Connecting List Components

Picking up where we left off, we can see that each of our main "panel" components are connected to Redux, as well as the TabBar component. However, none of the components within those panels are directly connected. Instead, we're passing data and action creators down as props from each panel to its children. It's perfectly fine to only have a few connected components, but as an app grows, this can become a pain point.

The Redux FAQ covers this topic in the question "Should I only connect my top component, or can I connect multiple components in my tree?". Quoting the answer:

> The current suggested best practice is to categorize your components as "presentational" or "container" components, and extract a connected container component wherever it makes sense:
>
> > Emphasizing "one container component at the top" in Redux examples was a mistake. Don't take this as a maxim. Try to keep your presentation components separate. Create container components by connecting them when it's convenient. Whenever you feel like you're duplicating code in parent components to provide data for same kinds of children, time to extract a container. Generally as soon as you feel a parent knows too much about "personal" data or actions of its children, time to extract a container.
>
> In fact, benchmarks have shown that more connected components generally leads to better performance than fewer connected components. In general, try to find a balance between understandable data flow and areas of responsibility with your components.

Following this idea, our next step will be to connect more individual components at a finer-grained level of detail.

Connecting the PilotDetails Component

We'll start with the `<PilotDetails>` component. Right now, the `<Pilots>` component is retrieving a list of all Pilot objects in its `mapState` function, plus the `currentPilot` ID value. Then, when it renders, it does a lookup to find which Pilot entry matches the selected ID, and passes that object to `<PilotDetails>`. We can connect `<PilotDetails>` directly, and remove that logic from `<Pilots>`.

This is a straightforward transformation. We'll add a `mapState` function to `<PilotDetails>`, look up the right Pilot object by ID if available, and return that entry. While we're at it, we'll also tweak the input components to be disabled, so that the user knows they can't actually interact with them.

> **Commit 9fbb8b9: Update PilotDetails to be connected**

features/pilots/PilotDetails.jsx

```
import React from "react";
+import {connect} from "react-redux";
import {Form, Dropdown} from "semantic-ui-react";

+import orm from "app/orm";
+import {selectCurrentPilot} from "./pilotsSelectors";

+const mapState = (state) => {
+    let pilot;
+
+    const currentPilot = selectCurrentPilot(state);
+
+    const session = orm.session(state.entities);
+    const {Pilot} = session;
+
+    if(Pilot.hasId(currentPilot)) {
+        pilot = Pilot.withId(currentPilot).ref;
+    }
```

```
+
+    return {pilot}
+}

// Omit component code for space

-export default PilotDetails;
+export default connect(mapState)(PilotDetails);
```

features/pilots/Pilots.jsx

```
    render() {
        const {pilots = [], selectPilot, currentPilot} = this.props;

-       const currentPilotEntry = pilots.find(pilot => pilot.id === curren
tPilot) || {}

// Omit irrelevant rendering code
-                           <PilotDetails pilot={currentPilotEntry} />
+                           <PilotDetails />
```

The `mapState` connection replaces the logic we had in `<Pilots>`.

Connecting the PilotsList Component

Next up is the `<PilotsList>` component. The `<Pilots>` component currently extracts the actual Pilot objects from the store, passes them as an array to `<PilotsList>`, and then each individual plain Pilot object is passed as a prop to the "presentational" list items. We *could* just move the current `mapState` logic from `<Pilots>` to `<PilotsList>` and leave it at that, but instead, we're going to implement **one of the most useful Redux techniques: a connected list that passes item IDs to connected list items**. Let's look at the implementation, then discuss some of the details of the specific approach we're using.

The `mapState` for `<PilotsList>` will need to return an array of IDs for all Pilot entries. `<PilotsList>` will then render its list of `<PilotsListRow>` components, and pass the appropriate pilot ID into each list item. Either the list or the list item will need to determine if that list item is currently selected. There's valid arguments either way, but since we're already passing a `selected` flag into each list item, we'll leave that in place.

> **Commit 871a01f: Update PilotsList to be connected**

features/pilots/PilotsList/PilotsList.jsx

```
+import {selectPilot} from "../pilotsActions";
+import {selectCurrentPilot} from "../pilotsSelectors";
+
+
+const mapState = (state) => {
+    const session = orm.session(state.entities);
+    const {Pilot} = session;
+
+    // Extract a list of IDs for each Pilot entry
+    const pilots = Pilot.all().toModelArray().map(pilotModel => pilotMode
l.getId());
+
+    const currentPilot = selectCurrentPilot(state);
+
+    // Return the list of pilot IDs and the current pilot ID as props
+    return {pilots, currentPilot};
+}
+
+// Make an object full of action creators that can be passed to connect
+// and bound up, instead of writing a separate mapDispatch function
+const actions = {
+    selectPilot,
+};
+

export class PilotsList extends Component {
    render() {
-      const {pilots, onPilotClicked, currentPilot} = this.props;
+       const {pilots = [], selectPilot, currentPilot} = this.props;

-       const pilotRows = pilots.map(pilot => (
+       const pilotRows = pilots.map(pilotID => (
            <PilotsListRow
-               pilot={pilot}
-               key={pilot.name}
-               onPilotClicked={onPilotClicked}
-               selected={pilot.id === currentPilot}
+               pilotID={pilotID}
+               key={pilotID}
+               onPilotClicked={selectPilot}
```

```
+                selected={pilotID === currentPilot}
             />
         ));
```

The `Model.getId()` method is useful if you don't happen to know the exact name of the ID field for a model type. Maybe it's actually `name`, or `guid`, or something else. We can declare the ID field name as part of the model declaration, and the `getId()` method will use that to look up the right field when asked.

There's a few other ways we could come up with the list of Pilot IDs. Since we *do* know the ID field name in the plain Pilot objects, we could do `Pilots.all().toRefArray().map(pilot => pilot.id)`. Or, if we wanted to bypass using Redux-ORM's API, we could directly access the `state.entities.Pilot.items` array, where Redux-ORM keeps a list of all Pilot IDs in the state.

pilots/PilotsList/PilotsListRow.jsx

```
+const mapState = (state, ownProps) => {
+    const session = orm.session(state.entities);
+    const {Pilot} = session;
+
+    let pilot;
+
+    if(Pilot.hasId(ownProps.pilotID)) {
+        const pilotModel = Pilot.withId(ownProps.pilotID);
+
+        // Access the underlying plain JS object using the "ref" field,
+        // and make a shallow copy of it
+        pilot = {
+            ...pilotModel.ref
+        };
+
+        // We want to look up pilotModel.mech.mechType.  Just in case the
+        // relational fields are null, we'll do a couple safety checks a
s we go.
+
+        // Look up the associated Mech instance using the foreign-key
+        // field that we defined in the Pilot Model class
+        const {mech} = pilotModel;
+
+        // If there actually is an associated mech, include the
+        // mech type's ID as a field in the data passed to the component
```

```
+        if(mech && mech.type) {
+            pilot.mechType = mech.type.id;

+        }
+    }
+
+    return {pilot};
+}
```

For `<PilotsListRow>`, we just copy over the lookup logic we had in `<Pilots>`, except that now we're only looking up one entry instead of all of them. Also, we're using the `pilotID` prop that the `<PilotsList>` parent component is passing down. The connected wrapper component for `<PilotsListRow>` makes all passed-in props available to `mapState` if we declare that `mapState` takes two arguments. By convention, the second argument is referred to as `ownProps`.

Note that when `mapState` is declared to take two arguments, it will be called more often. This is in case a change of passed-in props would result in a change to the values returned from `mapState`.

features/pilots/Pilots/Pilots.jsx

```
- // Delete the existing mapState function entirely

-export class Pilots extends Component {
+export default class Pilots extends Component {
    render() {
-        const {pilots = [], selectPilot, currentPilot} = this.props;

        return (
            <Segment>
                <Grid>
                    <Grid.Column width={10}>
                        <Header as="h3">Pilot List</Header>
-                       <PilotsList
-                           pilots={pilots}
-                           onPilotClicked={selectPilot}
-                           currentPilot={currentPilot}
-                       />
+                       <PilotsList />
                    </Grid.Column>

// Omit other rendering logic
```

```
-export default connect(mapState, actions)(Pilots);
```

With those changes in place, the `<Pilots>` component is no longer connected, and is actually now back to being entirely presentational. It renders several layout-related components, and two connected containers: `<PilotsList>` and `<PilotDetails>`.

Connecting the Mechs Components

We'll wrap up this section by applying the same sets of changes to the various components in the "Mechs" panel as well.

> **Commit 44f5150: Update MechDetails to be connected**

> **Commit 931e4dc: Update MechsList to be connected**