Router v4: Grok React Router in 20 minutes

Eduardo Vedes Follow
Aug 26, 2018 · 12 min read

"A brown mug next to a MacBook with lines of code on its screen" by Artem Sapegin on Unsplash

Hi fellow React Hitchhiker! Want a ride into React Router? Jump in. Let's go!

To understand the philosophy behind React Router, we need to know what a Single-Page Application (SPA) is.

What Is A Single-Page Application?

Basically it's a web application or web site that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server.

Why is this so good?!

1. avoids interruption of the user experience between successive pages

2. makes the application behave more like a desktop application

3. all the code resources are dynamically loaded and added to the page as necessary, usually in response to user actions

4. because it's kewl and kewl and extra-ultra-wide-4K-level-of-kewl.

SPAs are an industry standard now, and lots of companies are in a quest to find programmers to develop their projects.

What is React Router?

React Router is a tool that allows you to handle routes.

Since you're dealing with an SPA, you need a way to trigger the contents that are loaded on the screen. React Router introduces a concept called "Dynamic Routing", which is quite different from the "Static Routing" we are used to.

When you're dealing with "Static Routing" you declare your routes as part of your app's initialization before any rendering takes place (Rails, Express, Ember, Angular, and so on).

"Dynamic Routing" means that routing takes place as your app is rendering, not in a configuration or convention outside of a running app.

React Router v4 advocates and implements a component-based approach to routing.

It provides different Routing Components according to the needs of the application and platform.

In this specific case we're going to explore `<BrowserRouter>` because we want to use "dynamic routing" in a "web app" context and leave the other ones for other circumstances.

Who Created React Router?

These two amazing human beings, Michael Jackson and Ryan Florence. And they deserve loads, tons of claps! Together they started React Training.

Nowadays, correct me if I'm wrong, they followed separate paths:

Michael Jackson continues to develop React Training.

Ryan Florence created Reach.Tech.

Has React Router Anything To Do With Redux?

No. Although they typically appear together.

Are you sure? Yes 😃 I am sure 😄

They're both great and indispensable tools and as they are Higher Order Components (basically JavaScript functions that take a Component and return a new one), so it's common to find them "composed" together.

Setup, Let's Get Our Hands Dirty

Photo by Rose Elena on Unsplash

To guide you through this process we'll use `Create React App` (CRA).

In the end you'll have a clean boilerplate to build simple websites.

If by any chance React or Create React App are beyond your grasp, I recommend you to first get into those and then come back with a cup of coffee.

Okay, to those who stood with me: after installing CRA, you need to install the react-router package.

If you use npm just open your terminal, go to your CRA folder and type:

```
npm i -S react-router-dom
```

or

`yarn add react-router-dom` —if you use yarn as your package manager.

Just to check your `package.json` and make sure everything is okay, here's mine:

```
{
  "name": "react_router_boilerplate_01",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "npm": "^6.4.0",
    "react": "^16.4.2",
    "react-dom": "^16.4.2",
    "react-router-dom": "^4.3.1",
    "react-scripts": "1.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

As you can see at this point we have `react-router-dom` as a dependency.

Done, npm or yarn start and…

Bang! We're riding Ma! 🚵

The App We Are Building

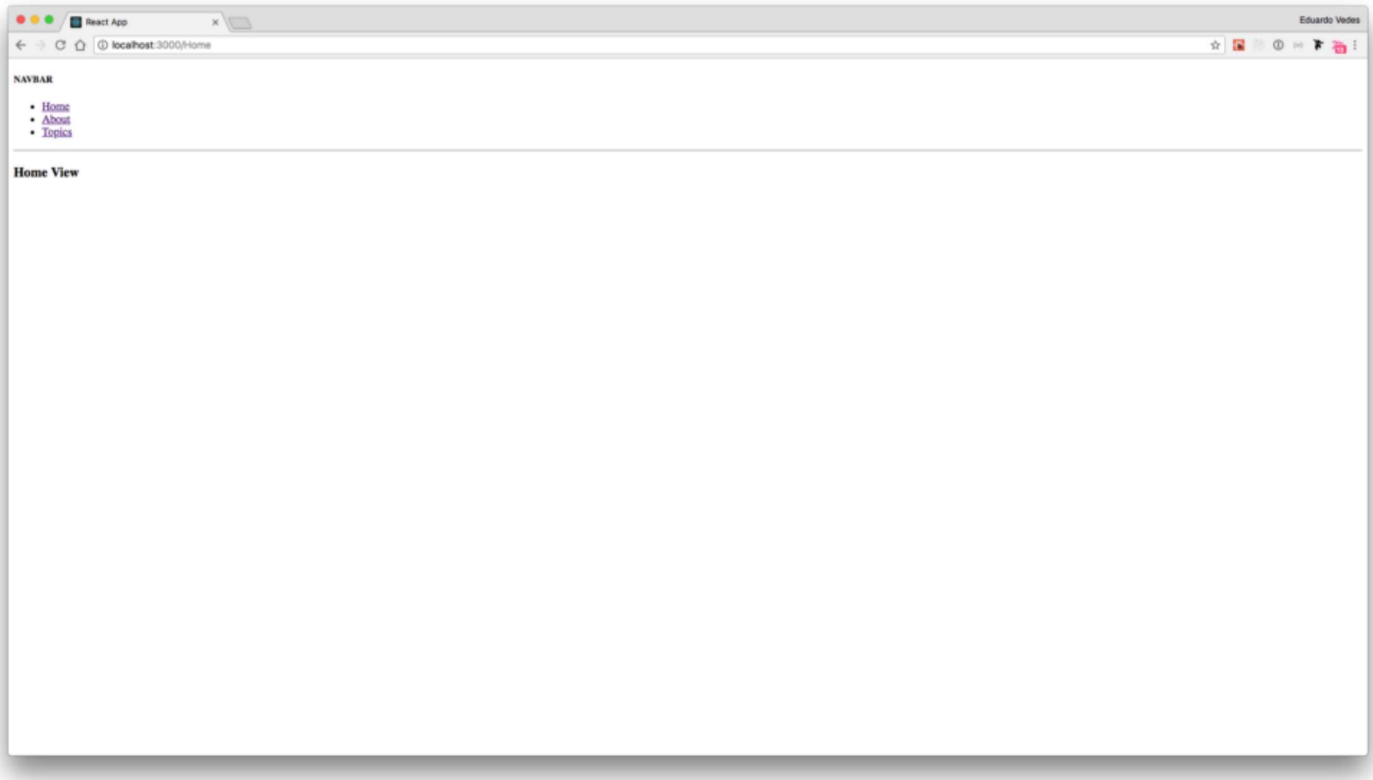
Let's do a simple personal website with a navigation bar that allows the user to switch between content. Our website will have three main sections called Home, About and Topics.

The NavBar will be an omnipresent component while the Home, About and Topics will be rendered below according to the routes selected.

Are you seeing the browser URL: `localhost:3000/Home` in the screenshot below?

It means that the Home route is triggered and the Home view is rendered.

This will be our final result:



And this… 👆, this is a website?

😸 Yes, It is!

A naked one! Just try not to feel bias towards other complexities like styling and so on! I don't want you to be distracted with anything other than grokking **how simple** it is to implement React Router v4.

So, after you've recovered from the shock, 🐱, let's take the next step and see my `/src/index.js` file.

/src/index.js

`index.js` is the first file to be loaded by CRA, the initialization point of everything in your App.

Let's take a look at what I've done:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router } from 'react-router-dom';
import { Routes } from './routes'; // where we are going to specify our routes
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(
  <Router>
    <Routes />
  </Router>,
  document.getElementById('root'),
);

registerServiceWorker();
```

So what are we doing here?

- We are importing the `<BrowserRouter />` component from the dependency we've installed and stating that we're going to call it `<Router />` from this point on:

```
import { BrowserRouter as Router } from 'react-router-dom';
```

- We are importing a `<Routes />` component, created by me, with the routes we're going to use in our web site—don't worry right now with this Component:

```
import { Routes } from './routes';
```

The `<Routes />` component is taking the place of the default CRA `<App />` component. It's basically the same—I just called it `<Routes />` because I feel it makes sense to turn the code more meaningful and readable.

You are not loading an unique App anymore but a **`Routes`** component that will handle the routes and will trigger the mounting and rendering of the components which shall load within each route.

- We are embracing `<Routes />` with the `<Router />` component.

As a matter of fact, `<Router />` works as a Higher Order Component that only knows its children in the future and interacts with them in a more wide scope, independently of who and how many they are.

You do not have to worry about how it works to use it. This is a much deeper and advanced matter.

Just make sure you understand that React.DOM is not anymore **loading a simple App**. It's loading the App embraced by a Component called `Router` that in a higher instance or scope can interact with it and with the browser `DOM` .

<Routes /> Component

```
import React from 'react';
import { Home } from './views/Home';
import { About } from './views/About';
import { TopicList } from './views/TopicList';
import { NoMatch } from './views/NoMatch';
import { TopicDetail } from './components/TopicDetail';
import { NavBar } from './components/NavBar';
import { Route, Switch, Redirect } from 'react-router-dom';

export const Routes = () => {
  return (
    <div>
      <NavBar />
      <Switch>
        <Route exact path="/Home" component={Home} />
        <Route exact path="/">
          <Redirect to="/Home" />
        </Route>
        <Route exact path="/About" component={About} />
        <Route exact path="/About" component={Home} />
        <Route exact path="/Topics" component={TopicList} />
        <Route path="/Topics/:topicId" component={TopicDetail} />
        <Route component={NoMatch} />
      </Switch>
    </div>
  );
};
```

So basically what does routes.js do?

It starts by importing React and a few components we'll take a look later. Just think of them as simple stateless components: Home, About, TopicList, TopicDetail, NavBar and NoMatch.

It also imports three components from the react-router-dom package which we'll need to invoke: `<Route />`, `<Switch />` and `<Redirect />`.

After the imports, we export the stateless component Routes which invokes the `NavBar` (which will be always in the screen) and a `<Switch />` component.

What does this <Switch /> guy do?

This component basically renders the first child `<Route>` or `<Redirect>` that matches the browser location.

It starts to test stuff like this: "is the browser URL in this `<Route />` path? No? Okay." Next Route. "Is the browser URL in this other route path? No."

Next Route. "Oh, I got it! It's in this one, let's trigger the Component `render` and finish the checking by now (I don't care with the other routes below…)"

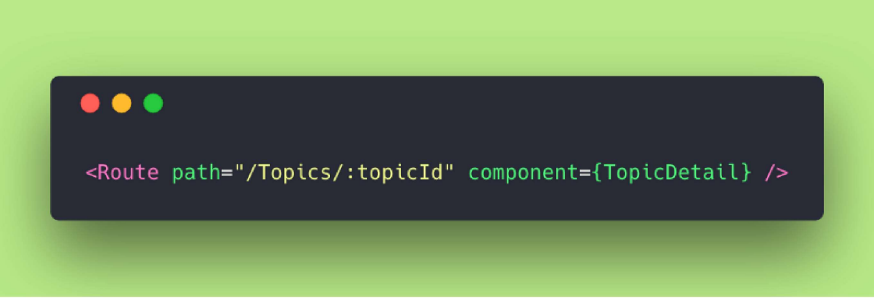
If by any chance this happens:

the second route will never be triggered because `Switch` will jump off before reaching it. He just goes to have a coffee… (and me too!!! 😃 Back!)

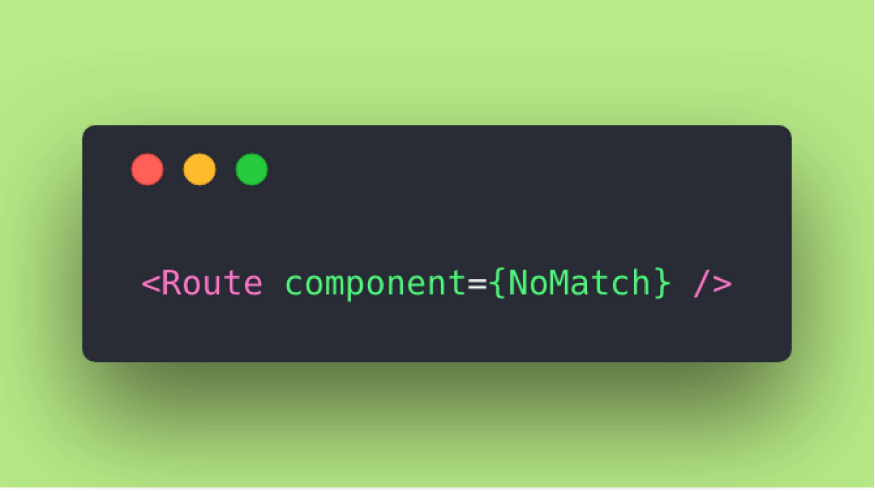
Inside `<Switch />` we define each `<Route />`.

Each `<Route />` tells this to the browser:
"Hey browser DOM! If `<Switch />` chooses me because your location is (exactly) this one, please render the following Component".

Or in other cases such as the one below, it says:
"Hey, browser, if by any instance your `<Switch />` chose me because location is `/Topics/` "something" render Component `TopicDetail`. Certainly it'll find out who is this **`:topicId`** (variable) thing that the user is asking us to match and route it accordingly".



Okay everyone. Because `<Switch />` has this default behavior of checking each route, we need to provide a fallback in case it doesn't match anything:



This last Route simply renders a default page stating that no route was matched, kind of an HTTP 404 error.

Remember that here we're dealing with an SPA and with "Dynamic Routing" so this is a simulation as if we were demanding routes to a server 😸. Actually we're not!

We just do not know what to render if the user, for instance, inserts something not mismatched into the URL like this: `http://localhost:3000/HelloWorld`.

As this route was not defined, we provide a `NoMatch` component to inform them about the non-existence of the route.

`<Redirect />` is there because if the user tries to load the URL without any route, `http://localhost:3000/`, it would get a `NoMatch` because there's no route defined for it. So the best way to handle this is make use of `<Redirect />` and push the user to the route of `/Home` which is by default our first screen of the app.

Why is this necessary?

Again, because usually the user would start the Application by typing it's general URL and without the `<Redirect />` the first rendered component would be `<NoMatch />`. We don't want that, we want the user to be redirected to the `<Home />` component.

Views And / Or Components

At this point in our guide, I'd like to stop a little bit to differentiate a View from a Component. This is not the essence of this guide, but will make sense after I show you the folder structure of my CRA.

When we are "Thinking in React" and we start making an App, and it starts to grow, sometimes we stop because we feel things are not in the right place.

This means that we need to give names to those things and keep them separated in different "drawers" or "folders".

Views and Components are things that are painted on screen. So what differentiates one thing from the other?

And are views not components? And components are not views?

Well, in terms of coding language, a View and a Component are certainly functions or classes—stateless components or stateful components as we call it in React lingo.

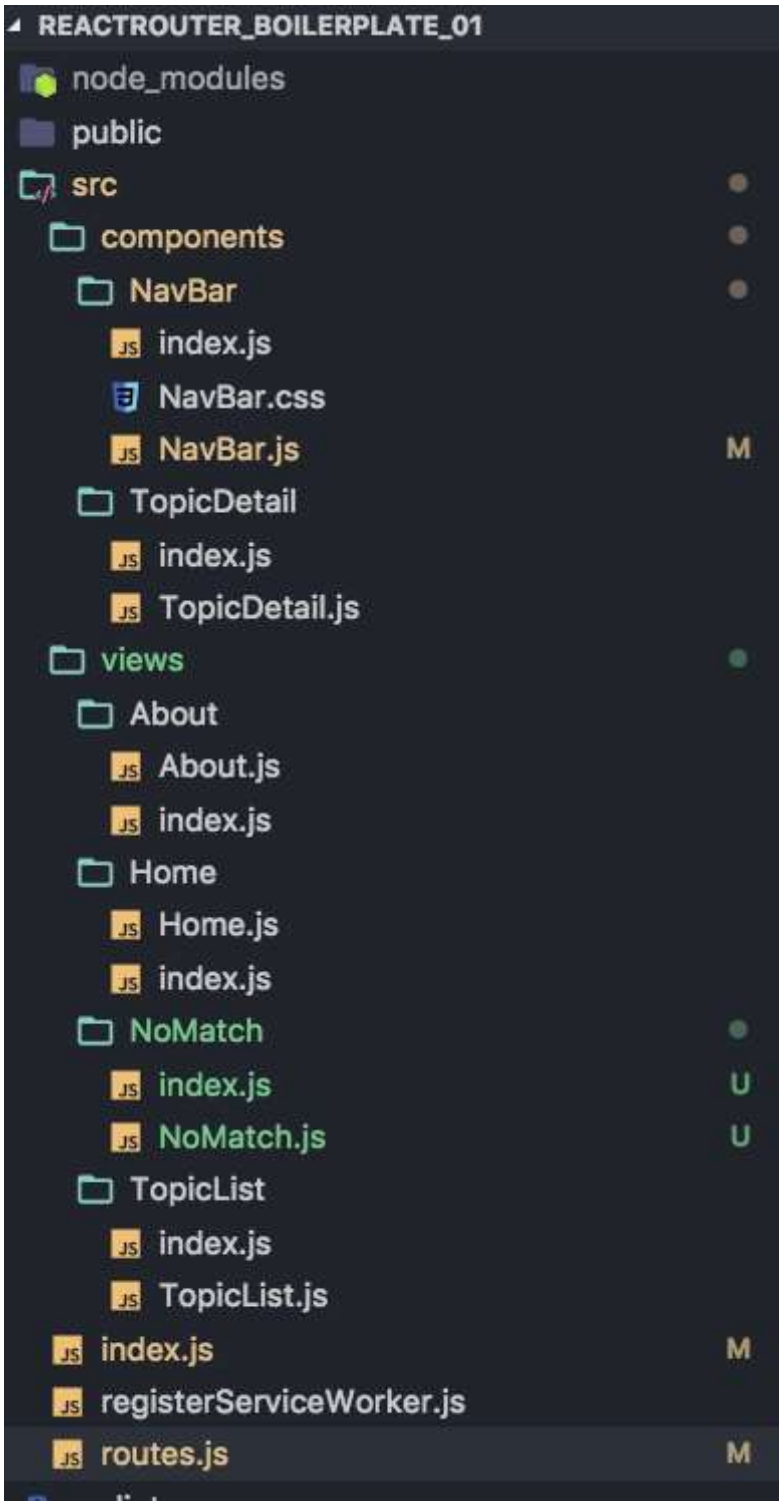
So what does differentiate them?

Well, a View has a route. Inside this View you can render a lot of components.

A component usually is an abstraction that can be invoked a lot of times in different views. It can be a button, a form, a chart. It can even be a more complex thing, while a view is unique and has a route.

This is a very simple concept that must be understood at the beginning, as soon as we start doing an app so small as a personal homepage.

Let's take a look at my CRA folder structure:



CRA folder structure

So, as you can see I—and 99% of the world—like to keep oranges and pears in different baskets. And so do you! I have faith in you! I trust you!

There are a lot of patterns on how to organize this stuff and a lot of discussion starts when we introduce more packages like Redux that transform a little bit the architecture of the app, or when we want to

paint on the screen Dashboards, Widgets, Cycling Pigs or more weird stuff…

But, to differentiate concepts, take a careful look at Views and Components.

`Home` , `About` , `TopicList` and `NoMatch` are views. They have their own proper routes that trigger them.

`NavBar` is an omnipresent component that's always invoked. It doesn't have a route.

`TopicDetails` is a component that will display Topic info when the `TopicList/:topicId` route is triggered. It's a reusable component that can be imported into other places and refactored or extended. It doesn't have a specific route.

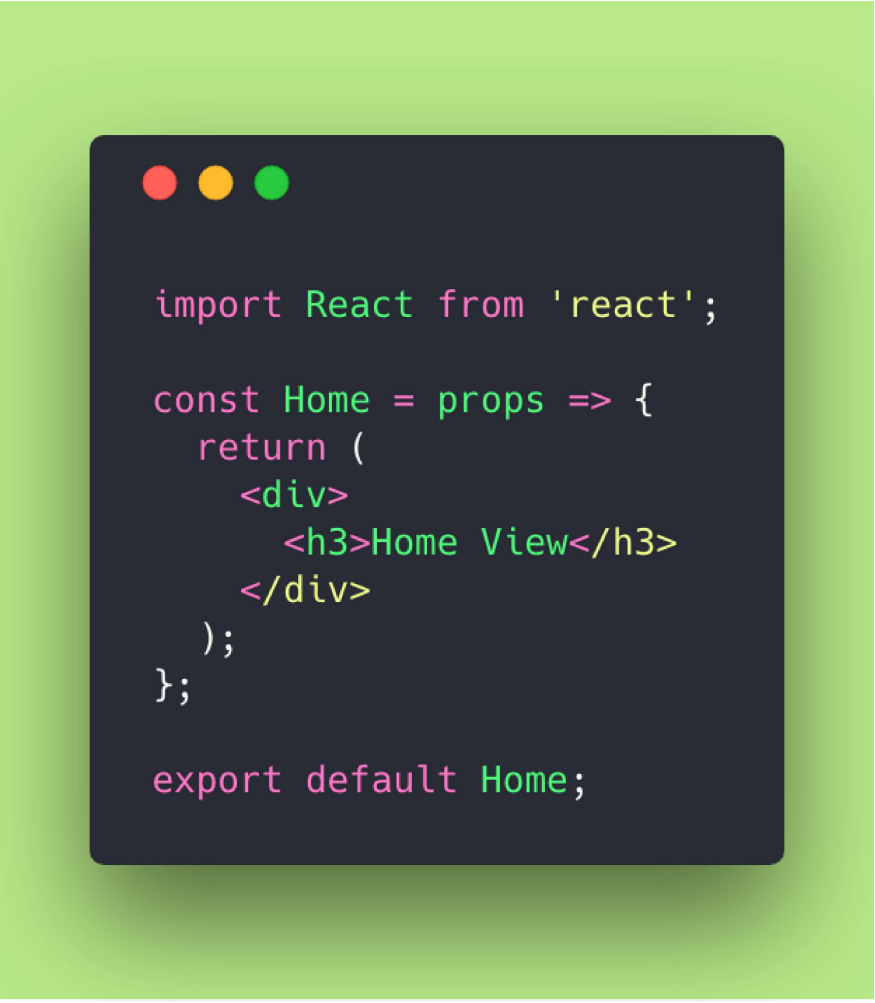
The Home / About Views

Inside the Home folder, I have an `index.js` and a `Home.js` file.

Having an `index.js` to export the other files is a good practice. Just trust me or bring some wine because this will be a long talk

… oh, let's just drink the wine and we will talk later!

index.js that exports Home view



```
import React from 'react';

const Home = props => {
  return (
    <div>
      <h3>Home View</h3>
    </div>
  );
};

export default Home;
```

Home.js view stateless component

This is a simple view that only exports its title. The `About` view is equal to this one.

Now let's take a look at the `TopicList View` because it's a little bit different.

TopicList and TopicDetail Views

TopicList View Code

So `TopicList View` has this detail of handling different routes. Remember that `/Topic/:topicId` route that `<Route />` told `<Switch />` to let `TopicDetail` handle? Here we are with that.

`TopicList` receives `{ match }` as a prop. Don't let the destructuring feature scare you. We could simply receive props and call `props.match`. This is simply how all the cool kids nowadays destructure props to improve readability and React flow. I also like it a lot! This is kind of like picking up a box with your mobile inside or picking up the mobile directly. As a matter of fact it was kept inside the box but at this moment you only need to check your e-mail 😸 so let the box stay where it is! Don't bring it with you to work!

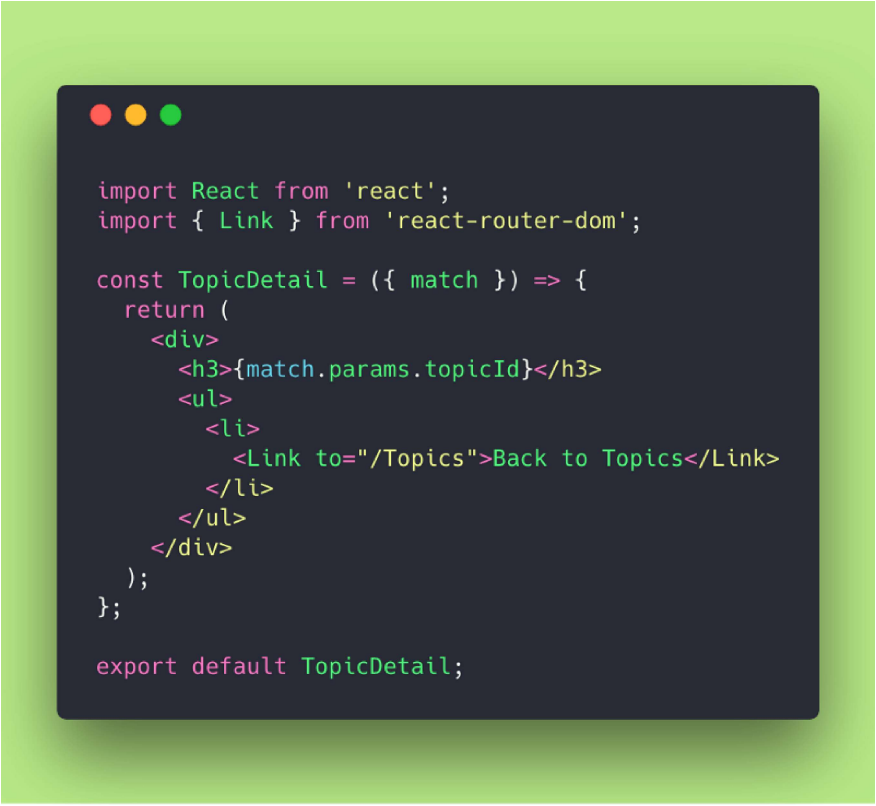
Anyways, let's keep focused on code.

In this file, we import a Component from React Router called `{ Link }` because we want to create Links 😀

We receive a match from the Route we've chosen when we've clicked `Topics` and we are rendering an unordered list with 3 options: `Topic1`, `Topic2` and `Topic3`.

Basically if the User chooses `Topic1 Link` in the screen, the `<Link />` will push the browser URL to that path `/Topics/Topic1`.

What happens next? `<Router />` and `<Switch />` detect that the URL changed and take a look into their info to check what route needs to be fired. So they discover that now the route triggered is the one for `/Topics/:topicId` and triggers `TopicDetail` rendering. `TopicDetail` will render `Topic1` details.



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

const TopicDetail = ({ match }) => {
  return (
    <div>
      <h3>{match.params.topicId}</h3>
      <ul>
        <li>
          <Link to="/Topics">Back to Topics</Link>
        </li>
      </ul>
    </div>
  );
};

export default TopicDetail;
```

TopicDetail Component

`TopicDetail` receives `match` from the Router and renders the `topicId` located at `match.params.topicId`.

The NavBar Component

The `NavBar` component has a special role here because it's omnipresent.

Its function is to allow the user to navigate the website and to show the sections (routes) available.

As you've seen in the beginning, it's inside `<Router />` but outside `<Switch />` so any view will always be composed with `NavBar` on top.

NavBar Component Code

As you can see, its role is basic. It only supplies `<Link />` and tells `<Router />` to ask `<Switch />` to trigger the chosen `<Route />` and render it on screen.

Last but not least

I think that by this time you probably have a basic understanding of how React Router works and can be used to do a simple website.

If you want to check the code or test it you can pull my repo, available on GitHub.

Bibliography

To make this article, I've used the React Router documentation that you can find here.

All the other sites I've used are linked along the document to add info or provide context to what I've tried to explain to you.

This article is part 1 of a series called "Hitchhiker's Guide to React Router v4." Parts 2–4 coming to freeCodeCamp throughout this week!

- **Part II: [match, location, history]—your best friends!**
- **Part III: recursive paths, to the infinity and beyond!**
- **Part IV: route config, the hidden value of defining a route configuration array;**

Thank you very much!