

Title: A Simple React Router v4 Tutorial

Answer:

DEMO - CodePen (<https://codepen.io/pshrmn/pen/YZXZqM>)

1. React Router v4 is a pure React rewrite of the popular React package. Previous versions of React Router used configuration disguised as pseudo-components and could be difficult to understand. With v4, everything is “just components”.

In this tutorial, we will be building a single-page application website for a local sports team. We will go over all of the basics needed to get our site up and routing. This will include:

1. Choosing our router
2. Creating our routes
3. Navigating between routes using links

The Code

Do you just want to see the website in action? Here it is. No fancy styling, just a simple, functional website.

Installation

React Router has been broken into three packages: `react-router`, `react-router-dom`, and `react-router-native`.

You should almost never have to install `react-router` directly. That package provides the core routing components and functions for React Router applications. The other two provide environment specific (browser and `react-native`) components, but they both also re-export all of `react-router`'s exports.

We are building a website (something that will be run in browsers), so we will install `react-router-dom`.

```
npm install --save react-router-dom
```

The Router

When starting a new project, you need to determine which type of router to use. For browser based projects, there are `<BrowserRouter>` and `<HashRouter>` components. The `<BrowserRouter>` should be used when you have a server that will handle dynamic requests (knows how to respond to any possible URI), while the `<HashRouter>` should be used for static websites (where the server can only respond to requests for files that it knows about).

Usually it is preferable to use a `<BrowserRouter>`, but if your website will be hosted on a server that only serves static files, then the `<HashRouter>` is a good solution.

For our project, we will assume that the website will be backed by a dynamic server, so our router component of choice is the `<BrowserRouter>`.

History

Each router creates a `history` (<https://github.com/ReactTraining/history>) object, which it uses to keep track of the current location[1] and re-render the website whenever that changes. The other components provided by React Router rely on having that `history` object available through React's context, so they must be rendered as descendants of a router component. A React Router component that does not have a router as one of its ancestors will fail to work.

Rendering a `<Router>`

Router components only expect to receive a single child element. To work within this limitation, it is useful to create an `<App>` component that renders the rest of your application. Separating your application from the router is also useful for server rendering because you can re-use the `<App>` on the server while switching the router to a `<MemoryRouter>` .

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

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

Now that we have chosen our router, we can start to render our actual application.

The `<App>`

Our application is defined within the `<App>` component. To simplify things, we will split our application into two parts. The `<Header>` component will contain links to navigate throughout the website. The `<Main>` component is where the rest of the content will be rendered.

```
// this component will be rendered by our <___Router>  
const App = () => (  
  <div>  
    <Header />  
    <Main />  
  </div>  
)
```

Note: You can layout your application any way that you would like, but separating routes and navigation makes it easier to show how React Router works for this tutorial.

We will define the content in the `<Main>` component first. This is where we will render our routes.

Routes

The `<Route>` component is the main building block of React Router. Anywhere that you want to only render content based on the location's pathname, you should use a `<Route>` element.

Path

A `<Route>` expects a `path` prop, which is a string that describes the pathname that the route matches — for example, `<Route path='/roster' />` should match a pathname that begins with `/roster` [2]. When the current location's pathname is matched by the `path`, the route will render a React element. When the path does not match, the route will not render anything [3].

```
<Route path='/roster'/>  
// when the pathname is '/', the path does not match  
// when the pathname is '/roster' or '/roster/2', the path matches
```

```
// If you only want to match '/roster', then you need to use  
// the "exact" prop. The following will match '/roster', but not  
// '/roster/2'.  
<Route exact path='/roster'/>
```

```
// You might find yourself adding the exact prop to most routes.  
// In the future (i.e. v5), the exact prop will likely be true by  
// default. For more information on that, you can check out this  
// GitHub issue:  
// https://github.com/ReactTraining/react-router/issues/4958 (https://github.com/ReactTraining,
```

Note: When it comes to matching routes, React Router only cares about the pathname of a location. That means that given the URL:

```
http://www.example.com/my-projects/one?extra=false
```

the only part that React Router attempts to match is `/my-projects/one`.

Matching paths

React Router uses the `path-to-regexp` package to determine if a route element's `path` prop matches the current location. It compiles the `path` string into a regular expression, which will be matched against the location's pathname. `path` strings have more advanced formatting options than will be covered here. You can read about them in the `path-to-regexp` ([https://github.com/pillarjs/path-to-regexp](https://github.com/pillarjs/path-to-regexp/tree/v1.7.0)) documentation (<https://github.com/pillarjs/path-to-regexp/tree/v1.7.0>).

When the route's path matches, a `match` object with the following properties will be created:

- `url` — the matched part of the current location's `pathname`
- `path` — the route's `path`
- `isExact` — `path === pathname`
- `params` — an object containing values from the `pathname` that were captured by `path-to-regexp`

You can use this route tester (<https://pshrmn.github.io/route-tester/#/>) to play around with matching routes to URLs.

Note: Currently, a route's path must be absolute[4].

Creating our routes

`<Route>`s can be created anywhere inside of the router, but often it makes sense to render them in the same place. You can use the `<Switch>` component to group `<Route>`s. The `<Switch>` will iterate over its `children` elements (the routes) and only render the first one that matches the current pathname.

For this website, the paths that we want to match are:

1. / — the homepage
2. /roster — the team's roster
3. /roster/:number — a profile for a player, using the player's number
4. /schedule — the team's schedule of games

In order to match a path in our application, all that we have to do is create a `<Route>` element with the `path` prop we want to match.

```
<Switch>
  <Route exact path="/" component={Home}/>
  {/* both /roster and /roster/:number begin with /roster */}
  <Route path="/roster" component={Roster}/>
  <Route path="/schedule" component={Schedule}/>
</Switch>
```

What does the `<Route>` render?

Routes have three props that can be used to define what should be rendered when the route's `path` matches. Only one should be provided to a `<Route>` element.

1. `component` — A React component. When a route with a `component` prop matches, the route will return a new element whose type is the provided React component (created using `React.createElement`).
2. `render` — A function that returns a React element [5]. It will be called when the path matches. This is similar to `component`, but is useful for inline rendering and passing extra props to the element.
3. `children` — A function that returns a React element. Unlike the prior two props, this will **always** be rendered, regardless of whether the route's path matches the current location.

```
<Route path="/page" component={Page} />
```

```
const extraProps = { color: 'red' }
<Route path="/page" render={(props) => (
  <Page {...props} data={extraProps}/>
)} />
```

```
<Route path="/page" children={(props) => (
  props.match
    ? <Page {...props}/>
    : <EmptyPage {...props}/>
)} />
```

Typically, either the `component` or `render` prop should be used. The `children` prop can be useful occasionally, but typically it is preferable to render nothing when the path does not match. We do not have any extra props to pass to the components, so each of our `<Route>`s will use the `component` prop.

The element rendered by the `<Route>` will be passed a number of props. These will be the `match` object, the current `location` object [6], and the `history` object (the one created by our router) [7].

`<Main>`

Now that we have figured our root route structure, we just need to actually render our routes. For this application, we will render our `<Switch>` and `<Route>` s inside of our `<Main>` component, which will place the HTML generated by a matched route inside of a `<main>` DOM node.

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

```
const Main = () => (  
  <main>  
    <Switch>  
      <Route exact path="/" component={Home}/>  
      <Route path="/roster" component={Roster}/>  
      <Route path="/schedule" component={Schedule}/>  
    </Switch>  
  </main>  
)
```

Note: The route for the homepage includes an `exact` prop. This is used to state that that route should only match when the pathname matches the route's path exactly.

Nested Routes

The player profile route `/roster/:number` is not included in the above `<Switch>` . Instead, it will be rendered by the `<Roster>` component, which is rendered whenever the pathname begins with `/roster` .

Within the `<Roster>` component we will render routes for two paths:

1. `/roster` — This should only be matched when the pathname is exactly `/roster` , so we should also give that route element the `exact` prop.
2. `/roster/:number` — This route uses a path param to capture the part of the pathname that comes after `/roster` .

```
const Roster = () => (  
  <Switch>  
    <Route exact path="/roster" component={FullRoster}/>  
    <Route path="/roster/:number" component={Player}/>  
  </Switch>  
)
```

It can be useful to group routes that share a common prefix in the same component. This allows for simpler parent routes and gives us a place to render content that is common across all routes with the same prefix.

As an example, `<Roster>` could render a title that would be displayed for all routes whose path begins with `/roster` .

```
const Roster = () => (  
  <div>  
    <h2>This is a roster page!</h2>  
    <Switch>  
      <Route exact path='/roster' component={FullRoster}/>  
      <Route path='/roster/:number' component={Player}/>  
    </Switch>  
  </div>  
)
```

Path Params

Sometimes there are variables within a pathname that we want to capture. For example, with our player profile route, we want to capture the player's number. We can do this by adding path params to our route's path string.

The `:number` part of the path `/roster/:number` means that the part of the pathname that comes after `/roster/` will be captured and stored as `match.params.number`. For example, the pathname `/roster/6` will generate a params object :

```
{ number: '6' } // note that the captured value is a string
```

The `<Player>` component can use the `props.match.params` object to determine which player's data should be rendered.

```
// an API that returns a player object  
import PlayerAPI from './PlayerAPI'
```

```
const Player = (props) => {  
  const player = PlayerAPI.get(  
    parseInt(props.match.params.number, 10)  
  )  
  if (!player) {  
    return <div>Sorry, but the player was not found</div>  
  }  
}
```

```
return (  
  <div>  
    <h1>{player.name} ({player.number})</h1>  
    <h2>{player.position}</h2>  
  </div>  
)
```

You can learn more about path params in the `path-to-regexp` (<https://github.com/pillarjs/path-to-regexp#parameters>) documentation (<https://github.com/pillarjs/path-to-regexp#parameters>).

Alongside the `<Player>` component, our website also includes `<FullRoster>`, `<Schedule>`, and `<Home>` components.

```
const FullRoster = () => (  
  <div>  
    <ul>  
      {  
        PlayerAPI.all().map(p => (  
          <li key={p.number}>  
            <Link to={`/roster/${p.number}`}>{p.name}</Link>  
          </li>  
        ))  
      }  
    </ul>  
  </div>  
)
```

```
const Schedule = () => (  
  <div>  
    <ul>  
      <li>6/5 @ Evergreens</li>  
      <li>6/8 vs Kickers</li>  
      <li>6/14 @ United</li>  
    </ul>  
  </div>  
)
```

```
const Home = () => (  
  <div>  
    <h1>Welcome to the Tornadoes Website!</h1>  
  </div>  
)
```

Links

Finally, our application needs a way to navigate between pages. If we were to create links using anchor elements, clicking on them would cause the whole page to reload. React Router provides a `<Link>` component to prevent that from happening. When clicking a `<Link>`, the URL will be updated and the rendered content will change without reloading the page.

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

```
const Header = () => (  
  <header>  
    <nav>  
      <ul>  
        <li><Link to='/>Home</Link></li>  
        <li><Link to='/roster'>Roster</Link></li>  
        <li><Link to='/schedule'>Schedule</Link></li>  
      </ul>  
    </nav>  
  </header>  
)
```

`<Link>`s use the `to` prop to describe the location that they should navigate to. This can either be a string or a location object (containing a combination of `pathname`, `search`, `hash`, and `state` properties). When it is a string, it will be converted to a location object.

```
<Link to={{ pathname: '/roster/7' }}>Player #7</Link>
```

Note: Currently, a link's pathname must be absolute [4].

Get Routing!

Hopefully at this point you are ready to dive into building your own website.

We have covered the most essential components that you will need to build a website (`<BrowserRouter>`, `<Route>`, and `<Link>`). Still, there are a few more components that this did not cover (and props of components that *were* covered). Fortunately, React Router has excellent documentation website (<https://reacttraining.com/react-router/web/guides/quick-start>) that you can use to find more in-depth information about its components. The website also provides a number of working examples with source code.

Edit 7/15: Added CodeSandbox demo

Notes

[1] locations are objects with properties to describe the different parts of a URL:

```
// a basic location object  
{ pathname: '/', search: "", hash: "", key: 'abc123' state: {} }
```

[2] You *can* render a pathless `<Route>`, which will match every location. This can be useful for accessing methods and variables that are stored in the `context`.

[3]—If you use the `children` prop, the route will render even when its `path` does not match the current location.

[4]—There is work being done to add support for relative `<Route>`s and `<Link>`s. Relative `<Link>`s are more complicated than they might initially seem to be because they should be resolved using their parent `match` object, not the current URL.

[5] — This is essentially a stateless functional component. Internally, the big difference between the components passed to `component` and `render` is that `component` will use `React.createElement` to create the element, while `render` will call the component as a function. If you were to define an inline function and pass it to the `component` prop, it would be much slower than using the `render` prop.

```
<Route path='/one' component={One}/>  
// React.createElement(props.component)
```

```
<Route path='/two' render={() => <Two />}/>  
// props.render()
```

[6] — The `<Route>` and `<Switch>` components can both take a `location` prop. This allows them to be matched using a location that is different than the actual location (the current URL).

[7] — They are also passed a `staticContext` prop, but that is only useful when doing server side rendering.

Tags: react