Usage with React Router · Redux

Usage with React Router

So you want to do routing with your Redux app. You can use it with React Router. Redux will be the source of truth for your data and React Router will be the source of truth for your URL. In most of the cases, **it is fine** to have them separate unless if you need to time travel and rewind actions that triggers the change URL.

Installing React Router

`react-router` is available on npm . This guides assumes you are using `react-router@^2.7.0`.

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

Configuring the Fallback URL

Before integrating React Router, we need to configure our development server. Indeed, our development server may be unaware of the declared routes in React Router configuration. For example, if you access `/todos` and refresh, your development server needs to be instructed to serve `index.html` because it is a single-page app. Here's how to enable this with popular development severs.

> **Note on Create React App**
>
> If you are using Create React App, you won't need to configure a fallback URL, it is automatically done.

Configuring Express

If you are serving your `index.html` from Express:

```
app.get('/*', (req,res) => {
  res.sendfile(path.join(__dirname, 'index.html'))
})
```

Configuring WebpackDevServer

If you are serving your `index.html` from WebpackDevServer: You can add to your webpack.config.dev.js:

```
  devServer: {
    historyApiFallback: true,
  }
```

Connecting React Router with Redux App

Along this chapter, we will be using the Todos example. We recommend you to clone it while reading this chapter.

First we will need to import `<Router />` and `<Route />` from React Router. Here's how to do it:

```
import { Router, Route, browserHistory } from 'react-router';
```

In a React app, usually you would wrap `<Route />` in `<Router />` so that when the URL changes, `<Router />` will match a branch of its routes, and render their configured components. `<Route />` is used to declaratively map routes to your application's component hierarchy. You would declare in `path` the path used in the URL and in `component` the single component to be rendered when the route matches the URL.

```
const Root = () => (
  <Router>
    <Route path="/" component={App} />
  </Router>
);
```

However, in our Redux App we will still need `<Provider />`. `<Provider />` is the higher-order component provided by React Redux that lets you bind Redux to React (see Usage with React).

We will then import the `<Provider />` from React Redux:

```
import { Provider } from 'react-redux';
```

We will wrap `<Router />` in `<Provider />` so that route handlers can get access to the `store`.

```
const Root = ({ store }) => (
  <Provider store={store}>
    <Router>
      <Route path="/" component={App} />
    </Router>
  </Provider>
```

```
);
```

Now `<App />` component will be rendered if the URL match '/'. Additionally, we will add the optional `(:filter)` parameter to `/`, we will need it further below when we will try to read the parameter `(:filter)` from the URL.

```
<Route path="/(:filter)" component={App} />
```

You will probably want to remove the hash from URL (e.g: `http://localhost:3000/#/?_k=4sbb0i`). For doing this, you will need to also import `browserHistory` from React Router:

```
import { Router, Route, browserHistory } from 'react-router';
```

and pass it to the `<Router />` in order to remove the hash from the URL:

```
    <Router history={browserHistory}>
      <Route path="/(:filter)" component={App} />
    </Router>
```

Unless you are targeting old browsers like IE9, you can always use `browserHistory`.

`components/Root.js`

```
import React, { PropTypes } from 'react';
import { Provider } from 'react-redux';
import { Router, Route, browserHistory } from 'react-router';
import App from './App';

const Root = ({ store }) => (
  <Provider store={store}>
    <Router history={browserHistory}>
      <Route path="/(:filter)" component={App} />
    </Router>
  </Provider>
);

Root.propTypes = {
  store: PropTypes.object.isRequired,
};

export default Root;
```

Navigating with React Router

React Router comes with a component that let you navigate around your application. We can use it in our example and change our container `<FilterLink />` component so we can change the URL using `<FilterLink />. The`activeStyle={}` property lets you apply a style on the active state.

`containers/FilterLink.js`

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

const FilterLink = ({ filter, children }) => (
  <Link
    to={filter === 'all' ? '' : filter}
    activeStyle={{
      textDecoration: 'none',
      color: 'black'
    }}
  >
    {children}
  </Link>
);

export default FilterLink;
```

`containers/Footer.js`

```
import React from 'react'
import FilterLink from '../containers/FilterLink'

const Footer = () => (
  <p>
    Show:
    {" "}
    <FilterLink filter="all">
      All
    </FilterLink>
    {", "}
    <FilterLink filter="active">
      Active
    </FilterLink>
    {", "}
```

```
    <FilterLink filter="completed">
      Completed
    </FilterLink>
  </p>
);

export default Footer
```

Now if you click on `<FilterLink />` you will see that your URL will change from `'/complete'`, `'/active'`, `'/'`. Even if you are going back with your browser, it will use your browser's history and effectively go to your previous URL.

Reading From the URL

Currently, the todo list is not filtered even after the URL changed. This is because we are filtering from `<VisibleTodoList />`'s `mapStateToProps()` is still binded to the `state` and not to the URL. `mapStateToProps` has an optional second argument `ownProps` that is an object with every props passed to `<VisibleTodoList />`

`components/App.js`

```
const mapStateToProps = (state, ownProps) => {
  return {
    todos: getVisibleTodos(state.todos, ownProps.filter) // previously was
getVisibleTodos(state.todos, state.visibilityFilter)
  };
};
```

Right now we are not passing anything to `<App />` so `ownProps` is an empty object. To filter our todos according to the URL, we want to pass the URL params to `<VisibleTodoList />`.

When previously we wrote: `<Route path="/(:filter)" component={App} />`, it made available inside `App` a `params` property.

`params` property is an object with every param specified in the url. *e.g:* `params` *will be equal to* `{ filter: 'completed' }` *if we are navigating to* `localhost:3000/completed`*. We can now read the URL from* `<App />`.

Note that we are using ES6 destructuring on the properties to pass in `params` to `<VisibleTodoList />`.

`components/App.js`

```
const App = ({ params }) => {
  return (
    <div>
      <AddTodo />
      <VisibleTodoList
        filter={params.filter || 'all'}
      />
      <Footer />
    </div>
  );
};
```

Next Steps

Now that you know how to do basic routing, you can learn more about React Router API

> **Note About Other Routing Libraries**
>
> *Redux Router* is an experimental library, it lets you keep entirely the state of your URL inside your redux store. It has the same API with React Router API but has a smaller community support than react-router.
>
> *React Router Redux* creates binding between your redux app and react-router and it keeps them in sync. Without this binding, you will not be able to rewind the actions with Time Travel. Unless you need this, React-router and Redux can operates completely apart.