Redux Setup

Initial Redux Configuration

CRA gave us a basic application template to start with. We now need to set up the initial Redux store and reducers, make the store available to our component tree, and make sure that it's being used. Let's jump right in and walk through the pieces.

> **Commit 44a18c4: Add initial Redux configuration and sample component**

First, we need to create our Redux store instance.

store/configureStore.js

```
import {createStore, applyMiddleware, compose} from "redux";

import thunk from "redux-thunk";

import rootReducer from "../reducers/rootReducer";

export default function configureStore(preloadedState) {
    const middlewares = [thunk];
    const middlewareEnhancer = applyMiddleware(...middlewares);

    const storeEnhancers = [middlewareEnhancer];

    const composedEnhancer = compose(...storeEnhancers);

    const store = createStore(
        rootReducer,
        preloadedState,
        composedEnhancer
    );
```

```
    return store;
}
```

I like to keep my store setup logic in a `configureStore` function that can be further improved as time goes on. I also try to keep the setup for each piece of the configuration process separate, so it's easy to follow what's going on, instead of cramming it into one big nested function call.

We need to add the `redux-thunk` middleware to our store. Right now it's the only middleware we're using, but we'll put it into a `middlewares` array to make it easier to add other middleware to the setup process down the road. Similarly, the only store enhancer we're using right now is `applyMiddleware`, but we'll do the same thing for clarity. Finally, we pass our three arguments to `createStore` and return the new store instance.

reducers/testReducer.js

```
const initialState = {
    data : 42
};

export default function testReducer(state = initialState, action) {
    return state;
}
```

reducers/rootReducer.js

```
import {combineReducers} from "redux";

import testReducer from "./testReducer";

const rootReducer = combineReducers({
    test : testReducer,
});

export default rootReducer;
```

Our initial reducers just pass along some test data so we can verify things are working.

Next, we use the React-Redux library and render a `<Provider>` around our root application component. That makes the Redux store accessible to any

Redux-connected React component in the tree.

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import {Provider} from "react-redux";

import App from './App';
import './index.css';

import configureStore from "./store/configureStore";
const store = configureStore();

ReactDOM.render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById('root')
);
```

Finally, we add a simple test component, connect it to Redux, and render it in our App component to verify that everything's hooked up properly:

SampleComponent.jsx

```
import React, {Component} from "react";
import {connect} from "react-redux";

const mapState = state => ({
    data : state.test.data
});

class SampleComponent extends Component {
    render() {
        const {data} = this.props;

        return (
            <div>
                Data from Redux: {data}
            </div>
        );
    }
}
```

```
export default connect(mapState)(SampleComponent);
```

App.js

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

import SampleComponent from "./SampleComponent";

class App extends Component {
    render() {
        return (
            <div className="App">
                <div className="App-header">
                    <img src={logo} className="App-logo" alt="logo"/>
                    <h2>Project Mini-Mek</h2>
                </div>
                <SampleComponent />
            </div>
        );
    }
}

export default App;
```

We should now see the text “Data from Redux: 42” in our page, right below the header bar.

Adding the Redux DevTools Extension

One of the original reasons for Redux’s creation was the goal of **“time-travel debugging”**: the ability to see the list of dispatched actions, view the contents of an action, see what parts of the state were changed after the action was dispatched, view the overall application state after that dispatch, and step back and forth between dispatched actions. Dan Abramov wrote the original Redux DevTools toolset, which showed that information with the UI as a component within the page.

Since then, Mihail Diordiev has built the **Redux DevTools Extension**, a browser extension that bundles together the core Redux DevTools logic and several community-built data visualizers as a browser extension, and adds a bunch of useful functionality on top of that. Connecting the DevTools

Extension to your store requires a few extra checks, so there's now a package

available that encapsulates the process needed to hook up the DevTools Extension to your store.

We'll add the helper package with `yarn add redux-devtools-extension`, then make a couple small changes to the store setup logic:

> **Commit d80abff: Add Redux DevTools Extension**

> **Commit b88a813: Use DevTools Extension when creating the store**

store/configureStore.js

```
- import {createStore, applyMiddleware, compose} from "redux";
+ import {createStore, applyMiddleware} from "redux";
+ import { composeWithDevTools } from 'redux-devtools-extension/developmen
tOnly';


- const composedEnhancer = compose(...storeEnhancers);
+ const composedEnhancer = composeWithDevTools(...storeEnhancers);
```

This should add the DevTools enhancer to our store, but only when we're in development mode. With that installed, the Redux DevTools browser extension can now view the contents of our store and the history of the dispatched actions.