Handling Tab State with Redux

While working on the new tab panel contents, a problem becomes noticeable: because the “current tab” value is stored as state in the `<TabBarContainer>` component, reloading the component tree resets the selected tab when the component instance gets wiped out. This is where Redux can help us, by moving our state outside the component tree. Fortunately, because we kept the tab state in the `<TabBarContainer>` component, we can replace the local component state version with one that pulls the value from Redux instead. The feature folder will contain the standard action constants, action creators, and reducers, as well as the selector functions to encapsulate looking up this piece of state. For simplicity, we’ll just look at the reducer and the new version of `<TabBarContainer>`:

> **Commit 6660974: Rewrite tabs handling to be driven by Redux**

features/tabs/tabsReducer.js

```
import {createReducer} from "common/utils/reducerUtils";

import {TAB_SELECTED} from "./tabsConstants";

const initialState = {
    currentTab : "unitInfo",
};

export function selectTab(state, payload) {
    return {
        currentTab : payload.tabName,
    };
}

export default createReducer(initialState, {
    [TAB_SELECTED] : selectTab,
});
```

features/tabs/TabBarContainer.jsx

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

import TabBar from "common/components/TabBar";

import {selectCurrentTab} from "./tabsSelectors";
import {selectTab} from "./tabsActions";

const mapState = (state) => {
    const currentTab = selectCurrentTab(state);

    return {currentTab};
}

const actions = {onTabClick : selectTab};

export default connect(mapState, actions)(TabBar);
```

A few things to note here. We're using one of the umpteen million `createReducer` utility functions out there, which lets you define separate reducer functions and a a lookup table instead of switch statements. We're also using the object shorthand syntax that `connect` supports for the `mapDispatch` argument, allowing you to pass in an object full of action creator functions to be bound up, instead of writing an actual `mapDispatch` function yourself.

In addition, the `TabBarContainer.jsx` file is now somewhat unnecessary. It no longer actually has a component in it - it's just importing a "plain" component, creating the connected wrapper component, and exporting the wrapper. This brings up one other point about file structure that's worth discussing.

Container and Component Structuring

The phrase "container component" simply means "any component whose primary job is to store state or fetch data from somewhere, and pass that to its children". The original version of `<TabBarContainer>` was a "container", because it was storing the current tab name in its own state and passing that down, without rendering any UI itself. The wrapper components generated by `connect()` are *also* “container components”, because their primary job is to

extract data from the Redux store and pass that data to the “plain” components they are wrapping.

At the moment, we have the `<TabBarContainer>` and `<TabBar>` components being defined in separate files. However, it’s very unlikely that we’re going to actually need to reuse the “plain” `<TabBar>` component again in our application, and this goes along with the general trend that I’ve seen. I’d say that the vast majority of the time, a given component type will only be connected once in an application. However, it’s also possible that a given component may or may not be connected as app development continues. This is one of the reasons why I dislike separating connected and unconnected components into separate `/component` and `/container` folders. If you’ve chosen to keep component definitions and connection definitions in separate files, those are now in separate folders. If you’ve opted to define both the component and its connection in the same file, now you end up having to move component files into the `/containers` folder when you decide to connect it.

Because of this, I find that the most straightforward approach is to define plain components and their connections in the same file, and organize them with “feature folders”. That way, you’re less likely to need to shuffle files around just to try to keep up with an organizational pattern.

So, we’ll do one last set of edits here. We’ll move the connection logic into `TabBar.jsx` itself, and export the connected wrapper component as the default.

> **Commit 022b24e: Connect TabBar in its own file, instead of TabBarContainer.js**

features/tabs/TabBar.jsx

```
import React from "react";
import {connect} from "react-redux";
import {Menu} from "semantic-ui-react";

import ToggleDisplay from "common/components/ToggleDisplay"
import ToggleDisplay from "common/components/ToggleDisplay

import Tab from "./Tab";


import {selectCurrentTab} from "./tabsSelectors";
import {selectTab} from "./tabsActions";

const mapState = (state) => {
    const currentTab = selectCurrentTab(state);

    return {currentTab};
}

const actions = {onTabClick : selectTab};

export const TabBar = (props) => {
    // Skip render logic, which hasn't changed
}

export default connect(mapState, actions)(TabBar);
```

This is the general pattern we'll follow going forward: "plain" components exported by name, and the connected wrapper components as the default exports.

```
import React from "react";

const ToggleDisplay = ({show, children}) => {
    const displayStyle = {
        display : show ? undefined : "none"
    };

    return (
        <span style={displayStyle}>
            {children}
        </span>
    );
}

export default ToggleDisplay;
```