**Step 1. Getting Started**

To get started you can either [clone the starter repo](https://github.com/dabit3/beginners-guide-to-react-router) and jump to step two, or follow along the next steps and set up your project manually.

**Manual Setup**

First, let’s get our environment set up with React, Babel, and webpack. First create a folder and cd into it. Then run npm init -y:

npm init -y

* -y just answers yes to all of the questions

Next, install react, react-router, and react-dom and save them as dependencies:

npm i react react-dom react-router --save

Next, install our dev dependencies. These will be webpack, webpack-dev-server, babel-core, babel-loader, babel-preset-es2015, and babel-preset-react

npm i webpack webpack-dev-server babel-core babel-loader babel-preset-es2015 babel-preset-react --save-dev

Now, let’s create the configuration files for webpack and babel:

touch .babelrc webpack.config.js

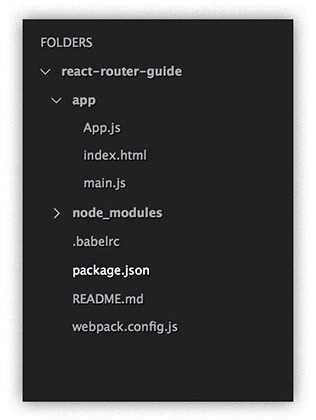
Next, let’s create a folder for our code. We’ll call this folder app:

mkdir app

In the app directory create three files: index.html app.js main.js

cd app  
touch index.html app.js main.js

Our file structure should now look like this:



Now, open the .babelrc file and add the presets for react and es2015:

{  
 "presets": [  
 "es2015",  
 "react"  
 ]  
}

In webpack.config.js, add the following configuration to get us started:

module.exports = {  
 entry: './app/main.js',  
 output: {  
 path: './app',  
 filename: 'bundle.js'  
 },  
 devServer: {  
 inline: true,  
 contentBase: './app',  
 port: 8100  
 },  
 module: {  
 loaders: [  
 {  
 test: /\.js$/,  
 exclude: /node\_modules/,  
 loader: 'babel'  
 }  
 ]  
 }  
}

*If you would like to learn more about webpack and babel,*[*check out my tutorial on beginning webpack.*](https://medium.com/@dabit3/beginner-s-guide-to-webpack-b1f1a3638460#.5tirb1odd)

Now that webpack and babel are set up. Let’s create a shortcut for webpack-dev-server. Open package.json and insert the following script in the “scripts” key:

"scripts": {  
 "start": "webpack-dev-server"  
}

Now, we can just run npm start to start our project.

Let’s now set up our HTML and React. Open index.html and create a base html page. Then, add a div with the id of root, and a script tag referencing bundle.js:

<!DOCTYPE html>  
 <html lang="en">  
 <head>  
 <meta charset="UTF-8">  
 <title>React Router</title>  
 </head>  
 <body>  
 <div id="root"></div>  
 <script src="./bundle.js"></script>  
 </body>  
</html>

Now, let’s go into our main.js and set up an entry point for our app. Type this into your main.js file:

import React from 'react'  
import ReactDOM from 'react-dom'  
import App from './App'  
ReactDOM.render(<App />, document.getElementById('root'))

Now, let’s go into App.js and create our app component. Open App.js and type the following:

import React, { Component } from 'react'  
import { Router, Route, Link, IndexRoute, hashHistory, browserHistory } from 'react-router'

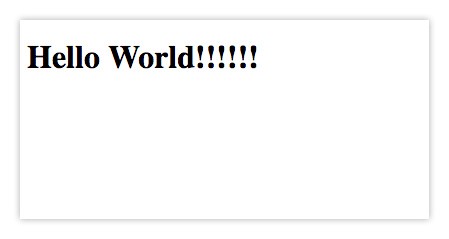
const App = () => <h1>Hello World!</h1>

export default App

We are not using Component or any of the Router / react-router components yet, but we are bringing them in so we can get started in step two.

Now, if you run the project and navigate to <http://localhost:8100/> you should get ‘Hello World!!!!!!’ on your screen:

npm start



Hello World

**Step 2. Basic Routing**

Let’s set up a basic route. We will replace the App component with a React class, which will return a Router component. Router will wrap all of the routes we are going to define.

Each route will be identified in a <Route> component. The <Route> component will take two properties: path and component. When a path matches the path given to the <Route> component, it will return the component specified.

In App.js, refactor the App component to look like this:

import React, { Component } from 'react'  
import { Router, Route, Link, IndexRoute, hashHistory, browserHistory } from 'react-router'

class App extends Component {  
 render() {  
 return (  
 <Router history={hashHistory}>  
 <Route path='/' component={Home} />  
 <Route path='/address' component={Address} />  
 </Router>  
 )  
 }  
}

const Home = () => <h1>Hello from Home!</h1>  
const Address = () => <h1>We are located at 555 Jackson St.</h1>

export default App

Now, if you navigate to <http://localhost:8100/> you should see our Home component, and if you navigate to <http://localhost:8100/#/address> you should see our Address component.

You will notice that there are random strings after the hash in your address bar:

*When using hash history, you’ll see an extra item in your query string that looks something like \_k=123abc. This is a key that history uses to look up persistent state data in window.sessionStorage between page loads.*[*Read more here.*](https://github.com/mjackson/history/blob/master/docs/HashHistoryCaveats.md)

If you would like a cleaner address, or you are using this in production, you may want to look into browserHistory vs hashHistory. When using browserHistory you must have a server that will always return your server at any route, for example if using nodejs, a configuration like the following (from the docs) would work:

const express = require('express')  
const path = require('path')  
const port = process.env.PORT || 8080  
const app = express()  
  
// serve static assets normally  
app.use(express.static(\_\_dirname + '/public'))  
  
// handle every other route with index.html, which will contain  
// a script tag to your application's JavaScript file(s).  
app.get('\*', function (request, response){  
 response.sendFile(path.resolve(\_\_dirname, 'public', 'index.html'))  
})  
  
app.listen(port)  
console.log("server started on port " + port)

To learn more about browserHistory, check out [this link.](https://github.com/reactjs/react-router/blob/master/docs/guides/Histories.md#browserhistory)

For the rest of this tutorial, we will be using hashHistory.

**Step 3. 404 route**

Now, what happens if we hit a route that is not defined? Let’s set up a 404 route and component that will return if the route is not found:

const NotFound = () => (  
 <h1>404.. This page is not found!</h1>)

Now, *below* our ‘/address’ route, create the following route:

<Route path='\*' component={NotFound} />

Now, if we navigate to some route that has not been defined (<http://localhost:8100/#/asdfasdf>) , we should see our 404 route.

**Step 4. IndexRoute and Links**

Now, let’s create add navigation to get us between pages.

To do this, we will be using the <Link> component. <Link> is similar to using an html anchor tag.

From the docs:

*The primary way to allow users to navigate around your application. <Link> will render a fully accessible anchor tag with the proper href.*

To do this, let’s first create a Nav component. Our Nav component will contain <Link> components, and will look like this:

const Nav = () => (  
 <div>  
 <Link to='/'>Home</Link>&nbsp;  
 <Link to='/address'>Address</Link>  
 </div>  
)

*Now we need a way to make our Nav component persistent across all pages.*To do this, we will wrap our child routes in a main <Route> component. We will also need to update our Home component, and create a new component called Container:

Container:

const Container = (props) => <div>  
 <Nav />  
 {props.children}  
</div>

*props.children will allow any routes wrapped within this route to be rendered in this component.*

Now, let’s rewrite our App component to look like this. We are wrapping our HomePage, Address and NotFound routes inside the new Container route. We are also setting HomePage to be our IndexRoute. That means that when we hit [http://localhost:8100](http://localhost:8100/#/), our Home component will render, as it is specified as the index:

class App extends Component {  
 render () {  
 return (  
 <Router history={hashHistory}>  
 <Route path='/' component={Container}>  
 <IndexRoute component={Home} />  
 <Route path='/address' component={Address} />  
 <Route path='\*' component={NotFound} />  
 </Route>  
 </Router>  
 )  
 }  
}

For reference, our full App.js code should look like [this](https://gist.github.com/dabit3/3d0d47c4a8bfccadfd5d15c58cfb1424).

Now, when we navigate to [http://localhost:8100](http://localhost:8100/), we should see our Home Component rendered, along with our Nav <Link> components!

**Step 5. Multiple child / IndexRoutes**

Now, let’s say we want to nest a twitter feed and an Instagram feed in our address component. Let’s create that functionality.

First, let’s rewrite our address route to take two new components: InstagramFeed and TwitterFeed:

class App extends Component {  
 render () {  
 return (  
 <Router history={hashHistory}>  
 <Route path='/' component={Container}>  
 <IndexRoute component={Home} />  
 <Route path='address' component={Address}>  
 <IndexRoute component={TwitterFeed} />  
 <Route path='instagram' component={Instagram} />  
 </Route>  
 <Route path='\*' component={NotFound} />  
 </Route>  
 </Router>  
 )  
 }  
}

We’ve set the IndexRoute of address to be TwitterFeed, and have added the Instagram route there as well.

Now, let’s create our InstagramFeed and TwitterFeed components. These will be very basic just so we know we’ve hit the correct routes:

const Instagram = () => <h3>Instagram Feed</h3>  
const TwitterFeed = () => <h3>Twitter Feed</h3>

Finally, go into the Address component, and add the Links to the new components as well as props.children, so the components will be rendered:

const Address = (props) => <div>  
 <br />  
 <Link to='/address'>Twitter Feed</Link>&nbsp;  
 <Link to='/address/instagram'>Instagram Feed</Link>  
 <h1>We are located at 555 Jackson St.</h1>  
 {props.children}  
</div>

Now, when we navigate to <http://localhost:8100/#/address>, the address component should be rendered as well as the TwitterFeed component:



For reference, the code up to now should look like [this](https://gist.github.com/dabit3/0c2014b421f2bf98cd95d176f0b29bad).

**Step 6. activeStyle / activeClassName and IndexLink**

We will now look at how to style to a Link based on whether the route is active. There are two main ways to do this, either adding style directly or through a class.

From the docs:

*<Link> can know when the route it links to is active and automatically apply an activeClassName and/or activeStyle when given either prop. The <Link> will be active if the current route is either the linked route or any descendant of the linked route.****To have the link be active only on the exact linked route, use***[***<IndexLink>***](https://github.com/reactjs/react-router/blob/5e483bff96859aa23f42e79aa979632ddcbfd2bc/docs/API.md#indexlink)***instead or set theonlyActiveOnIndex prop.***

First, let’s look at activeStyle. To apply activeStyle, you simply add activeStyle as a property to a <Link> and pass in the styling you would like the <Link> to have:

<Link activeStyle={{color:'#53acff'}} to=''>Home</Link>

Let’s update our Nav component to implement this:

const Nav = () => (  
 <div>  
 <Link activeStyle={{color:'#53acff'}} to='/'>Home</Link>&nbsp;  
 <Link activeStyle={{color:'#53acff'}} to='/address'>Address</Link>&nbsp;  
 <Link activeStyle={{color:'#53acff'}} to='/about'>About</Link>  
 </div>  
)

Now, let’s take a look at how this looks in our browser. You may notice that when you click on address, that Home is still highlighted:



This is because when using <Link> along with activeStyle, the <Link> will be active if the current route is either the linked route *or any descendant of the linked route.*

This means that because Address is a descendent of Home, it stays highlighted. To fix this, we can pass the onlyActiveOnIndex property to our Link component:

<Link onlyActiveOnIndex activeStyle={{color:'#53acff'}} to='/'>Home</Link>

Now, when we look at our browser, the link will only be highlighted if we are on the exact link:



There is also a sibling component of <Link> called <IndexLink>. <IndexLink> that is only active when the current route is exactly the linked route.

From the docs:

*An <IndexLink> is like a*[*<Link>*](https://github.com/reactjs/react-router/blob/5e483bff96859aa23f42e79aa979632ddcbfd2bc/docs/API.md#link)*, except it is only active when the current route is exactly the linked route. It is equivalent to<Link> with the onlyActiveOnIndex prop set.*

To implement this, first bring in <IndexLink> from react-router:

import { ..., IndexLink } from 'react-router'

Now, simply replace the <Link> components in nav with <IndexLink> components:

const Nav = () => (  
 <div>  
 <IndexLink activeStyle={{color:'#53acff'}} to='/'>Home</IndexLink>&nbsp;  
 <IndexLink activeStyle={{color:'#53acff'}} to='/address'>Address</IndexLink>&nbsp;  
 <IndexLink activeStyle={{color:'#53acff'}} to='/about'>About</IndexLink>  
 </div>  
)

Now, how about adding classes vs styles? To do this, we can use activeClassName. Let’s set up an active style in our index.html:

<style>  
 .active {  
 color:#53acff  
 }  
</style>

Now, we’ll replace activeStyle with activeClassName in our Nav component:

const Nav = () => (  
 <div>  
 <IndexLink activeClassName='active' to='/'>Home</IndexLink>&nbsp;  
 <IndexLink activeClassName='active' to='/address'>Address</IndexLink>&nbsp;  
 <IndexLink activeClassName='active' to='/about'>About</IndexLink>  
 </div>  
)

For reference, our code should now look like [this](https://gist.github.com/dabit3/ae4eeea9906c26e5643145664d540d0d).

**Step 7. Named Components**

Using Named Components, we can specify component as props to a <Route>.

From the docs:

*When a route has one or more named components, the child elements are available by name on this.props. In this case this.props.children will be undefined. All route components can participate in the nesting.*

Let’s now dig into the code and see how this would actually look.

First, let’s create a new Component that will be rendering our Named Components. These components will be available as props:

const NamedComponents = (props) => (  
 <div>  
 {props.title}<br />  
 {props.subTitle}  
 </div>  
)

Next, let’s create two new components called Title and Subtitle:

const Title = () => (  
 <h1>Hello from Title Component</h1>  
)  
const SubTitle = () => (  
 <h1>Hello from SubTitle Component</h1>  
)

Now, let’s create a new route for our NamedComponents component, and define the Title and Subtitle components in the IndexRoute:

<Route path='/namedComponent' component={NamedComponents}>  
 <IndexRoute components={{ title: Title, subTitle: SubTitle }} />  
</Route>

Finally, let’s add a link to our nav to navigate to this component:

<IndexLink activeClassName='active' to='/namedComponent'>Named Components</IndexLink>

Now, we should see our new Named Components link when we look at our browser, and when clicking on the link we should see our Title and SubTitle components rendering on the screen:



For reference, our code should now look like [this](https://gist.github.com/dabit3/5a75ecdba89dc2a45c1aaaf2727ddad1).

**Step 8. Route Parameters**

An essential part of many applications is the ability to read route parameters from a url.

To implement this, let’s revisit our About component. First, let’s rewrite the path in our Router to take an optional parameter, we’ll call it name:

<Route path='/about/:name' component={About} />

Now, let’s rewrite our About component to use this name variable:

const About = (props) => (  
 <div>  
 <h3>Welcome to the About Page</h3>  
 <h2>{props.params.name}</h2>  
 </div>  
)

Now, if we visit <http://localhost:8100/#/about/nader> we will see my name displayed below “Welcome to the About Page”.

The only issue here is that if we revisit <http://localhost:8100/#/about>, we get a 404 because there is no name parameter. To fix this, we can make the parameter optional by wrapping it in parenthesis:

<Route path='/about(/:name)' component={About} />

Now, if we visit <http://localhost:8100/#/about> we no longer get a 404, and can still access the name variable.

We can also take this one step further by checking to see if props.name is available and displaying some content:

{ props.params.name && <h2>Hello, {props.params.name}</h2>}

Now, the content will only be shown if there is a name parameter available.

For reference, our code should now look like [this](https://gist.github.com/dabit3/a31358742f837cf4826d55828931543f).

**Step 9. Query String Parameters**

You can also pass in query strings as props to any component that will be rendered at a specific route, and access these parameters as props.location.query.

To see how this works, let’s create a new component called Query, and render a property called props.location.query.message:

const Query = (props) => (  
 <h2>{props.location.query.message}</h2>  
)

Now, let’s set up our new Query Route within the address route we already have created:

...  
<Route path='/address' component={Address}>  
 <IndexRoute component={TwitterFeed} />  
 <Route path='instagram' component={Instagram} />  
 <Route path='query' component={Query} />  
</Route>  
...

Finally, let’s link to this route by creating a new Link component, and passing in a query string called message and giving it a value. This is done in the ‘to’ property that we have already used.

Instead of passing a link to ‘to’, we instead pass in an object the the pathname and query properties defined:

<IndexLink   
 activeClassName='active'   
 to={{   
 pathname: '/address/query',   
 query: { message: 'Hello from Route Query' }   
 }}>Route Query</IndexLink>

Now, if we click on our Route Query link, we should see our message rendered on the screen:

