Typical Usage

I primarily use Redux-ORM as a specialized “super-selector” and “super-immutable-update” tool. This means that I work with it in my selector functions, thunks, reducers, and `mapState` functions. Here’s some of the practices I’ve come up with.

Entity Selection

Because I consistently use the `ORM` singleton instance across my application, I’ve created a selector that encapsulates extracting the current `entities` slice and returning a `Session` instance initialized with that data:

```
import {createSelector} from "reselect";
import orm from "./schema";

export const selectEntities = state => state.entities;

export const getEntitiesSession = createSelector(
    selectEntities,
    entities => orm.session(entities),
);
```

Using that selector, I can retrieve a `Session` instance inside of a `mapState` function, and look up pieces of data as needed by that component.

This has a couple benefits. In particular, because many different `mapState` functions might be trying to do data lookups right in a row, only one `Session` instance should be created per store update, so this should be something of a performance optimization. Redux-ORM does offer a `createSelector()` function that is supposed to create optimized selectors that track which models were accessed, but I haven’t yet gotten around to actually trying that. I may look into it later when I do some perf/optimization passes on my own application.

On the other hand, since `Session` instances replace their `session.state` value when they apply updates, it's a bad idea to reuse the current `Session` from that selector if you need to do any kind of update operations. I always create new `Session` instances inside my reducers, and also in places like thunks or sagas where I might want to generate new items before including them in an action. I wrote a small selector to help with that process:

```
export function getUnsharedEntitiesSession(state) {
    const entities = selectEntities(state);
    return schema.from(entities);
}
```

Overall, I make it a point to keep all my components unaware of Redux-ORM's existence, and only pass plain data as props to my components. That means I usually will read the `model.ref` property to retrieve the real underlying JS object from the store, and include that as part of the return value in `mapState`.

Writing Entity-Based Reducers

Most of my entity-related reducers are based more around a specific action case rather than a certain Model class. Because of this, some of my reducers are fairly generic, and take an item type and an item ID as parameters in the action payload. As an example, here's a generic reducer for updating the attributes of any arbitrary Model instance:

```
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;
}
```

```
}
```

Not all my reducers are this generic - some of them do end up specifically referencing certain model types in specific ways. In some cases, I can build up higher-level functionality by reusing these generic building block reducers.

My reducers generally follow the same pattern: extract parameters from payload, create Session instance, apply updates to models and return the new state from `session.state`. There's admittedly some verbosity there, which I could probably abstract out a bit more if I wanted to, but the overall consistency and the simplification of the actual update logic is a big win in my book.

I've also written a couple small utilities that aid with the process of looking up a given model by its type and ID:

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

export function getModelIdentifiers(model) {
    return {
        itemID : model.getId(),
        itemType : model.getClass().modelName,
    };
}
```

Many of my actions contain `itemType` and `itemID` pairs in their payloads. Part of that is that I personally lean towards keeping my actions fairly minimal, with more work in the thunks *and* the reducers as necessary, and don't like blindly merging data from an action straight into my state.

Handling Iterative Updates

In Redux-ORM 0.8.x, processing multiple updates in a row could be a bit tricky, especially if later steps of the update process needed refer to some of the earlier results while those earlier update steps were still queued up and waiting to be applied. Fortunately, 0.9's changes to `Session` update behavior simplified things. The current `session.state` object is always the current underlying state values, and any Model instance that still represents an entry

underlying state values, and any Model instance that still represents an entry that exists in the store will return the correct values when you access its

fields. So, it's possible you may still have to be careful about sequencing update logic at times, but it's a lot simpler with 0.9 than it was with 0.8.