    </BrowserRouter>
  )

  render(<App />, document.getElementById('root'))
```

HEY!

New API Concept: Since our app is meant for the browser, we need to wrap it in `<BrowserRouter>` which comes from v4. Also notice we import from `react-router-dom` now (which means we `npm install react-router-dom` not `react-router`). Hint! It's called `react-router-dom` now because there's also a native version (https://reacttraining.com/react-router/native) .

The first thing that stands out when looking at an app built with React Router v4 is that the "router" seems to be missing. In v3 the router was this giant thing we rendered directly to the DOM which orchestrated our application. Now, besides `<BrowserRouter>`, the first thing we throw into the DOM is our application itself.

Another v3-staple missing from the v4 example is the use of `{props.children}` to nest components. This is because in v4, wherever the `<Route>` component is written is where the sub-component will render to if the route matches.

Inclusive Routing

In the previous example, you may have noticed the `exact` prop. So what's that all about? V3 routing rules were "exclusive" which meant that only one route would win. V4 routes are "inclusive" by default which means more than one `<Route>` can match and render at the same time.

In the previous example, we're trying to render either the `HomePage` or the `UsersPage` depending on the path. If the `exact` prop were removed from the example, both the `HomePage` and `UsersPage` components would have rendered at the same time when visiting `/users` in the browser.

HEY!

To understand the matching logic better, review path-to-regexp (https://www.npmjs.com/package/path-to-regexp) which is what v4 now uses to determine whether routes match the URL.

To demonstrate how inclusive routing is helpful, let's include a `UserMenu` in the header, but only if we're in the user's part of our application:

JavaScript

```
const PrimaryLayout = () => (
  <div className="primary-layout">
    <header>
      Our React Router 4 App
      <Route path="/users" component={UsersMenu} />
    </header>
    <main>
      <Route path="/" exact component={HomePage} />
      <Route path="/users" component={UsersPage} />
    </main>
  </div>
)
```

Now, when the user visits `/users`, both components will render. Something like this was doable in v3 with certain patterns, but it was more difficult. Thanks to v4's inclusive routes, it's now a breeze.

Exclusive Routing

If you need just one route to match in a group, use `<Switch>` to enable exclusive routing:

JavaScript

```
const PrimaryLayout = () => (
  <div className="primary-layout">
    <PrimaryHeader />
    <main>
      <Switch>
        <Route path="/" exact component={HomePage} />
        <Route path="/users/add" component={UserAddPage} />
        <Route path="/users" component={UsersPage} />
        <Redirect to="/" />
      </Switch>
    </main>
  </div>
)
```

Only one of the routes in a given `<Switch>` will render. We still need `exact` on the `HomePage` route though if we're going to list it first. Otherwise the home page route would match when visiting paths like `/users` or `/users/add`. In fact, strategic placement is the name-of-the-game when using an exclusive routing strategy (as it always has been with traditional routers). Notice that we strategically place the routes for `/users/add` before `/users` to ensure the correct matching. Since the path `/users/add` would match for `/users` and `/users/add`, putting the `/users/add` first is best.

Sure, we could put them in any order if we use `exact` in certain ways, but at least we have options.

The `<Redirect>` component will always do a browser-redirect if encountered, but when it's in a `<Switch>` statement, the redirect component only gets rendered if no other routes match first.

To see how `<Redirect>` might be used in a non-switch circumstance, see **Authorized Route** below.

"Index Routes" and "Not Found"

While there is no more `<IndexRoute>` in v4, using `<Route exact>` achieves the same thing. Or if no routes resolved, then use `<Switch>` with `<Redirect>` to redirect to a default page with a valid path (as I did with `HomePage` in the example), or even a not-found page.

Nested Layouts

You're probably starting to anticipate nested sub layouts and how you might achieve them. I didn't think I would struggle with this concept, but I did. React Router v4 gives us a lot of options, which makes it powerful. Options, though, means the freedom to choose strategies that are not ideal. On the surface, nested layouts are trivial, but depending on your choices you may experience friction because of the way you organized the router.

To demonstrate, let's imagine that we want to expand our users section so we have a "browse users" page and a "user profile" page. We also want similar pages for products. Users and products both need sub-layout that are special and unique to each respective section. For example, each might have different navigation tabs. There are a few approaches to solve this, some good and some bad. The first approach is not very good but I want to show you so you don't fall into this trap. The second approach is much better.

For the first, let's modify our `PrimaryLayout` to accommodate the browsing and profile pages for users and products:

JavaScript

```
const PrimaryLayout = props => {
  return (
    <div className="primary-layout">
      <PrimaryHeader />
      <main>
        <Switch>
          <Route path="/" exact component={HomePage} />
```

```
            <Route path="/users" exact component={BrowseUsersPage} />
            <Route path="/users/:userId" component={UserProfilePage} />
            <Route path="/products" exact component={BrowseProductsPage} />
            <Route path="/products/:productId" component={ProductProfilePage}
            <Redirect to="/" />
          </Switch>
        </main>
      </div>
    )
  }
```

While this does technically work, taking a closer look at the two user pages starts to reveal the problem:

JavaScript

```
const BrowseUsersPage = () => (
  <div className="user-sub-layout">
    <aside>
      <UserNav />
    </aside>
    <div className="primary-content">
      <BrowseUserTable />
    </div>
  </div>
)

const UserProfilePage = props => (
  <div className="user-sub-layout">
    <aside>
      <UserNav />
    </aside>
    <div className="primary-content">
      <UserProfile userId={props.match.params.userId} />
    </div>
  </div>
)
```

HEY!

New API Concept: `props.match` is given to any component rendered by `<Route>`. As you can see, the `userId` is provided by `props.match.params`. See more in v4 documentation (https://reacttraining.com/react-router/web/example/url-params). Alternatively, if any component needs access to `props.match` but the component wasn't rendered by a `<Route>` directly, we can use the withRouter() (https://reacttraining.com/react-router/web/api/withRouter) Higher Order Component.

Each user page not only renders its respective content but also has to be concerned with the sub layout itself (and the sub layout is repeated for each). While this example is small and might seem trivial, repeated code can be a problem in a real application. Not to mention, each time a `BrowseUsersPage` or `UserProfilePage` is rendered, it will create a new instance of `UserNav` which means all of its lifecycle methods start over. Had the navigation tabs required initial network traffic, this would cause unnecessary requests — all because of how we decided to use the router.

Here's a different approach which is better:

JavaScript

```
const PrimaryLayout = props => {
  return (
    <div className="primary-layout">
      <PrimaryHeader />
      <main>
        <Switch>
          <Route path="/" exact component={HomePage} />
          <Route path="/users" component={UserSubLayout} />
          <Route path="/products" component={ProductSubLayout} />
          <Redirect to="/" />
        </Switch>
      </main>
    </div>
  )
}
```

Instead of four routes corresponding to each of the user's and product's pages, we have two routes for each section's layout instead.

HEY!

Notice the above routes do not use the `exact` prop anymore because we want `/users` to match any route that starts with `/users` and similarly for products.

With this strategy, it becomes the task of the sub layouts to render additional routes. Here's what the `UserSubLayout` could look like:

JavaScript

```
const UserSubLayout = () => (
  <div className="user-sub-layout">
    <aside>
      <UserNav />
    </aside>
    <div className="primary-content">
      <Switch>
        <Route path="/users" exact component={BrowseUsersPage} />
        <Route path="/users/:userId" component={UserProfilePage} />
      </Switch>
    </div>
  </div>
)
```

The most obvious win in the new strategy is that the layout isn't repeated among all the user pages. It's a double win too because it won't have the same lifecycle problems as with the first example.

One thing to notice is that even though we're deeply nested in our layout structure, the routes still need to identify their full path in order to match. To save yourself the repetitive typing (and in case you decide to change the word "users" to something else), use `props.match.path` instead:

JavaScript

```
const UserSubLayout = props => (
  <div className="user-sub-layout">
```

```
      <aside>
        <UserNav />
      </aside>
      <div className="primary-content">
        <Switch>
          <Route path={props.match.path} exact component={BrowseUsersPage} />
          <Route path={`${props.match.path}/:userId`} component={UserProfileP
        </Switch>
      </div>
    </div>
  )
```

Match

As we've seen so far, `props.match` is useful for knowing what `userId` the profile is rendering and also for writing our routes. The `match` object gives us several properties including `match.params`, `match.path`, `match.url` and several more (https://reacttraining.com/react-router/web/api/match) .

match.path vs match.url

The differences between these two can seem unclear at first. Console logging them can sometimes reveal the same output making their differences even more unclear. For example, both these console logs will output the same value when the browser path is `/users`:

JavaScript

```
const UserSubLayout = ({ match }) => {
  console.log(match.url)   // output: "/users"
  console.log(match.path)  // output: "/users"
  return (
    <div className="user-sub-layout">
      <aside>
        <UserNav />
```

```
        </aside>
        <div className="primary-content">
          <Switch>
            <Route path={match.path} exact component={BrowseUsersPage} />
            <Route path={`${match.path}/:userId`} component={UserProfilePage}
          </Switch>
        </div>
      </div>
    )
  }
```

HEY!

ES2015 Concept: `match` is being destructured (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) at the parameter level of the component function. This means we can type `match.path` instead of `props.match.path`.

While we can't see the difference yet, `match.url` is the actual path in the browser URL and `match.path` is the path written for the router. This is why they are the same, at least so far. However, if we did the same console logs one level deeper in `UserProfilePage` and visit `/users/5` in the browser, `match.url` would be `"/users/5"` and `match.path` would be `"/users/:userId"`.

Which to choose?

If you're going to use one of these to help build your route paths, I urge you to choose `match.path`. Using `match.url` to build route paths will eventually lead a scenario that you don't want. Here's a scenario which happened to me. Inside a component like `UserProfilePage` (which is rendered when the user visits `/users/5`), I rendered sub components like these:

JavaScript

```
const UserComments = ({ match }) => (
  <div>UserId: {match.params.userId}</div>
)
```

```
const UserSettings = ({ match }) => (
  <div>UserId: {match.params.userId}</div>
)

const UserProfilePage = ({ match }) => (
  <div>
    User Profile:
    <Route path={`${match.url}/comments`} component={UserComments} />
    <Route path={`${match.path}/settings`} component={UserSettings} />
  </div>
)
```

To illustrate the problem, I'm rendering two sub components with one route path being made from `match.url` and one from `match.path`. Here's what happens when visiting these pages in the browser:

- Visiting `/users/5/comments` renders "UserId: undefined".
- Visiting `/users/5/settings` renders "UserId: 5".

So why does `match.path` work for helping to build our paths and `match.url` doesn't? The answer lies in the fact that `{${match.url}/comments}` is basically the same thing as if I had hard-coded `{'/users/5/comments'}`. Doing this means the subsequent component won't be able to fill `match.params` correctly because there were no params in the path, only a hardcoded `5`.

It wasn't until later that I saw this part of the documentation (https://reacttraining.com/react-router/core/api/match) and realized how important it was:

> match:
>
> - path - (string) The path pattern used to match. **Useful for building nested `<Route>`s**
> - url - (string) The matched portion of the URL. **Useful for building nested `<Link>`s**

Avoiding Match Collisions

Let's assume the app we're making is a dashboard so we want to be able to add and edit users by visiting `/users/add` and `/users/5/edit`. But with the previous examples, `users/:userId` already points to a `UserProfilePage`. So does that mean that the route with `users/:userId` now needs to point to yet another sub-sub-layout to accomodate editing and the profile? I don't think so. Since both the edit and profile pages share the same user-sub-layout, this strategy works out fine:

JavaScript

```
const UserSubLayout = ({ match }) => (
  <div className="user-sub-layout">
    <aside>
      <UserNav />
    </aside>
    <div className="primary-content">
      <Switch>
        <Route exact path={props.match.path} component={BrowseUsersPage} />
        <Route path={`${match.path}/add`} component={AddUserPage} />
        <Route path={`${match.path}/:userId/edit`} component={EditUserPage}
        <Route path={`${match.path}/:userId`} component={UserProfilePage} /
      </Switch>
    </div>
  </div>
)
```

Notice that the add and edit routes strategically come before the profile route to ensure there the proper matching. Had the profile path been first, visiting `/users/add` would have matched the profile (because "add" would have matched the `:userId`.

Alternatively, we can put the profile route first if we make the path `${match.path}/:userId(\\d+)` which ensures that `:userId` must be a number. Then visiting `/users/add` wouldn't create a conflict. I learned this trick in the docs for path-to-regexp (https://github.com/pillarjs/path-to-regexp#custom-match-parameters).

Authorized Route

It's very common in applications to restrict the user's ability to visit certain routes depending on their login status. Also common is to have a "look-and-feel" for the unauthorized pages (like "log in" and "forgot password") vs the "look-and-feel" for the authorized ones (the main part of the application). To solve each of these needs, consider this main entry point to an application:

JavaScript

```
class App extends React.Component {
  render() {
    return (
      <Provider store={store}>
        <BrowserRouter>
          <Switch>
            <Route path="/auth" component={UnauthorizedLayout} />
            <AuthorizedRoute path="/app" component={PrimaryLayout} />
          </Switch>
        </BrowserRouter>
      </Provider>
    )
  }
}
```

HEY!

Using react-redux (https://github.com/reactjs/react-redux) works very similarly with React Router v4 as it did before, simply wrap `<BrowserRouter>` in `<Provider>` and it's all set.

There are a few takeaways with this approach. The first being that I'm choosing between two top-level layouts depending on which section of the application we're in. Visiting paths like `/auth/login` or `/auth/forgot-password` will utilize the `UnauthorizedLayout` — one that looks appropriate for those contexts. When the user is logged in, we'll ensure all paths have an `/app` prefix which uses `AuthorizedRoute` to determine if the user is logged in or not. If the user tries to visit a page starting with `/app` and they aren't logged in, they will be redirected to the login page.

`AuthorizedRoute` isn't a part of v4 though. I made it myself with the help of v4 docs (https://reacttraining.com/react-router/web/example/auth-workflow) . One amazing new feature

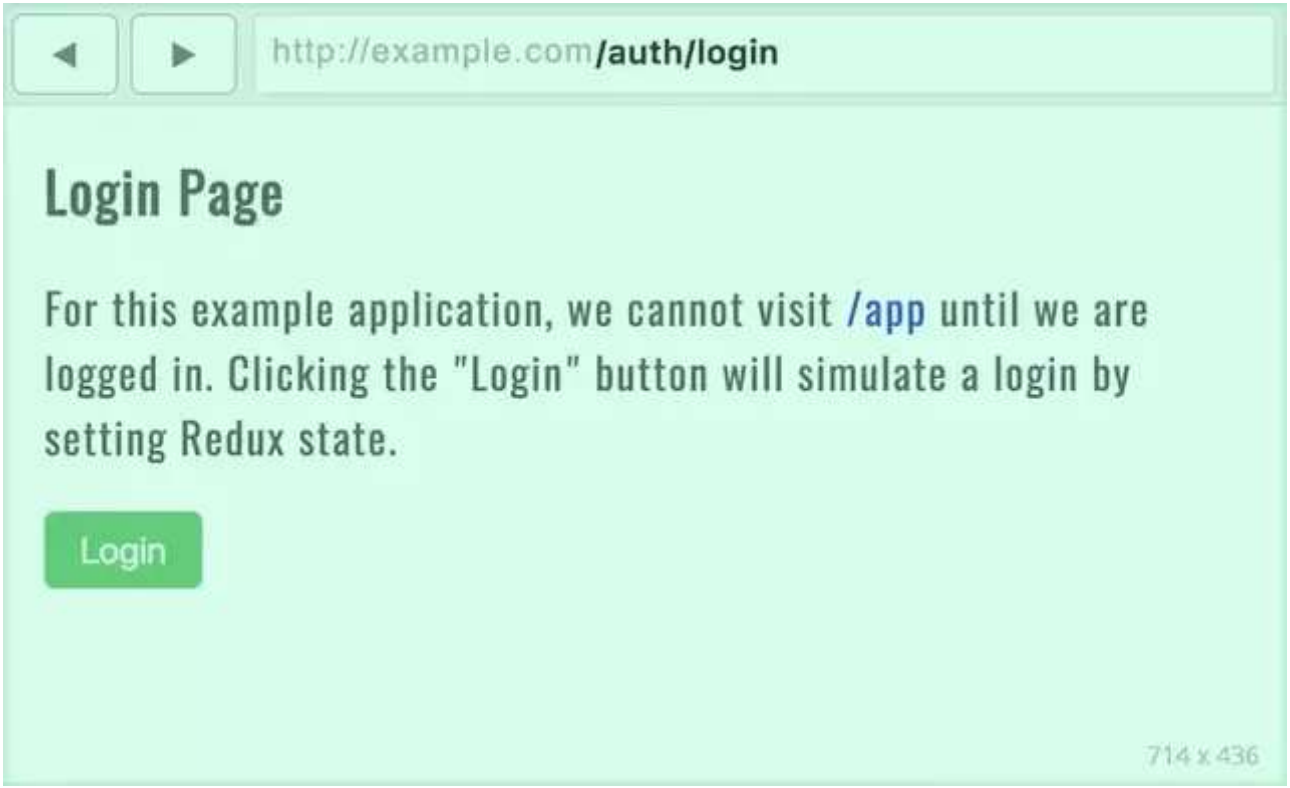
in v4 is the ability to create your own routes for specialized purposes. Instead of passing a `component` prop into `<Route>`, pass a `render` callback instead:

JavaScript

```
class AuthorizedRoute extends React.Component {
  componentWillMount() {
    getLoggedUser()
  }

  render() {
    const { component: Component, pending, logged, ...rest } = this.props
    return (
      <Route {...rest} render={props => {
        if (pending) return <div>Loading...</div>
        return logged
          ? <Component {...this.props} />
          : <Redirect to="/auth/login" />
      }} />
    )
  }
}

const stateToProps = ({ loggedUserState }) => ({
  pending: loggedUserState.pending,
  logged: loggedUserState.logged
})

export default connect(stateToProps)(AuthorizedRoute)
```

While your login strategy might differ from mine, I use a network request to `getLoggedUser()` and plug `pending` and `logged` into Redux state. `pending` just means the request is still in route.

Click here to see a fully working Authentication Example at CodePen (https://codepen.io/bradwestfall/project/editor/XWNWge/?preview_height=50&open_file=src/app.js) .



(https://codepen.io/bradwestfall/project/editor/XWNWge/?preview_height=50&open_file=src/app.js)

Other mentions

There's a lot of other cool aspects React Router v4. To wrap up though, let's be sure to mention a few small things so they don't catch you off guard.

<Link> vs <NavLink>

In v4, there are two ways to integrate an anchor tag with the router: `<Link>` (https://reacttraining.com/react-router/web/api/Link) and `<NavLink>` (https://reacttraining.com/react-router/web/api/NavLink)

`<NavLink>` works the same as `<Link>` but gives you some extra styling abilities depending on if the `<NavLink>` matches the browser's URL. For instance, in the example application (https://codepen.io/bradwestfall/project/editor/XWNWge/#) , there is a `<PrimaryHeader>` component that looks like this:

JavaScript

```
const PrimaryHeader = () => (
  <header className="primary-header">
    <h1>Welcome to our app!</h1>
    <nav>
      <NavLink to="/app" exact activeClassName="active">Home</NavLink>
      <NavLink to="/app/users" activeClassName="active">Users</NavLink>
      <NavLink to="/app/products" activeClassName="active">Products</NavLir
    </nav>
  </header>
)
```

The use of `<NavLink>` allows me to set a class of `active` to whichever link is active. But also, notice that I can use `exact` on these as well. Without `exact` the home page link would be active when visiting `/app/users` because of the inclusive matching strategies of v4. In my personal experiences, `<NavLink>` with the option of `exact` is a lot more stable than the v3 `<Link>` equivalent.

URL Query Strings

There is no longer way to get the query-string of a URL from React Router v4. It seems to me that the decision was made (https://github.com/ReactTraining/react-router/issues/4410) because there is no standard for how to deal with complex query strings. So instead of v4 baking an opinion into the module, they decided to just let the developer choose how to deal with query strings. This is a good thing.

Personally, I use query-string (https://www.npmjs.com/package/query-string) which is made by the always awesome sindresorhus (https://twitter.com/sindresorhus) .

Dynamic Routes

One of the best parts about v4 is that almost everything (including `<Route>`) is just a React component. Routes aren't magical things anymore. We can render them conditionally whenever we want. Imagine an entire section of your application is available to route to when certain conditions are met. When those conditions aren't met, we can remove routes. We can even do

some crazy cool recursive route (https://reacttraining.com/react-router/web/example/recursive-paths) stuff.

React Router 4 is easier because it's Just Components™ (https://youtu.be/Mf0Fy8iHp8k?t=3m22s)