Saving Form Edits

We can now edit a draft item, but when we click “Stop Editing”, the original item is still unchanged. We need to overwrite the values in the original entry with the updated values in the draft entry.

Updating Existing Models

Earlier, we wrote a function called `copyEntity`, which serializes the requested model (and its relations) to plain JS objects, and recreates them in the draft slice. We now need to reverse the process, but this gets a bit tricker. We already have an entry in the `entities` slice, so we can’t just call `createEntity()` on that slice. If we tried deleting it and immediately recreating it with the new values, Redux-ORM would also clear out all the relational fields that reference that same item, which would break things. **What we really need is a way to update the existing model “in-place”**.

This is the other part that Tommi Kaikkonen helped me figure out. **We can write custom `updateFrom()` methods on our Model classes, which take another model of the same type, and recursively update themselves and all their relations**. It’s basically the inverse operation of the sample `toJSON()` function I showed earlier.

> **Commit b31b7b4: Add updateFrom() methods to models**

features/pilots/Pilot.js

```
    toJSON() {
        return {...this.ref};
    }

+   updateFrom(otherPilot) {
+       this.update(otherPilot.ref);
```

```
+    }
}
```

Again, our current model classes don't have any complex relations, so the implementation is really simple. We just call the built-in `update()` method, and pass it the plain JS object that the other model wraps around (which, in our case, is stored in the other slice of state).

The process of updating a 1:many collection gets considerably more complicated, as you have to take into account adds, updates, and deletions. In my real app, I wound up writing a reusable helper function to encapsulate the somewhat complex logic. I'll skip showing that here, but as a summary, it:

- Compares the set of item IDs between the old collection and new collection and determines items that are added, updated, and deleted
- For all updated items, calls `oldModel.updateFrom(updatedModel)`
- For all added items, does the same serialize/`toJSON()`/`parse()` approach we've already seen to insert the item into the original slice of state
- For all deleted items, calls `oldModel.delete()`

Nothing incredibly original, just taking care of the diffing and bookkeeping.

Applying Changes to Entities

We're going to write another core logic function that can load a model from a "source" slice, load its equivalent from the "destination" slice, and tell the destination entry to update itself with the source entry. Then we'll write a case reducer that uses that function with `editingEntities` as the source and `entities` as the destination.

> **Commit fd80c81: Add logic to apply item edits to the "current data" slice**

features/editing/editingReducer.js

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

+import orm from "app/orm";

import {
```

```
    EDIT_ITEM_EXISTING,
    EDIT_ITEM_UPDATE,

+   EDIT_ITEM_APPLY,
    EDIT_ITEM_STOP,
} from "./editingConstants";

+import {getModelByType} from "common/utils/modelUtils";

import {
    readEntityData,
+    updateEntitiesState,
    updateEditingEntitiesState,
} from "./editingUtils";


+export function updateEditedEntity(sourceEntities, destinationEntities, p
ayload) {
+   // Start by reading our "work-in-progress" data
+   const readSession = orm.session(sourceEntities);
+
+   const {itemType, itemID} = payload;
+
+   // Look up the model instance for the requested item
+   const model = getModelByType(readSession, itemType, itemID);
+
+   // We of course will be updating our "current" relational data
+   let writeSession = schema.from(destinationEntities);
+
+   const ModelClass = writeSession[itemType];
+
+   if(ModelClass.hasId(itemID)) {
+       // Look up the original Model instance for the top item
+       const existingItem = ModelClass.withId(itemID);
+
+       if(existingItem.updateFrom) {
+           // Each model class should know how to properly update itsel
f and its
+           // relations from another model of the same type.  Ask the ori
ginal model to
+           // update itself based on the "work-in-progress" model, whic
h queues up a
+           // series of immutable add/update/delete actions internally
+           existingItem.updateFrom(model);
+       }
+   }
+
```

```
+   // The session contains our new "current" relational data
+   const updatedEntities = writeSession.state;

+   return updatedEntities;
+}
+
+

+export function editItemApply(state, payload) {
+   const entities = selectEntities(state);
+   const editingEntities = selectEditingEntities(state);

+   const updatedEntities = updateEditedEntity(editingEntities, entitie
s, payload);
+   return updateEntitiesState(state, updatedEntities);
+}

const editingFeatureReducer = createReducer({}, {
    [EDIT_ITEM_EXISTING] : editItemExisting,
    [EDIT_ITEM_UPDATE] : editItemUpdate,
+   [EDIT_ITEM_APPLY] : editItemApply,
    [EDIT_ITEM_STOP] : editItemStop,
});
```

The logic is fairly straightforward. For `updateEditedEntity()`, we create two separate Redux-ORM sessions, retrieve the models from each, call `existingItem.updateFrom(model)`, and return the immutably updated state as normal. The `editItemApply()` case reducer just selects the two state slices, calls `updateEditedEntity()`, and returns the updated root state.

Updating Pilot Save Logic

Our pilots reducer currently responds to the `PILOT_SELECT` action by resetting `isEditing : false`. That was fine when we were just showing and editing the existing entry, but now we have some additional steps going on. We really need to make sure that our editing data is cleaned up when another pilot is selected.

We're going to update the `selectPilot()` action creator to explicitly dispatch the "stop editing pilot" logic before it actually dispatches `PILOT_SELECT`. In the process, since we *know* that you can only ever edit the current Pilot, we can rework the thunks to retrieve the current pilot from state instead of passing it in as a parameter. (There's a very good discussion to be had about which approach is better, but we'll go ahead and make the changes for sake of

illustrating the idea.)

> **Commit d425bc8: Rework pilot logic to explicitly stop editing on selection**

features/pilots/pilotsReducer.js

```
        currentPilot : isSamePilot ? null : newSelectedPilot,
-       // Any time we select a different pilot, we stop editing
-       isEditing : false,
```

features/pilots/pilotsActions.js

```
+import {selectCurrentPilot, selectIsEditingPilot} from "./pilotsSelector
s";

export function selectPilot(pilotID) {
-   return {
-       type : PILOT_SELECT,
-       payload : {currentPilot : pilotID},
-   };
+   return (dispatch, getState) => {
+       const state = getState();
+       const isEditing = selectIsEditingPilot(state);
+
+       if(isEditing) {
+           dispatch(stopEditingPilot())
+       }
+
+       dispatch({
+           type : PILOT_SELECT,
+           payload : {currentPilot : pilotID},
+       });
+   }
}

-export function startEditingPilot(pilotID) {
+export function startEditingPilot() {
    return (dispatch, getState) => {
+       const currentPilot = selectCurrentPilot(getState());

-       dispatch(editExistingItem("Pilot", pilotID));
+       dispatch(editExistingItem("Pilot", currentPilot));
```

```
        dispatch({type : PILOT_EDIT_START});
    }

}

-export function stopEditingPilot(pilotID) {
+export function stopEditingPilot() {
    return (dispatch, getState) => {
+       const currentPilot = selectCurrentPilot(getState());

        dispatch({type : PILOT_EDIT_STOP});
-       dispatch(stopEditingItem("Pilot", pilotID));
+       dispatch(stopEditingItem("Pilot", currentPilot));
    }
}
```

From there, we can simply add in the “apply edits” action as part of the “stop editing” thunk:

> **Commit 6b13991: Save changes to pilot entries when editing is stopped**

features/pilots/pilotsActions.js

```
import {
    editExistingItem,
+   applyItemEdits,
    stopEditingItem
} from "features/editing/editingActions";

export function stopEditingPilot() {
    return (dispatch, getState) => {
        const currentPilot = selectCurrentPilot(getState());

        dispatch({type : PILOT_EDIT_STOP});
+       dispatch(applyItemEdits("Pilot", currentPilot));
        dispatch(stopEditingItem("Pilot", currentPilot));
    }
}
```

And now, **whenever we hit the “Stop Editing” button, our changes are saved, and the list entry should be updated with whatever changes we made in the form!**.