Structuring Reducer Logic for Features

As mentioned previously, we're using a "feature-first" folder structure. There's tradeoffs with both "file-type-first" and "feature-first" approaches.

With "file-type-first", you know where to look for a certain type of code, and it's really easy to pull together all the slice reducers into a root reducer file. However, each feature gets split across several folders, and if multiple features need to interact with a specific slice of state, the files or folders for that slice of reducer logic will start to get awfully crowded.

With "feature-first", you know exactly where *all* the files for a given feature are, but the process of combining the reducer logic together can be a bit tricker, especially if you're using the standard approach of `combineReducers`. In particular, **what happens when multiple features need to interact with the same slice of state?**

In our case, we've created an `entities` slice containing the "tables" for our relational data entries. Because this is a central data location for our application, several different features are going to need to make updates to the `entities` slice. We *could* put all the reducer logic from the different features together into one giant `entitiesReducer`, but that would get ugly rather quickly. **What we really need is a way for each individual feature to separately apply updates to the `entities` slice.**

Looking Beyond `combineReducers`

I'll paraphrase some of the key concepts from the **Structuring Reducers** section of the Redux docs:

> It's important to understand that your entire application really only has **one single reducer function**: the function that you've passed into `createStore` as the first argument. It's good programming practice to

> take pieces of code that are very long or do many different things, and break them into smaller pieces that are easier to understand. Since a
>
> Redux reducer is just a function, the same concept applies. **You can split some of your reducer logic out into another function, and call that new function from the parent function**.
>
> In Redux it is very common to structure reducer logic by delegating to other functions based on slice of state. Redux refers to this concept as "reducer composition", and it is by far the most widely-used approach to structuring reducer logic. In fact, it's so common that Redux includes a utility function called `combineReducers()`, which specifically abstracts the process of delegating work to other reducer functions based on slices of state. However, **it's important to note that it is not the only pattern that can be used**.
>
> Once you go past the core use case for `combineReducers`, **it's time to use more "custom" reducer logic**, whether it be specific logic for a one-off use case, or a reusable function that could be widely shared. There's third-party reducer utilities that are available, or if none of the published utilities solve your use case, **you can always write a function yourself that does just exactly what you need**.

It's time to put some of these ideas into practice.

Using `reduceReducers`

If you happened to look at the `reducerUtils.js` file previously, you may have noted a function in there I haven't talked about yet: `reduceReducers()`. Here it is:

```
export function reduceReducers(...reducers) {
    return (previous, current) =>
        reducers.reduce(
            (p, r) => r(p, current),
            previous
        );
}
```

This was actually originally written by Andrew Clark, and is available as a package at https://github.com/acdlite/reduce-reducers . I swiped it and pasted it into `reducerUtils.js` to skip taking on another dependency.

it into `reducerUtils.js` to skip taking on another dependency.

`reduceReducers` is a nifty little utility. It lets us supply multiple reducer functions as arguments and effectively forms a pipeline out of those functions, then returns a new reducer function. If we call that new reducer with the top-level state, it will call the first input reducer with the state, pass the output of that to the second input reducer, and so on. (If I were more Functional-Programming-minded, I'd guess that there's probably a lot of similarities with `compose()`, but this is about as far as my mind goes in that direction. I'm sure someone will be happy to correct me or clarify things.)

Let's rework our root reducer to make use of `reduceReducers`:

> **Commit 7d7726d: Update root reducer to wrap around combined reducer**

app/reducers/rootReducer.js

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

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


const rootReducer = reduceReducers(
    combinedReducer,
);

export default rootReducer;
```

Our old "root reducer" is now the `combinedReducer`, and we pass that into `reduceReducers` to get the new root reducer.

It's important to note that **the combined reducer should be first in the pipeline, because that defines the shape of the state**. (See the Initializing State section of the Redux docs for more info on why and how the combined

State section of the Redux docs for more info on why and how the combined reducer defines the state shape.)

Writing a Higher Order Reducer

A "higher order reducer" is any function that takes a reducer as an argument, and returns a new reducer. That includes `combineReducers` and `reduceReducers`. Now, we're going to write one of our own.

If all the reducer functions given to `reduceReducers` are called with the top-level state object, but a given feature only needs to apply updates to one piece of the state, we're going to have a lot of repetitive logic as each feature reducer tries to apply its updates. In addition, `reduceReducers` will be calling these feature reducers will be called on every dispatched action, so it would be nice to ensure the feature reducers only respond if the action is something they care about.

Putting these ideas together, we can build a higher order reducer function called `createConditionalSliceReducer`:

> **Commit 5c37b11: Add createConditionalSliceReducer utility**

common/utils/reducerUtils.js

```
export function createConditionalSliceReducer(sliceName, fnMap) {
    // Create a reducer that knows how to handle one slice of state, wit
h these action types
    const sliceReducer = createReducer({}, fnMap);

    // Create a new wrapping reducer
    return (state, action) => {
        // Check to see if this slice reducer knows how to handle this act
ion
        if(fnMap[action.type]) {
            // If it does, pass the slice to the slice reducer, and updat
e the slice
            return {
                ...state,
                [sliceName] : sliceReducer(state[sliceName], action),
            };
        }
```

```
        // Otherwise, return the existing state unchanged
        return state;
    }
}
```

We've already seen that our `createReducer` utility takes a lookup table mapping action types to case handler reducer functions. Now, `createConditionalSliceReducer` takes the same lookup table, plus the name of a state slice to update. It uses `createReducer` to create a reducer that can handle a single slice, and returns a new reducer that checks incoming actions and only responds if any of the given case reducers know how to handle that action type.

Creating a Generic Entity Update Reducer

Looking at our Redux-ORM models, we can see that the basic process for updating a given model's values will be identical no matter what type of model it is. We need to create a `Session` instance, retrieve the right Model class for the item, look up the specific Model instance by ID, and then tell it to queue an update for the given fields. We can write a generic reducer to update any model instance by its type and ID.

We're going to create a new feature folder, `features/entities/`, and add the reducer logic there. We then need to create a top-level "feature reducer" function to execute that logic, and add the feature reducer to our root reducer.

> **Commit 08b7d9b: Add an "entity feature" reducer and a generic "entity update" reducer**

features/entities/entityReducer

```
import {ENTITY_UPDATE} from "./entityConstants";

import {createConditionalSliceReducer} from "common/utils/reducerUtils";

import orm from "app/orm";

export function updateEntity(state, payload) {
    const {itemType, itemID, newItemAttributes} = payload;
```

```
    const session = orm.session(state);
    const ModelClass = session[itemType];


    let newState = state;

    if(ModelClass.hasId(itemID)) {
        const modelInstance = ModelClass.withId(itemID);

        modelInstance.update(newItemAttributes);

        newState = session.state;
    }

    return newState;
}

const entityHandlers = {
    [ENTITY_UPDATE] : updateEntity,
};

const entityCrudFeatureReducer = createConditionalSliceReducer("entitie
s", entityHandlers);

export default entityCrudFeatureReducer;
```

The basic `updateEntity()` function should be straightforward. We extract the `itemType`, `itemID`, and `newItemAttributes` fields from the action payload. The `state` parameter in this case should be *just* the `entities` slice of our root state. We create a Redux-ORM Session instance from our "tables", retrieve the right Model class by name, look up the specific Model instance by ID, and create a new state object with the updates applied.

From there, we create a lookup table for the action types this module knows how to handle, and use the `createConditionalSliceReducer` utility to generate a new reducer that will only respond to the actions in the lookup table, and ensure that only the `entities` slice is updated.

app/reducers/rootReducer.js

```
import mechsReducer from "features/mechs/mechsReducer";

+import entityCrudReducer from "features/entities/entityReducer";

// Omit combined reducer
```

```
// Omit combined reducer

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

We then add the top-level entity "feature reducer" as another argument to `reduceReducers`, and make sure that it's *after* the combined reducer. Now, **we should be able to dispatch actions to update the contents of any one specific model instance in our state**.