Generic Reducer Logic for Editing Entities

Before we create the actual core reducer logic, we'll add some utility functions to encapsulate behavior. We'll have a small function to update `state.entities`, and another to update `state.editingEntities`. We'll also create a small function to access a given Redux-ORM Model instance, and ask it to copy its entire contents into a plain JS object:

> **Commit 17d04ef: Add editing utility functions**

features/editing/editingUtilities.js

```
import orm from "app/orm";
import {getModelByType} from "common/utils/modelUtils";

export function updateEditingEntitiesState(state, updatedEditingEntitie
s) {
    return {
        ...state,
        editingEntities : updatedEditingEntities,
    };
}

export function updateEntitiesState(state, updatedEntities) {
    return {
        ...state,
        entities : updatedEntities,
    };
}

export function readEntityData(entities, itemType, itemID) {
    const readSession = orm.session(entities);

    // Look up the model instance for the requested item
    const model = getModelByType(readSession, itemType, itemID);
```

```
    const data = model.toJSON();


    return data;
}
```

(Yes, those two "update" functions are trivial, and not *really* needed here, but it's a small bit of encapsulation.)

As we saw earlier with code that used `parse()`, here we're expecting that all our Model instances will have a `toJSON()` method. Again, **that's something we're writing ourselves as a common convention, not anything that's built in to Redux-ORM**.

On that note, let's go ahead and write those `toJSON()` methods for our current model classes. Since the model classes are *very* simple, the methods will be trivial:

> **Commit f0cca7b: Add toJSON() methods for Redux-ORM model classes**

features/pilots/Pilot.js

```
    static parse(mechData) {
        return this.create(mechData);
    }

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

For now, all we're going to do is make a shallow copy of the actual plain JS object that's in the store. Once our data starts using more complex relations, this would be the place to recurse through all of the 1:1 and 1:many relations, and call `toJSON()` on them as well. **Since we don't have any classes like that right now, here's a quick example of what this might look like**:

```
class SomeHypotheticalModelWithRelations extends Model {
    toJSON() {}
        return {
```

```
            ...this.ref,
            someSingleRelation : this.someSingleRelation.toJSON(),

            someMultipleRelation : this.someMultipleRelation.withModels.ma
p(item => item.toJSON())
        };
    }
}
```

Nothing fancy - just translate the relational fields back into a nested plain JS object as appropriate.

With all that in place, it's time to figure out what our editing reducer actually needs to do.

Writing the Core Editing Reducer Logic

To start with, we need our editing reducer to handle three basic tasks:

- When we start editing an item, we need to copy it from the "current data" `entities` slice to the "draft data" `editingEntities` slice
- We need to be able to update the item in `editingEntities` with new values
- When we're done editing the item, we need to delete the copy from `editingEntities`, since it's not being used any more

We're going to define three initial actions for those tasks: `EDIT_ITEM_EXISTING`, `EDIT_ITEM_UPDATE`, and `EDIT_ITEM_STOP`.

Remember that our editing reducer is a "top-level" reducer, and receives the *entire* state tree as its `state` argument. That means it has access to both `state.entities` and `state.editingEntities`.

Looking back at our "generic entity CRUD" reducer, we can see that **we already have individual case reducer functions that know how to create, update, and delete an entity in a slice of state**. They're currently being used together to handle updates to the `state.entities` slice as a whole, but **we can import and reuse those individual case reducer functions as part of the editing reducer logic**.

I'll link the commit that implements the reducers for these three cases, and explain each piece in turn:

> **Commit f12a784: Add generic reducer logic for editing an entity**

features/editing/editingReducer.js

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

import {
    createEntity,
    updateEntity,
    deleteEntity
} from "features/entities/entityReducer";

import {
    EDIT_ITEM_EXISTING,
    EDIT_ITEM_UPDATE,
    EDIT_ITEM_STOP
} from "./editingConstants";

import {selectEntities} from "features/entities/entitySelectors";
import {selectEditingEntities} from "./editingSelectors";
import {
    readEntityData,
    updateEditingEntitiesState,
} from "./editingUtils";

export function copyEntity(sourceEntities, destinationEntities, payload) {
    const {itemID, itemType} = payload;

    const newItemAttributes = readEntityData(sourceEntities, itemType, ite
mID);
    const creationPayload = {itemType, itemID, newItemAttributes}

    const updatedEntities = createEntity(destinationEntities, creationPayl
oad);
    return updatedEntities;
}

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

    const updatedEditingEntities = copyEntity(entities, editingEntities, p
```

```
ayload);
    return updateEditingEntitiesState(state, updatedEditingEntities);

}

export function editItemUpdate(state, payload) {
    const editingEntities = selectEditingEntities(state);

    const updatedEditingEntities = updateEntity(editingEntities, payload);
    return updateEditingEntitiesState(state, updatedEditingEntities);
}

export function editItemStop(state, payload) {
    const editingEntities = selectEditingEntities(state);

    const updatedEditingEntities = deleteEntity(editingEntities, payload);
    return updateEditingEntitiesState(state, updatedEditingEntities);
}

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

export default editingFeatureReducer;
```

First, `copyEntity()` is an internal helper function, which isn't assigned to handle any specific cases. It's called with two different state slices, one as a source and the other as a destination, and the payload from an action. It uses the `readEntityData()` utility method we wrote to look up a Model instance and copy its entire contents to a plain JS object. Then, it uses the existing `createEntity()` case reducer to parse that object and create a copy of the destination slice with the new item "inserted into the tables".

I want to pause and re-emphasize that idea: **we're copying items by "serializing" the Models to plain objects, and then re-parsing them to create the duplicate Model instances**. The "parsing" concept is the same function we're calling when we load the data from our fake API - we're reusing that here as our means of insertion.

The `editItemExisting()` case reducer is used to start the editing process for an item that already exists in the "current data" `state.entities` slice. We use small selectors to retrieve the `state.entities` and `state.editingEntities`

slices, then pass them to `copyEntity()`. That returns us the newly updated

`editingEntities` slice, but we need to return that as part of the overall top-level state. We use the `updateEditingEntitiesState()` utility we wrote to do that.

Similarly, the `editItemUpdate()` case reducer reads the `editingEntities` slice, passes that to the existing `updateEntity` reducer, and returns the new top-level state with the updated `editingEntities` slice.

Finally, `editItemStop()` deletes the item out of the `editingEntities` slice, using the existing `deleteEntity()` case reducer to do the work.

Again, the idea here is that **we already have a set of "primitive", low-level reducer functions for manipulating entities in state, so we can reuse them as building blocks to create higher-level logic**. Because hey, it's all just functions!