React Router v4: The Complete Guide

September 26, 2017, By **Manjunath M (https://www.sitepoint.com/author/manjunathm/)**

React Router (https://github.com/ReactTraining/react-router) **is the standard routing library for React. When you need to navigate through a React application with multiple views, you'll need a router to manage the URLs. React Router does that, keeping your application UI and the URL in sync.**

This tutorial introduces you to React Router v4 and what you can do with it.

Introduction

React is a popular library for creating single-page applications (SPAs) that are rendered on the client side. An SPA might have multiple **views** (aka **pages**), and unlike the conventional multi-page apps, navigating through these views shouldn't result in the entire page being reloaded. Instead, we want the views to be rendered inline within the current page. The end user, who's accustomed to multi-page apps, expects the following features to be present in an SPA:

> Each view in an application should have a URL that uniquely specifies that view. This is so that the user can bookmark the URL for reference at a later time — e.g. `www.example.com/products`.
> The browser's back and forward button should work as expected.
> The dynamically generated nested views should preferably have a URL of their own too — e.g. `example.com/products/shoes/101`, where 101 is the product id.

Routing is the process of keeping the browser URL in sync with what's being rendered on the page. React Router lets you handle routing **declaratively**. The declarative routing approach allows you to control the data flow in your application, by saying "the route should look like this":

```
<Route path="/about" component={About}/>
```

You can place your `<Route>` component anywhere that you want your route to be rendered. Since `<Route>`, `<Link>` and all the other React Router API that we'll be dealing with are just components, you can easily get used to routing in React.

A note before getting started. There's a common misconception that React Router is an official routing solution developed by Facebook. In reality, it's a third-party library that's widely popular for its design and simplicity. If your requirements are limited to routers for navigation, you could implement a custom router from scratch without much hassle. However, understanding how the basics of React Router will give you better insights into how a router should work.

Overview

This tutorial is divided into different sections. First, we'll be setting up React and React Router using npm. Then we'll jump right into React Router basics. You'll find different code demonstrations of React Router in action. The examples covered in this tutorial include:

1. basic navigational routing
2. nested routing
3. nested routing with path parameters
4. protected routing

All the concepts connected with building these routes will be discussed along the way. The entire code for the project is available on this GitHub repo (https://github.com/blizzerand/react-router-v4-demo). Once you're inside a particular demo directory, run `npm install` to install the dependencies. To serve the application on a development server, run `npm start` and head over to `http://localhost:3000/` to see the demo in action.

Let's get started!

Setting up React Router

I assume you already have a development environment up and running. If not, head over to "Getting Started with React and JSX (https://www.sitepoint.com/getting-started-react-jsx/)". Alternatively, you can use Create React App (https://www.sitepoint.com/create-react-app/) to generate the files required for creating a basic React project. This is the default directory structure generated by Create React App:

```
react-routing-demo-v4
   ├── .gitignore
   ├── package.json
   ├── public
   │   ├── favicon.ico
   │   ├── index.html
   │   └── manifest.json
   ├── README.md
   ├── src
   │   ├── App.css
   │   ├── App.js
   │   ├── App.test.js
   │   ├── index.css
   │   ├── index.js
   │   ├── logo.svg
   │   └── registerServiceWorker.js
   └── yarn.lock
```

The React Router library comprises three packages: `react-router`, `react-router-dom`, and `react-router-native`. `react-router` is the core package for the router, whereas the other two are environment specific. You should use `react-router-dom` if you're building a website, and `react-router-native` if you're on a mobile app development environment using React Native.

Use npm to install `react-router-dom`:

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

React Router Basics

Here's an example of how our routes will look:

```
<Router>
  <Route exact path="/" component={Home}/>
  <Route path="/category" component={Category}/>
  <Route path="/login" component={Login}/>
  <Route path="/products" component={Products}/>
</Router>
```

Router

You need a router component and several route components to set up a basic route as exemplified above. Since we're building a browser-based application, we can use two types of routers from the React Router API:

1. `<BrowserRouter>`
2. `<HashRouter>`

The primary difference between them is evident in the URLs that they create:

```
(https://www.sitepoint.com/)
// <BrowserRouter>
http://example.com/about

// <HashRouter>
http://example.com/#/about
```

The `<BrowserRouter>` is more popular amongst the two because it uses the HTML5 History API to keep track of your router history. The `<HashRouter>`, on the other hand, uses the hash portion of the URL (`window.location.hash`) to remember things. If you intend to support legacy browsers, you should stick with `<HashRouter>`.

Wrap the `<BrowserRouter>` component around the App component.

index.js

```
/* Import statements */
import React from 'react';
import ReactDOM from 'react-dom';

/* App is the entry point to the React code.*/
import App from './App';

/* import BrowserRouter from 'react-router-dom' */
import { BrowserRouter } from 'react-router-dom';

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

Note: A router component can only have a single child element. The child element can be an HTML element — such as div — or a react component.

For the React Router to work, you need to import the relevant API from the `react-router-dom` library. Here I've imported the `BrowserRouter` into `index.js`. I've also imported the `App` component from `App.js`. `App.js`, as you might have guessed, is the entry point to React components.

The above code creates an instance of history for our entire App component. Let me formally introduce you to history.

history

> `history` is a JavaScript library that lets you easily manage session history anywhere JavaScript runs. history provides a minimal API that lets you manage the history stack, navigate, confirm navigation, and persist state between sessions. — React Training docs (https://github.com/ReactTraining/history)

Each router component creates a history object that keeps track of the current location (`history.location`) and also the previous locations in a stack. When the current location changes, the view is re-rendered and you get a sense of navigation. How does the current location change? The history object has methods such as `history.push()` and `history.replace()` to take care of that. `history.push()` is invoked when you click on a `<Link>` component, and `history.replace()` is called when you use `<Redirect>`. Other methods — such as `history.goBack()` and `history.goForward()` — are used to navigate through the history stack by going back or forward a page.

Moving on, we have Links and Routes.

Links and Routes

The `<Route>` component is the most important component in React router. It renders some UI if the current location matches the route's path. Ideally, a `<Route>` component should have a prop named `path`, and if the pathname is matched with the current location, it gets rendered.

(https://www.sitepoint.com/)

The `<Link>` component, on the other hand, is used to navigate between pages. It's comparable to the HTML anchor element. However, using anchor links would result in a browser refresh, which we don't want. So instead, we can use `<Link>` to navigate to a particular URL and have the view re-rendered without a browser refresh.

We've covered everything you need to know to create a basic router. Let's build one.

Demo 1: Basic Routing

src/App.js

(https://www.sitepoint.com/)

```
/* Import statements */
import React, { Component } from 'react';
import { Link, Route, Switch } from 'react-router-dom';

/* Home component */
const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
)

/* Category component */
const Category = () => (
  <div>
    <h2>Category</h2>
  </div>
)

/* Products component */
const Products = () => (
  <div>
    <h2>Products</h2>
  </div>
)

/* App component */
class App extends React.Component {
  render() {
    return (
      <div>
        <nav className="navbar navbar-light">
          <ul className="nav navbar-nav">

           /* Link components are used for linking to other views */
            <li><Link to="/">Homes</Link></li>
            <li><Link to="/category">Category</Link></li>
            <li><Link to="/products">Products</Link></li>

          </ul>
        </nav>
```

```
(https://www.sitepoint.com/)
      // Route components are rendered if the path prop matches the current
URL */
            <Route path="/" component={Home}/>
            <Route path="/category" component={Category}/>
            <Route path="/products" component={Products}/>

       </div>
     )
   }
}
```

We've declared the components for Home, Category and Products inside `App.js`. Although this is okay for now, when the component starts to grow bigger, it's better to have a separate file for each component. As a rule of thumb, I usually create a new file for a component if it occupies more than 10 lines of code. Starting from the second demo, I'll be creating a separate file for components that have grown too big to fit inside the `App.js` file.

Inside the App component, we've written the logic for routing. The `<Route>`'s path is matched with the current location and a component gets rendered. The component that should be rendered is passed in as a second prop.

Here `/` matches both `/` and `/category`. Therefore, both the routes are matched and rendered. How do we avoid that? You should pass the `exact= {true}` props to the router with `path='/'`:

```
<Route exact={true} path="/" component={Home}/>
```

If you want a route to be rendered only if the paths are exactly the same, you should use the exact props.

Nested Routing

To create nested routes, we need to have a better understanding of how `<Route>` works. Let's do that.

`<Route>` has three props that you can you use to define what gets rendered:

component. We've already seen this in action. When the URL is matched, the router creates a React element from the given component using `React.createElement`.
render. This is handy for inline rendering. The render prop expects a function that returns an element when the location matches the route's path.
children. The children prop is similar to render in that it expects a function that returns a React element. However, children gets rendered regardless of whether the path is matched with the location or not.

Path and match

The ***path*** is used to identify the portion of the URL that the router should match. It uses the Path-to-RegExp library to turn a path string into a regular expression. It will then be matched against the current location.

If the router's path and the location are successfully matched, an object is created and we call it the **match** object. The match object carries more information about the URL and the path. This information is accessible through its properties, listed below:

`match.url`. A string that returns the matched portion of the URL. This is particularly useful for building nested `<Link>`s
`match.path`. A string that returns the route's path string — that is, `<Route path="">`. We'll be using this to build nested `<Route>`s.
`match.isExact`. A boolean that returns true if the match was exact (without any trailing characters).
`match.params`. An object containing key/value pairs from the URL parsed by the Path-to-RegExp package.

Now that we know all about `<Route>`s, let's build a router with nested routes.

(https://www.sitepoint.com/)

Switch Component

Before we head for the demo code, I want to introduce you to the `<Switch>` component. When multiple `<Route>`s are used together, all the routes that match are rendered inclusively. Consider this code from demo 1. I've added a new route to demonstrate why `<Switch>` is useful.

```
<Route exact path="/" component={Home}/>
<Route path="/products" component={Products}/>
<Route path="/category" component={Category}/>
<Route path="/:id" render = {()=> (<p> I want this text to show up for all routes
other than '/', '/products' and '/category' </p>)}/>
```

If the URL is `/products`, all the routes that match the location `/products` are rendered. So, the `<Route>` with path `:id` gets rendered along with the `Products` component. This is by design. However, if this is not the behavior you're expecting, you should add the `<Switch>` component to your routes. With `<Switch>`, only the first child `<Route>` that matches the location gets rendered.

Demo 2: nested routing

Earlier on, we created routes for `/`, `/category` and `/products`. What if we wanted a URL of the form `/category/shoes`?

src/App.js

(https://www.sitepoint.com/)

```
import React, { Component } from 'react';
import { Link, Route, Switch } from 'react-router-dom';
import Category from './Category';

class App extends Component {
  render() {

    return (
      <div>
        <nav className="navbar navbar-light">
          <ul className="nav navbar-nav">
            <li><Link to="/">Homes</Link></li>
            <li><Link to="/category">Category</Link></li>
            <li><Link to="/products">Products</Link></li>
          </ul>
       </nav>

    <Switch>
      <Route exact path="/" component={Home}/>
      <Route path="/category" component={Category}/>
       <Route path="/products" component={Products}/>
    </Switch>

    </div>
    );
  }
}
export default App;

/* Code for Home and Products component omitted for brevity */
```

Unlike the earlier version of React Router, in version 4, the nested `<Route>`s should preferably go inside the parent component. That is, the Category component is the parent here, and we'll be declaring the routes for `category/:name` inside the parent component.

src/Category.jsx

(https://www.sitepoint.com/)

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

const Category = ({ match }) => {
return( <div> <ul>
    <li><Link to={`${match.url}/shoes`}>Shoes</Link></li>
    <li><Link to={`${match.url}/boots`}>Boots</Link></li>
    <li><Link to={`${match.url}/footwear`}>Footwear</Link></li>

  </ul>
  <Route path={`${match.path}/:name`} render= {({match}) =>( <div> <h3>
{match.params.name} </h3></div>)}/>
  </div>)
}
export default Category;
```

First, we've declared a couple of links for the nested routes. As previously mentioned, `match.url` will be used for building nested links and `match.path` for nested routes. If you're having trouble understanding the concept of match, `console.log(match)` provides some useful information that might help to clarify it.

```
<Route path={`${match.path}/:name`}
  render= {({match}) =>( <div> <h3> {match.params.name} </h3></div>)}/>
```

This is our first attempt at dynamic routing. Instead of hard-coding the routes, we've used a variable within the pathname. `:name` is a path parameter and catches everything after `category/` until another forward slash is encountered. So, a pathname like `products/running-shoes` will create a `params` object as follows:

```
{
  name: 'running-shoes'
}
```

The captured data should be accessible at `match.params` or `props.match.params` depending on how the props are passed. The other interesting thing is that we've used a `render` prop. `render` props are pretty handy for inline functions that don't require a component of their own.

Demo 3: Nested routing with Path parameters

Let's complicate things a bit more, shall we? A real-world router will have to deal with data and display it dynamically. Assume that we have the product data returned by a server API of the form below.

src/Products.jsx

(https://www.sitepoint.com/)

```
const productData = [
{
  id: 1,
  name: 'NIKE Liteforce Blue Sneakers',
  description: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin
molestie.',
  status: 'Available'

},
{
  id: 2,
  name: 'Stylised Flip Flops and Slippers',
  description: 'Mauris finibus, massa eu tempor volutpat, magna dolor euismod
dolor.',
  status: 'Out of Stock'

},
{
  id: 3,
  name: 'ADIDAS Adispree Running Shoes',
  description: 'Maecenas condimentum porttitor auctor. Maecenas viverra fringilla
felis, eu pretium.',
  status: 'Available'
},
{
  id: 4,
  name: 'ADIDAS Mid Sneakers',
  description: 'Ut hendrerit venenatis lacus, vel lacinia ipsum fermentum vel.
Cras.',
  status: 'Out of Stock'
},

];
```

We need to create routes for the following paths:

- `/products`. This should display a list of products.
- `/products/:productId`. If a product with the `:productId` exists, it should display the product data, and if not, it should display an error message.

src/Products.jsx

```
/* Import statements have been left out for code brevity */

const Products = ({ match }) => {

   const productsData = [
    {
        id: 1,
        name: 'NIKE Liteforce Blue Sneakers',
        description: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Proin molestie.',
        status: 'Available'

    },

    //Rest of the data has been left out for code brevity

];
 /* Create an array of `<li>` items for each product
  var linkList = productsData.map( (product) => {
    return(
      <li>
        <Link to={`${match.url}/${product.id}`}>
          {product.name}
        </Link>
      </li>
      )

    })

  return(
    <div>
        <div>
         <div>
           <h3> Products</h3>
           <ul> {linkList} </ul>
         </div>
        </div>

        <Route path={`${match.url}/:productId`}
            render={ (props) => <Product data= {productsData} {...props} />}/>
        <Route exact path={match.url}
```

```
            render={() => (
                <div>Please select a product.</div>
                )}
            />
        </div>
    )
}
```

First, we created a list of `<Links>`s using the `productsData.id`s and stored it in `linkList`. The route takes a parameter in the path string which corresponds to that of the product id.

```
<Route path={`${match.url}/:productId`}
  render={ (props) => <Product data= {productsData} {...props} />}/>
```

You may have expected `component = { Product }` instead of the inline render function. The problem is that we need to pass down `productsData` to the Product component along with all the existing props. Although there are other ways you can do this, I find this method to be the easiest. `{...props}` uses the ES6's spread syntax (https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator) to pass the whole props object to the component.

Here's the code for Product component.

src/Product.jsx

```
      (https://www.sitepoint.com/)
/* Import statements have been left out for code brevity */

const Product = ({match,data}) => {
  var product= data.find(p => p.id == match.params.productId);
  var productData;

  if(product)
    productData = <div>
      <h3> {product.name} </h3>
      <p>{product.description}</p>
      <hr/>
      <h4>{product.status}</h4>  </div>;
  else
    productData = <h2> Sorry. Product doesnt exist </h2>;

  return (
    <div>
      <div>
         {productData}
      </div>
    </div>
  )
}
```

The `find` method is used to search the array for an object with an id property that equals `match.params.productId`. If the product exists, the `productData` is displayed. If not, a “Product doesn’t exist” message is rendered.

Protecting Routes

For the final demo, we’ll be discussing techniques concerned with protecting routes. So, if someone tries to access `/admin`, they’d be required to log in first. However, there are some things we need to cover before we can protect routes.

Redirect

Like the server-side redirects, `<Redirect>` will replace the current location in the history stack with a new location. The new location is specified by the `to` prop. Here's how we'll be using `<Redirect>`:

```
<Redirect to={{pathname: '/login', state: {from: props.location}}}
```

So, if someone tries to access the `/admin` while logged out, they'll be redirected to the `/login` route. The information about the current location is passed via state, so that if the authentication is successful, the user can be redirected back to the original location. Inside the child component, you can access this information at `this.props.location.state`.

Custom Routes

A custom route is a fancy word for a route nested inside a component. If we need to make a decision whether a route should be rendered or not, writing a custom route is the way to go. Here's the custom route declared among other routes.

src/App.js

```
/* Add the PrivateRoute component to the existing Routes */
<Switch>
  <Route exact path="/" component={Home} data={data}/>
  <Route path="/category" component={Category}/>
  <Route path="/login" component={Login}/>
  <PrivateRoute authed={fakeAuth.isAuthenticated} path='/products' component =
{Products} />
</Switch>
```

`fakeAuth.isAuthenticated` returns true if the user is logged in and false otherwise.

Here's the definition for PrivateRoute:

src/App.js

```
/* PrivateRoute component definition */
const PrivateRoute = ({component: Component, authed, ...rest}) => {
  return (
    <Route
      {...rest}
      render={(props) => authed === true
        ? <Component {...props} />
        : <Redirect to={{pathname: '/login', state: {from: props.location}}} />}
/>
  )
}
```

The route renders the Admin component if the user is logged in. Otherwise, the user is redirected to `/login`. The good thing about this approach is that it is evidently more declarative and `PrivateRoute` is reusable.

Finally, here's the code for the Login component:

src/Login.jsx

```
import React from 'react';
import { Redirect } from 'react-router-dom';

class Login extends React.Component {

  constructor() {
    super();

    this.state = {
      redirectToReferrer: false
    }
    // binding 'this'
    this.login = this.login.bind(this);
  }

  login() {

    fakeAuth.authenticate(() => {
      this.setState({ redirectToReferrer: true })
    })
  }

  render() {
    const { from } = this.props.location.state || { from: { pathname: '/' } }
    const { redirectToReferrer } = this.state;

    if (redirectToReferrer) {
      return (
        <Redirect to={from} />
      )
    }

    return (
      <div>
        <p>You must log in to view the page at {from.pathname}</p>
        <button onClick={this.login}>Log in</button>
      </div>
    )
  }
}
```

```
/* A fake authentication function */
export const fakeAuth = {

  isAuthenticated: false,
  authenticate(cb) {
    this.isAuthenticated = true
     setTimeout(cb, 100)
  },
}
```

The line below demonstrates object destructuring (https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment), which is a part of the ES6 specification.

```
const { from } = this.props.location.state || { from: { pathname: '/' } }
```

Let's fit the puzzle pieces together, shall we? Here's the final demo of the application that we built using React router:

Demo 4: Protecting Routes

Summary

As you've seen in this article, React Router is a powerful library that complements React for building better, declarative routes. Unlike the prior versions of React Router, in v4, everything is "just components". Moreover, the new design pattern perfectly fits into the React way of doing things.

In this tutorial, we learned:

- how to setup and install React Router
- the basics of routing and some essential components such as `<Router>`, `<Route>` and `<Link>`
- how to create a minimal router for navigation and nested routes
- how to build dynamic routes with path parameters

(https://www.sitepoint.com/)

Finally, we learned some advanced routing techniques for creating the final demo for protected routes.


'http

Meet the author

Manjunath M (https://www.sitepoint.com/author/manjunathm/) (https://twitter.com/blizzerand) (https://github.com/blizzerand)

Full-stack JavaScript developer. Rubyist by passion. Amateur musician. Favorite quote? "Being a jack of all trades doesn't mean you're a master at none."

(https://www.sitepoint.com/)