React Router

- React-Router v4
- React-Router v2

React Router v4

This new version of react router is totally different from previous versions.
Everything is “component” based. A bit hard to get your head around, but here goes.

Imports and npm

React router is now broken up into either *react-router-dom* or *react-router-native*. The former for browser based code and the latter for react native applications.

We will review the main components, which are:

- BrowserRouter
- Route
- Switch
- Link
- Redirect

BrowserRouter

This is the top level component that you will wrap all of your other components in. In Electron I found using MemoryRouter worked better and if you need to have access to your own history component, use just the Router component.

Router

Sidenote

> If using redux, BrowserRouter will go inside of redux’s Provider component. Also, there is a weird thing where components using connect won’t rerender. To make them rerender, you must wrap the connect call with react-router’s withRouter higher order function

```
class MyConnectedComponent extends React.Component{
        render() {
        ....
    }
}
export default withRouter(connect()(MyConnectedComponent));
```

One way of using BrowserRouter, would be to have an index.js file that will wrap you main app.js file. app.js would have your top bar/navigation and then your initial routes. You can also embed Routes in your ReacDOM render call, just make sure only one Child component is under the BrowserRouter component. This can easily be done by wrapping Routes in a .

As I start working more with this, I can see where you can use routes deeper within the application to conditionally show components.

Here is an example of an the initial component with the BrowserRouter wrapping other Routes:

```
import {
  BrowserRouter,
  Route
} from 'react-router-dom';
...
const App = () => (
  <BrowserRouter>
    <div className="container">
      <Header /> //Will always show
      //Routes below will conditionally show.
      <Route exact path="/" component={Home} />
      <Route path="/about" render={ () => <About title='About' /> } />
      <Route path="/teachers" component={Teachers} />
      <Route path="/courses" component={Courses} />
    </div>
  </BrowserRouter>
);
```

Router

router-internal

Route

Route is the component that you will use to define your routes. Here are the basic props:

- **path** - This is the path to match to. Begins with '/'
- **exact** - Boolean, just include if you want an exact match
- **component** or **render** - either pass a component or you can pass a anonymous function.

If you need to pass props to a component, then you would need to use the "render" option in Route:

```
<Route path="/about" render={() => <About title="my About" /> } />
```

Dynamic Routes

You can also make routes dynamic. For example, if you wanted to have a path to an item include the item id or name as part of the path, you can use a colon in front of the dynamic part of the path:

```
<Route path="/items/:itemId" ... />
```

The above path would then match if the url was /item/12334 or anything after the /item/. When using dynamic routes, the component or render function gets passed a prop called match with a params object. In this object you can access the defined dynamic route. In the above case, it would be *match.params.itemId* The best way to access or go to these dynamic routes is to use the Link component.

```
<Link to=`/items/${itemId}` />
```

You could map through an array to create multiple links that would match with multiple pages with each page getting an itemId that could then be rendered. Here is an example:

```
    <Route
       path='/items/:itemId'
       render={ ({match}) => {
           const item = lookupAndGetItem(itemId); // or use find on an array
           return (<Item item={item} />);
          }}
    />
```

Link and NavLink

Link and NavLink components allow you to create links to specific routes. The main difference between the two is that NavLink will give an "active" class name to the last NavLink clicked.

```
    <ul className="main-nav">
      <li><NavLink exact to="/">Home</NavLink></li>
      <li><NavLink to="/about">About</NavLink></li>
      <li><NavLink to="/teachers">Teachers</NavLink></li>
      <li><NavLink to="/courses">Courses</NavLink></li>
    </ul>
```

If you want the active class name to be something other than "active", you can pass the **activeClassName** prop and send your custom class name through. May be useful when each link that is active looks different than the others.

You can even pass the active style if you want to override some of the active class name's properties. This is done with the **activeStyle** prop. You will pass this prop a style object. Note that the **active** class name is still applied and then this style object will override whatever you pass or simply add more style to the link when active.

```
      <li><NavLink to="/teachers" activeStyle={{ background: "blue"}}>Teachers</NavLink>
</li>
      <li><NavLink to="/courses" activeClassName="myActiveClass">Courses</NavLink></li>
```

Redirect

The Redirect component allows you to programatically go to a route. Keep in mind that, by default, the current location is replaced by the new one.

If you want to push a new entry onto the history instead, pass a push prop as well.

```
//add to history
<Redirect to="/courses/html" push/>
```

When redirecting from nested routes, you may want to take a different approach with the Redirect component. For example, if you go to a page with nested routes and want to have that page when first loaded "/courses" show one of the routes by default, you will want to put the Redirect **inside** a Route component of its own:

```
    <Route exact path="/courses" render={() => <Redirect to="/courses/html" />} />
    <Route path="/courses/html" component={HTML} />
    <Route path="/courses/css" component={CSS} />
    <Route path="/courses/javascript" component={Javascript} />
```

Note the use of **exact** and then using the **render** prop to render our Redirect component.

Redirect using the history prop

There will be times when you will not be able to render a Redirect component, but still need to navigate to a new path.

In these cases you can use the history prop passed to all components rendered from a Route component.

```
class Home extends React.Component {
    handleSubmit = (e) => {
        e.preventDefault();
        let something = e....value;
        this.props.history.push(newPath);
    }
    render() {
        <form onSubmit={this.handleSubmit}>
            ...
        </form>
    }
}
```

match prop

Each route that we redirect or link to gets a set of props that we can use in various ways.

[React Router match doc]

A match object contains information about how a <Route path> matched the URL. match objects contain the following properties:

- params - (object) Key/value pairs parsed from the URL corresponding to the dynamic segments of the path
- isExact - (boolean) true if the entire URL was matched (no trailing characters)
- path - (string) The path pattern used to match. Useful for building nested s
- url - (string) The matched portion of the URL. Useful for building nested s

You'll have access match objects in various places:

- Route component as this.props.match

- Route render as ({ match }) => ()
- Route children as ({ match }) => ()
- withRouter as this.props.match
- matchPath as the return value

If a Route does not have a path, and therefore always matches, you'll get the closest parent match. Same goes for withRouter.

```
import React from 'react';
import { Route, NavLink, Redirect } from 'react-router-dom';
import HTML from './courses/HTML';
import CSS from './courses/CSS';
import Javascript from './courses/JavaScript';

const Courses = ({match, }) => (
  <div className="main-content courses">
    <div className="course-header group">
      <h2>Courses</h2>
      <ul className="course-nav">
        <li><NavLink to={`${match.url}/html`}>HTML</NavLink></li>
        <li><NavLink to={`${match.url}/css`}>CSS</NavLink></li>
        <li><NavLink to={`${match.url}/javascript`}>JavaScript</NavLink></li>
      </ul>
    </div>

    <Route exact path={`${match.path}`} render={() => <Redirect to=
{`${match.path}/html`} />} />
    <Route path={`${match.path}/html`} component={HTML} />
    <Route path={`${match.path}/css`} component={CSS} />
    <Route path={`${match.path}/javascript`} component={Javascript} />
  </div>
);

export default Courses;
```

By doing the above, if you ever change the route that links to this component, you won't break your nested routes.

Displaying 404 Error Routes using Switch

Switch React Router V4 Docs

If you have a list of Route components, without Switch it is very possible that multiple routes will match and multiple routes will be renedered.

This is by design so advanced stuff could be done with Route components.

If, however, you wrap a bunch of Route components in a Switch component, then the first Route that matches will render and that is it.

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

<Switch>
  <Route exact path="/" component={Home}/>
  <Route path="/about" component={About}/>
  <Route path="/:user" component={User}/>
  <Route component={NoMatch}/>
</Switch>
```

Creating Dynamic Routes Paths

You can create Route parameters by putting a colon in front of the parameter:

```
<Route path="/teachers/:name" component={Teachers} />
```

The **:name** is the parameter and it will be made available to the Teachers component via the passed route props.

![]](https://dl.dropboxusercontent.com/s/gykblaxe43nw603/reacrouter-props-match.png)

You can access this parameter via **props.match.params.paramName**.

You can even add multiple parameters in a single path:

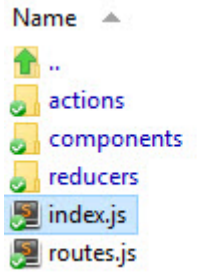
```
<Route path="/teachers/:topic/:fname-:lname" component={Teachers} />

//Teachers component
const Teachers = ({ match }) => {
    let name = `${match.params.fname} ${match.params.lname}`;
    let topic = match.params.topic;
}
```

React Router v4 URL Params

React-Router v2

```
npm install –save react-router
```

Allows us to use the "route" in the URL to go to different sets of components. For example, http://myWebSite.com(http://mywebsite.com) is the root route "/" and http://myWebSite.com/firstRoute(http://mywebsite.com/firstRoute) would be another route. The main Route component will be set up in the base or index.js code.



In your base component (index.js) you will have something like this:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { createStore, applyMiddleware } from 'redux';
import { Router, browserHistory } from 'react-router';

import reducers from './reducers';
import routes from './routes';

const createStoreWithMiddleware = applyMiddleware()(createStore);
ReactDOM.render(
    <Provider store={createStoreWithMiddleware(reducers)}>
        <Router history={browserHistory} routes={routes}/>
    </Provider>
    , document.querySelector('.container'));
```

The above example also has redux, but you can see that the Router component has history and routes props. The history prop can take either "browserHistory", "hashHistory" and something else I can't remember. The browser history means that it will be looking for routes immediately after the /. The hash history will be looking for routes after a /#. You will also set up a routes.js files that will define all the routes.

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

    import App from './components/app';

    const Greeting = () => {
        return <div> Hit there </div>;
    };

    export default (
    <Route path="/" component={App}>
        <Route path="greet" component={Greeting} />
        <Route path="greet2" component={Greeting} />
        <Route path="greet3" component={Greeting} />
    </Route>
    );
```

You will not ever have a component in your routes.js file, however it is here to show how you can have children under the main component. The "/" path means that the App component will show as the root. Since the App ("/") route has children, the App will also show even when on the children routes. One other very important item you will need if you have a route with child components/routes, is a reference to the this.props.children prop in the parent component/route. This means that in the App component we need something like this:

```
import React, { Component } from 'react';

export default class App extends Component {
    render() {
        return (
            <div>
            Main App
            {this.props.children}
            </div>
        );
    }
}
```

This means that when you are at the root, you will only see "Main App", but when you go to "/greet", you will see "Main App" and the child component. Most of the time you will want to show your Main app component and another component when the root route is selected. To do this you will use the IndexRoute react-router component.

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

import App from './components/app';
import PostsIndex from './components/posts_index';

export default (
    <Route path="/" component={App}>
        <IndexRoute component={PostsIndex} />
        <Route path="greet" component={Greeting} />
        <Route path="greet2" component={Greeting} />
        <Route path="greet3" component={Greeting} />
    </Route>
);
```

Note that when "/greet" is chosen, you will no longer see the IndexRoute component.

Navigating To New Pages

When you want a user to be able to navigate to a new page via a link, you can use react routers **Link** component.

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

class PostsIndex extends React.Component {
    render() {
        return (
            <div>
                <div className="text-xs-right">
                <Link to="posts/new" className="btn btn-primary"> Add Post </Link>
            </div>
                list of blog posts
            </div>
        );
    }

}
```

Many time you will want to programmatically go to a new route. This may be handled differently depending on the type of history you are using, but here are a few that I know of:

hashHistory.push('/todo');

Also, note, that if you are in a middleware function, you most likely will have a replace function as a parameter. This can be used to move to a new route. See Middleware Page(#RRM)

Using Context for Router Info

There are times when you will want to have Router information (like the push function) in a component that is far away from the Router component that encloses your app. You can use React Context for this. I'm not sure if it would be better to import something from react-router versus doing the following, but, whatever works.

```
class test extends React.Component {
//Must first declare that we want to receive context from React
    static contextTypes = {
        router: React.PropTypes.object
    }

    componentWillMount() {
    //using the router push method when component mounts
        if (something) {
            this.context.router.push('/');
        }
    }

    componentWillUpdate(nextProps) {
        //Need to check after the initial mount, so that when
        //component gets rerendered we know if we need to do something.
        if (nextProps.something) {
            this.context.router.push('/');
        }
    }
}
```

React-Router Middleware

Middleware allows you to do something before react-router does something. This example is for when you want to check if a user is logged in before taking them to a route:

```
//React Router middleware below.
var requireLogin = (nextState, replace, next) => {
    //firebase.auth().currentUser -> null means not logged in, else returns current user
    if (!firebase.auth().currentUser) {
        replace('/');
    }
        next();
    };

// <TodoApp />,
ReactDOM.render(
    <Provider store={store}>
        <Router history={hashHistory}>
            <Route path="/" component={Main}>
            <IndexRoute component={AppLogin} />
            <Route path="todo" component={TodoApp} onEnter={requireLogin} />
            </Route>
        </Router>
    </Provider>,
    document.getElementById('app')
);
```

The **requireLogin** function is the middleware. We can inject this into any of react-router's routes. Here we are using it on the TodoApp components route. **onEnter=requireLogin** is how we inject this middleware. Now anytime this route is called, first react-router will run the **requireLogin()** function.