Code Cleanup

In Connected Lists, Forms, and Performance, we connected our lists directly to Redux, set up basic editing for the UnitInfo form, and added the ability to toggle "editing" mode for a selected Pilot. This time, we'll **look at some advanced techniques for managing form change events, implement editing for model entries, and use a custom reducer structure to handle feature logic**.

But, before we dive in to the real work for this section, we've got a few bits of cleanup that will help us later.

Handling Data Reloads

There's a problem with the current `entitiesReducer` logic. If we click the "Load Unit Info" button twice in a session, the reducer will try to load the same sample data again, and will throw errors because data with those IDs already exists in state. We can fix this by having the case reducer delete any existing entries before it loads in the new data.

> **Commit e5af624: Clear out existing models on load to avoid conflicts when reloading data**

app/reducers/entitiesReducer.js

```
    const {pilots, designs, mechs} = payload;

+   // Clear out any existing models from state so that we can avoid
+   // conflicts from the new data coming in if data is reloaded
+   [Pilot, Mech, MechDesign].forEach(modelType => {
+       modelType.all().toModelArray().forEach(model => model.delete());
+   });
```

We *could* potentially also resolve this by using the `Model.upsert()` method I mentioned earlier, but it's safest to just delete the old entries before we insert the new ones.

Cleaning Up the PilotDetails Component

Since we're going to eventually add callback functions to handle the inputs in `<PilotDetails>`, now is a good time to convert it from a functional component to a class component.

> **Commit f46122b: Convert PilotDetails to a class component**

We're also going to rework the `<PilotDetails>` form a bit. The "gunnery" and "piloting" fields are going to have a limited range of values, so we might as well make them dropdowns. Also, Semantic-UI-React's `<Form.Field>` component supports some useful shorthand syntax. You can pass a label string and a component type as a prop, and it will render them. It will also forward any unknown props onwards to the input component it's rendering. here's an example:

```
-<Form.Field name="rank" width={16}>
-   <label>Rank</label>
-   <Dropdown
-       fluid
-       selection
-       options={RANKS}
-       value={rank}
-       disabled={!canStopEditing}
-   />
-</Form.Field>
+<Form.Field
+   name="rank"
+   label="Rank"
+   width={16}
+   control={Dropdown}
+   fluid
+   selection
+   options={RANKS}
+   value={rank}
```

```
+   disabled={!canStopEditing}
+/>
```

The `name`, `label`, and `width` props are handled by the `<Form.Field>` component. It renders an instance of `<Dropdown>`, and passes it the remaining props. (Order doesn't matter here, I just wrote them with the Field's props first and the Dropdown's props last.)

I suppose it's a tossup which formatting you prefer. I can see arguments both ways. But, given that I already did the work, we'll go with the latter approach :)

> **Commit 7fb450e: Rework PilotDetails form layout for consistency**