Managing Draft Data for Forms

As mentioned in the last section, our current Pilot editing logic updates the actual data object for that Pilot in the store. Because both the list row and the edit form are referencing the same object, all edits in the form are immediately reflected in the list row. That looks kinda cool when you first see it, but it's not really a desired behavior. What if we wanted the user to be able to cancel the edits, or confirm they actually wanted to save the new values, or even reset the changes?

I faced this issue when working on my own application. It took me a while to figure out the approach I now use, although in hindsight, the idea sure *feels* obvious, and I'm kinda kicking myself for not working it out sooner. (In my defense, it was still early on in my first actual usage of Redux.) I also have to give a big shout-out and thanks to **Tommi Kaikkonen**, author of Redux-ORM, who took the time to listen to my use case, answer my questions, and help me work out the idea.

I summarized the basic idea in the **Structuring Reducers - Normalizing State Shape** section of the Redux docs. I'll quote the relevant section here:

> An application that does a lot of editing of entities might want to **keep two sets of "tables" in the state, one for the "current" item values and one for the "work-in-progress/draft" item values. When an item is edited, its values could be copied into the "work-in-progress" section, and any actions that update it would be applied to the "work-in-progress" copy**, allowing the editing form to be controlled by that set of data while another part of the UI still refers to the original version. "Resetting" the edit form would simply require removing the item from the "work-in-progress" section and re-copying the original data from "current" to "work-in-progress", while "applying" the edits would involve copying the values from the "work-in-progress" section to

> the “current” section.

This really goes back to one of the core concepts of both React and Redux: **whenever possible, represent your application’s behavior through state**. In this case, we want to have two different representations of the same item. So, **copy it, leave the original alone, and modify the copy**. (This also applies to non-relational form data as well.)

Adding the Draft State Slice

Implementing this “draft slice” capability will require a noticeable amount of work. We need to add this new “draft” state slice to our store, then write the reducer logic to actually copy items from our current `entities` slice into the “draft” slice.

The good news is that this first part is pretty easy. We’ve already got a small reducer in our core app folder that defines the `entities` slice. We just need to copy that reducer’s initial state handling so that the “tables” are created for us, and add it in to the root reducer.

> **Commit d93f3a0: Add an “editingEntities” state slice to store draft data**

app/reducers/editingEntitiesReducer.js

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

import orm from "app/ormm";
const defaultEditingEntities = orm.getEmptyState();

export default createReducer(defaultEditingEntities, {
});
```

This reducer will only define the initial shape of the state. We won’t add any actual update logic here - all that goes elsewhere.

app/reducers/rootReducer.js

```
import entitiesReducer from "./entitiesReducer";
+import editingEntitiesReducer from "./editingEntitiesReducer";
```

```
import tabReducer from "features/tabs/tabReducer";


const combinedReducer = combineReducers({
    entities : entitiesReducer,
+   editingEntities : editingEntitiesReducer,
    unitInfo : unitInfoReducer,
    pilots : pilotsReducer,
    mechs : mechsReducer,
    tabs : tabReducer,
});
```

We'll add `editingEntities` as another top-level slice in our state tree.

We'll also throw in a small utility function to simplify looking up a Model instance from a Session:

> **Commit cbf7fc3: Add a utility function to look up a model by type and ID**

common/utils/modelUtils.js

```
export function getModelByType(session, itemType, itemID) {
    const modelClass = session[itemType];
    const model = modelClass.withId(itemID);
    return model;
}
```

Finally, we'll create our initial empty "editing feature" reducer. Similar to how we defined the "generic entity CRUD" reducer, we'll add it to the end of the list of "top-level" reducers in our root reducer:

> **Commit 6a29f3b: Create an empty editing feature reducer**

features/editing/editingReducer.js

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

const editingFeatureReducer = createReducer({}, {
});
```

```
export default editingFeatureReducer;
```

app/reducers/rootReducer.js

```
import entityCrudReducer from "features/entities/entityReducer";
+import editingFeatureReducer from "features/editing/editingReducer";

const rootReducer = reduceReducers(
    combinedReducer,
    entityCrudReducer,
+   editingFeatureReducer,
);
```