Adding Hot Module Replacement

Out of the box, `create-react-app` will watch your files on disk, and whenever you save changes, it will recompile the application and reload the entire page. That's nice, but we can actually set up some improvements on that. In particular, the Webpack build tool used by `create-react-app` supports a feature known as **Hot Module Replacement**, or HMR, which can hot-swap the newly compiled versions of files into your already-open application. That lets us see our changes much faster. This works great with a React app, and even better in combination with Redux, since we can reload our component tree but still keep the same application state as before.

Using HMR requires adding some specific options to a Webpack configuration. Fortunately, CRA already has those options turned on in its Webpack config, so every CRA app has HMR enabled. From there, we need to use the `module.hot` API that Webpack provides to actually listen for changes to specific files, re-import the new versions, and use the new versions in our app. We can do that for both our React component tree and our Redux reducers.

Component Hot Reloading

The first step is to use HMR to add reloading for our React component tree. The best place to do this is in `index.js`, which already calls `ReactDOM.render()`.

> **Commit 4d70273: Configure Hot Module Replacement for the React component tree**

index.js

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

```
// Save a reference to the root element for reuse
const rootEl = document.getElementById("root");

// Create a reusable render method that we can call more than once
let render = () => {
    // Dynamically import our main App component, and render it
    const App = require("./App").default;

    ReactDOM.render(
        <Provider store={store}>
            <App />
        </Provider>,
        rootEl
    );
};

if(process.env.NODE_ENV !== "production") {
    if(module.hot) {
        // Support hot reloading of components.
        // Whenever the App component file or one of its dependencies
        // is changed, re-import the updated component and re-render it
        module.hot.accept("./App", () => {
            setTimeout(render);
        });
    }
}

render();
```

We start by extracting the core rendering logic into a separate function. That way we can call it once on startup to render the application normally, but also call it again whenever there's edits during development.

Webpack's `module.hot` API should only be available in development, so we check to see if that exists so that we don't add HMR when the code runs in production. We also wrap it in a check for `process.env.NODE_ENV` as well. This is a standard approach used to remove dev-only code from the final production build. If the comparison is false, then Webpack will not include the dead code when the app bundle is built.

We use `module.hot.accept()` to add a callback that will run whenever the `App.js` file is changed. File change events "bubble up" if they're not handled, so listening for changes to `App.js` should catch edits from *any* file imported by our component tree.

Finally, we re-run the render function whenever there's a changed file version, re-import the `App` component to get the latest version, and re-render using that updated component.

With this in place, editing one of our components (such as changing the text in `SampleComponent.jsx`) should now just reload the component tree, rather than reloading the entire page.

Reducer Hot Reloading

We can also configure our project to hot-reload our reducers as well. Right now, if we edit the initial state in our `testReducer.js` from `data : 42` to `data : 123`, we'll see the whole page reload, just like it did for component edits before we added the hot reloading. We need to listen for updates to the root reducer file, re-import it, and then replace the old reducer in the store with the new one.

> **Commit 2d8bc29: Configure Hot Module Replacement for the reducer logic**

store/configureStore.js

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

+   if(process.env.NODE_ENV !== "production") {
+       if(module.hot) {
+           module.hot.accept("../reducers/rootReducer", () =>{
+               const newRootReducer = require("../reducers/rootReducer").
default;
+               store.replaceReducer(newRootReducer)
+           });
+       }
+   }

    return store;
```

Like with component HMR, we only do this check if we’re running in development. We listen for changes to the root reducer file, re-import it, and tell the store to swap out the reducer function it’s using to update the state.

With that done, editing that default `state.test.data` value should show the updated value on the screen almost immediately, without having to refresh the entire page.