React Router DOM: How to handle routing in web apps

July 28, 2021 · 14 min read

***Editor's note**: This React Router DOM tutorial was last updated on 11 August 2021. It may still contain information that is out of date.*

We've covered React Router extensively, including how to use Hooks alongside and instead of React Router, how to use React Router with Redux, and other advanced use cases. But if you're just starting out with React Router, all that might be too much to wrap your head around.

Not to worry. In this post, I'll get you started with the basics of the web version, React Router DOM. We'll cover the general concept of a router and how to set up and install React Router. We'll also review the essential components of the framework and demonstrate how to build routes with parameters, like `/messages/10`.

Here's what you'll learn:

To demonstrate how React Router DOM works, we'll create an example React app. You can find an interactive demo on CodeSandbox. For reference, the code of the final example is available on GitHub.

What is a router in React?

Single-page applications (SPAs) rewrite sections of a page rather than loading entire new pages from a server.

Twitter is a good example of this type of application. When you click on a tweet, only the tweet's information is fetched from the server. The page does not fully reload:

These applications are easy to deploy and greatly improve the user experience. However, they also bring challenges.

One of them is browser history. Because the application is contained in a single page, it cannot rely on the browser's forward/back buttons per se. It needs something else. Something that, according to the application's state, changes the URL to push or replace URL history events within the browser. At the same time, it also needs to rebuild the application state from information contained within the URL.

On Twitter, for example, notice how the URL changes when a tweet is clicked:

And how a history entry is generated:

This is the job of a router.

A router allows your application to navigate between different components, changing the browser URL, modifying the browser history, and keeping the UI state in sync.

What is React Router?

React is a popular library for building SPAs. However, as React focuses only on building user interfaces, it doesn't have a built-in solution for routing.

React Router is the most popular routing library for React. It allows you define routes in the same declarative style:

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

But let's not get ahead of ourselves. Let's start by creating a sample project and setting up React Router.

I'm going to use Create React App to create a React app. You can install (or update) it with:

```
npm install -g create-react-app
```

You just need to have Node.js version 6 or superior installed.

Next, execute the following command:

```
create-react-app react-router-example
```

In this case, the directory `react-router-example` will be created. If you `cd` into it, you should see a structure similar to the following:

What does React Router DOM do?

React Router includes three main packages:

- `react-router`, the core package for the router
- `react-router-dom`, which contains the DOM bindings for React Router. In other words, the router components for websites

- `react-router-native`, which contains the React Native bindings for React Router. In other words, the router components for an app development environment using React Native

React Router DOM enables you to implement dynamic routing in a web app. Unlike the traditional routing architecture in which the routing is handled in a configuration outside of a running app, React Router DOM facilitates component-based routing according to the needs of the app and platform.

React Router DOM is the most appropriate choice if you're writing a React application that will run in the browser.

What is the difference between React Router and React Router DOM?

React Router is the core package for the router. React Router DOM contains DOM bindings and gives you access to React Router by default.

In other words, you don't need to use React Router and React Router DOM together. If you find yourself using both, it's OK to get rid of React Router since you already have it installed as a dependency within React Router DOM.

Note, however, that React Router DOM is only available on the browser, so you can only use it for web applications.

Can I use React Router DOM in React Native?

The react-router-native package enables you to use React Router in React Native apps. The package contains the React Native bindings for React Router.

Because React Router DOM is only for apps that run in a web browser, it is not an appropriate package to use in React Native apps. You would use react-router-native instead.

Installation

Because we are creating a web app, let's install `react-router-dom`:

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

At this point, you can execute:

```
npm start
```

A browser window will open `http://localhost:3000/` and you should see something like this:

Now let's create a simple SPA with React and React Router.

The React Router API: `<Router>`, `<Link>`, and `<Route>`

The React Router API is based on three components:

- `<Router>`: The router that keeps the UI in sync with the URL
- `<Link>`: Renders a navigation link
- `<Route>`: Renders a UI component depending on the URL

`<Router>`

Only in some special cases you'll have to use `<Router>` directly (for example when working with Redux), so the first thing you have to do is to choose a router implementation.

In a web application, you have two options:

- `<BrowserRouter>`, which uses the HTML5 History API.
- `<HashRouter>`, which uses the hash portion of the URL (window.location.hash)

If you're going to target older browsers that don't support the HTML5 History API, you should stick with `<HashRouter>`, which creates URLs with the following format:

```
http://localhost:3000/#/route/subroute
```

Otherwise, you can use `<BrowserRouter>`, which creates URLs with the following format:

```
http://localhost:3000/route/subroute
```

I'll use `<BrowserRouter>`, so in `src/index.js`, I'm going to import this component from `react-router-dom` and use it to wrap the `<App>` component:

```
import React from 'react';
// ...
import { BrowserRouter } from 'react-router-dom';

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

It's important to mention that a router component can only have one child element. For example, the following code:

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

Throws this error message:

The main job of a `<Router>` component is to create a history object to keep track of the location (URL). When the location changes because of a navigation action, the child component (in this case `<App>`) is re-rendered.

Most of the time, you'll use a `<Link>` component to change the location.

`<Link>`

Let's create a navigation menu. Open `src/App.css` to add the following styles:

```
ul {
  list-style-type: none;
  padding: 0;
}

.menu ul {
  background-color: #222;
  margin: 0;
}

.menu li {
  font-family: sans-serif;
  font-size: 1.2em;
  line-height: 40px;
  height: 40px;
  border-bottom: 1px solid #888;
}

.menu a {
```

In `scr/App.js`, replace the last `<p>` element in the `render()` function so it looks like this:

```
render() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <h1 className="App-title">Welcome to React</h1>
      </header>
      <div className="menu">
          <ul>
            <li> <Link to="/">Home</Link> </li>
            <li> <Link to="/messages">Messages</Link> </li>
            <li> <Link to="/about">About</Link> </li>
          </ul>
      </div>
    </div>
  );
}
```

Don't forget to import the `<Link>` component at the top of the file:

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

In the browser, you should see something like this:

As you can see, this JSX code:

```
<ul>
<li> <Link to="/">Home</Link> </li>
<li> <Link to="/messages">Messages</Link> </li>
<li> <Link to="/about">About</Link> </li>
</ul>
```

Generates the following HTML code:

```
<ul>
<li> <a href="/">Home</a> </li>
<li> <a href="/messages">Messages</a> </li>
<li> <a href="/about">About</a> </li>
</ul>
```

However, those aren't regular anchor elements. They change the URL without refreshing the page. Test it.

And now add a `<a>` element to the JSX code and test one more time:

```
<ul>
<li> <Link to="/">Home</Link> </li>
<li> <Link to="/messages">Messages</Link> </li>
<li> <Link to="/about">About</Link> </li>
<li>
    <a href="/messages">Messages (with a regular anchor element)</a>
  </li>
</ul>
```

Do you notice the difference?

`<Route>`

Right now, the URL changes when a link is clicked, but not the UI. Let's fix that.

I'm going to create three components for each route. First, `src/component/Home.js` for the route `/` :

```
import React from 'react';

const Home = () => (
  <div>
    <h2>Home</h2>
    My Home page!
  </div>
);

export default Home;
```

Then, `src/component/Messages.js` for the route `/messages`:

```
import React from 'react';

const Messages = () => (
  <div>
    <h2>Messages</h2>
    Messages
  </div>
);

export default Messages;
```

And finally, `src/component/About.js` for the route `/about`:

```
import React from 'react';

const About = () => (
  <div>
    <h2>About</h2>
    This example shows how to use React Router!
  </div>
);

export default About;
```

To specify the URL that corresponds to each component, you use the `<Route>` in the following way:

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

With other router libraries (and even in previous versions of React Router), you have to define these routes in a special file, or at least, outside your application.

This doesn't apply to React Router 4. These components can be placed anywhere inside of the router, and the associated component will be rendered in that place, just like any other component.

Note: The most recent stable version is React Router v5.2.0, released on 11 May 2020. React Router v.6.0.0 is currently in beta. For more information, check out React Router's version history.

So in `src/App.js`, import all these components and add a section after the menu:

```
// …
import {
  Route,
  Link
} from 'react-router-dom'

import Home from './components/Home';
import About from './components/About';
import Messages from './components/Messages';

class App extends Component {
  render() {
    return (
      <div className="App">
        ...
        <div className="menu">
            ...
        </div>
        <div className="App-intro">
```

In the browser, you should see something like this:

However, look what happens when you go to the other routes:

By default, routes are inclusive; more than one `<Route>` component can match the URL path and render at the same time.

Routes are the most important concept in React Router. Let's talk about routes in the next section.

Understanding routes

The matching logic of the `<Route>` component is delegated to the path-to-regexp library. I encourage you to check all the options and modifiers of this library and test it live with the express-route-tester.

In the previous example, since the `/message` and `/about` paths also contain the character `/`, they are also matched and rendered.

With this behavior, you can display different components just by declaring that they belong to the same (or a similar) path.

There's more than one solution.

The first one uses the `exact` property to render the component only if the defined path matches the URL path exactly:

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

If you test the application, you'll see that everything works fine.

The routes `/message` and `/about` are still evaluated, but they are not an exact match for `/` now.

However, if we know that only one route will be chosen, we can use a `<Switch>` component to render only the first route that matches the location:

```
// …
import {
  Route,
  Link,
  Switch
} from 'react-router-dom'
// …
class App extends Component {
  render() {
    return (
      …
      <div className="App-intro">
          <Switch>
            <Route exact path="/" component={Home} />
            <Route path="/messages" component={Messages} />
            <Route path="/about" component={About} />
          </Switch>
        </div>
      </div>
```

`<Switch>` will make the path matching exclusive rather than inclusive (as if you were using `<Route>` components).

For example, even if you duplicate the route for the `Messages` component:

```
<Switch>
  <Route exact path="/" component={Home} />
  <Route path="/messages" component={Messages} />
  <Route path="/messages" component={Messages} />
  <Route path="/about" component={About} />
</Switch>
```

When visiting the `/messages` path, the `Messages` component will be rendered only once.

However, notice that you'll still need to specify the exact property for the `/` path, otherwise, `/message` and `/about` will also match `/`, and the `Home` component will always be rendered (since this is the first route matched):

But what happens when a nonexistent path is entered? For example `http://localhost:3000/non-existent`:

In a regular JavaScript `switch` statement, you'll specify a default clause for this case, right?

In a `<Switch>` component, this default behavior can be implemented with a `<Redirect>` component:

```
// …
import {
  Route,
  Link,
  Switch,
  Redirect
} from 'react-router-dom'
// …
class App extends Component {
  render() {
    return (
      …
      <div className="App-intro">
          <Switch>
            <Route exact path="/" component={Home} />
            <Route path="/messages" component={Messages} />
            <Route path="/about" component={About} />
            <Redirect to="/" />
          </Switch>
```

This component will navigate to a new location overriding the current one in the history stack:

Now, let's cover something a little more advanced, nested routes.

Nested routes

A nested route is something like `/about/react`.

Let's say that for the messages section, we want to display a list of messages. Each one in the form of a link like `/messages/1`, `/messages/2`, and so on, that will lead you to a detail page.

You can start by modifying the `Messages` component to generate links for five sample messages in this way:

```
import React from 'react';

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

const Messages = () => (
  <div>
    <ul>
    {
        [...Array(5).keys()].map(n => {
            return <li key={n}>
                    <Link to={`messages/${n+1}`}>
                      Message {n+1}
                    </Link>
                  </li>;
        })
    }
    </ul>
```

This should be displayed in the browser:

To understand how to implement this, you need to know that when a component is rendered by the router, three properties are passed as parameters:

- `match`
- `location`
- `history`

For our purposes, we'll be using the `match` parameter.

When there's a match between the router's path and the URL location, a `match` object is created with information about the URL and the path. Here are the properties of this object:

- `params` : Key/value pairs parsed from the URL corresponding to the parameters
- `isExact` : `true` if the entire URL was matched (no trailing characters)
- `path` : The path pattern used to match
- `url` : The matched portion of the URL

This way, in the `Messages` component, we can destructure the properties object to use the `match` object:

```
const Messages = ({ match }) => (
<div>
...
</div>
)
```

Replace `/messages` with the matched URL of the `match` object:

```
const Messages = ({ match }) => (
  <div>
    <ul>
    {
        [...Array(5).keys()].map(n => {
            return <li key={n}>
                    <Link to={`${match.url}/${n+1}`}>
                      Message {n+1}
                    </Link>
                  </li>;
        })
    }
    </ul>
  </div>
);
```

This way you're covered if the path ever changes.

And after the message list, declare a `<Route>` component with a parameter to capture the message identifier:

```
import Message from './Message';
//…

const Messages = ({ match }) => (
  <div>
    <ul>
       ...
    </ul>
    <Route path={`${match.url}/:id`} component={Message} />
  </div>
);
```

In addition, you can enforce a numerical ID in this way:

```
<Route path={`${match.url}/:id(\\d+)`} component={Message} />
```

If there's a match, the `Message` component will be rendered. Here's its definition:

```
import React from 'react';

const Message = ({ match }) => (
  <h3>Message with ID {match.params.id}</h3>
);

export default Message;
```

In this component, the ID of the message is displayed. Notice how the ID is extracted from the `match.params` object using the same name that it's defined in the path.

If you open the browser, you should see something similar to the following:

But notice that for the initial page of the messages section (`/messages`) or if you enter in the URL and invalid identifier (like `/messages/a`), nothing is printed under the list. A message would be nice, don't you think?

You can add another route for this case, but instead of creating another component to just display a message, we can use the `render` property of the `<Route>` component:

```
<Route
  path={match.url}
  render={() => <h3>Please select a message</h3>
/>
```

You can define what is rendered by using one of the following properties of `<Route>` :

- `component` to render a component
- `render` , a function that returns the element or component to be rendered

- `children`, a function that also returns the element or component to be rendered. However, the returned element is rendered regardless of whether the path is matched or not

Finally, we can wrap the routes in a `<Switch>` component to guarantee that only one of the two is matched:

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

const Messages = ({ match }) => (
  <div>
    <ul>
     ...
    </ul>
    <Switch>
      <Route path={`${match.url}/:id(\\d+)`} component={Message}
/>
      <Route
        path={match.url}
        render={() => <h3>Please select a message</h3>}
      />
    </Switch>
```

However, you have to be careful. If you declare route that renders the message first:

```
 <Switch>
      <Route
        path={match.url}
        render={() => <h3>Please select a message</h3>}
      />
      <Route path={`${match.url}/:id(\\d+)`} component={Message} />
</Switch>
```

The `Message` component will never be rendered because a path like `/messages/1` will match the path `/messages`.

If you declare the routes in this order, add `exact` to avoid this behavior:

```
<Switch>
    <Route
      exact
      path={match.url}
      render={() => <h3>Please select a message</h3>}
    />
    <Route path={`${match.url}/:id(\\d+)`} component={Message} />
</Switch>
```

How to set the default route in React

Let's say we encounter a nonexistent route/path in our app. Instead of letting the browser show an error, we can customize a 404 page to tell our users with a neat UI that the page they are requesting is not available.

It might look something like this:

```
<Switch>
  <Route exact path="/" component={Home} />
  <Route path="/messages" component={Messages} />
  <Route path="/about" component={About} />
  <Route component={NotFound} />
</Switch>
```

The `<Route component={NotFound} />` displays the `NotFound` component when no other routes match the requested path. It is purposely set at the bottom of the `Switch` so it will be the last to be matched; putting it at the beginning will cause the `NotFound` page to always render.

Now, let's say we don't like the idea of using a `NotFound` page to display a nonexistent page. Instead, we want to set a default page to be loaded when a nonexistent route is hit.

Referring back to the above example, we want our `Home` page to be the default page. This `Home` page will be displayed when the user navigates to a nonexistent path or route in our app. So instead of a 404 page, a normal page is displayed.

Using an asterisk (`*`)

To set up a default page in React Router, pass an asterisk (`*`) to the `Route` 's `path` prop:

```
<Switch>
  <Route exact path="/" component={Home} />
  <Route path="/messages" component={Messages} />
  <Route path="/about" component={About} />
  <Route path="*" component={Home} />
</Switch>
```

This `<Route path="*" component={Home} />` handles nonexistent routes in a special way. The asterisk at the `path` prop causes the route to be called when a nonexistent path is hit. It then displays the `Home` component.

`Home` is now set as the default page. If we navigate to `localhost:3000/planets` , the `Home` component will be displayed under the `localhost:3000/planets` URL. The path does not exist in our routing configuration, so React Router displays our default page, the `Home` page.

Using `Redirect`

We can use another technique to set default page in React Router:

```
<Switch>
  <Route exact path="/" component={Home} />
  <Route path="/messages" component={Messages} />
  <Route path="/about" component={About} />
  <Redirect to="/" />
</Switch>
```

This method redirects the URL to `/` when a nonexistent path is hit in our application and sets the route `/` and `Home` as our default page.

So if we navigate to `localhost:3000/planets` in our browser, React Router will redirect to `localhost:3000` and display the `Home` component because the route `localhost:3000/planets` does not exist in our routing config.

Using `Router` with no path props

We did this earlier, but this time we will do away with the `Notfound` page and set the `Route` to call our default component.

```
<Switch>
  <Route exact path="/" component={Home} />
  <Route path="/messages" component={Messages} />
  <Route path="/about" component={About} />
  <Route component={Home} />
</Switch>
```

We want the `Home` component to be our default component. The `<Route component={Home} />` is run when no route is matched and the `Home` component is displayed instead.

Conclusion

In a few words, a router keeps your application UI and the URL in sync.

React Router is the most popular router library for React, and since version 4, React Router declarative defines routes with components in the same style as React.

In this post, you have learned how to set up React Router, its most important components, how routes work, and how to build dynamic nested routes with path parameters.

But there's still a lot of more to learn. For example, there a <NavLink>component that is a special version of the `<Link>` component that adds the properties `activeClassName` and `activeStyle` to give you styling options when the link matches the location URL.

The official documentation covers some basic examples as well as more advanced, interactive use cases.

Full visibility into production React apps

Debugging React applications can be difficult, especially when users experience issues that are hard to reproduce. If you're interested in monitoring and tracking Redux state, automatically surfacing JavaScript errors, and tracking slow network requests and component load time, try LogRocket.

LogRocket is like a DVR for web and mobile apps, recording literally everything that happens on your React app. Instead of guessing why problems happen, you can aggregate and report on what state your application was in when an issue occurred. LogRocket also monitors your app's performance, reporting with metrics like client CPU load, client memory usage, and more.

The LogRocket Redux middleware package adds an extra layer of visibility into your user sessions. LogRocket logs all actions and state from your Redux stores.

Modernize how you debug your React apps — start monitoring for free.

Esteban Herrera Follow

Family man. Java and JavaScript developer. Swift and VR/AR hobbyist. Like books, movies, and still trying many things. Find me at eherrera.net

#react

4 Replies to "React Router DOM: How to handle routing in web..."

Lwin Moe Says:
June 20, 2019 at 2:12 am

Reply

this is a really nice one for beginner.

Arun Nohwar Says:
May 11, 2020 at 2:26 pm

Reply

This is extremely good , so nicely explained and provides a comprehensive understanding about the concept of router-dom and how to use it. Really thanks for such a great artcle.

Atila Santos Says:
May 12, 2020 at 6:21 pm

Reply

Perfect, I was working by feeling and experience with another technologies, but not by knowlegde. I could open my eyes with this arcticle. Thanks a lot

Sonic Says:
September 22, 2021 at 8:05 am

Reply

this is already outdated for react router v6

Leave a Reply

Enter your comment here...