Core Concepts

Sessions

The Session class is used to interact with an underlying set of data. If you use the reducer generated by `createReducer(orm)`, Redux-ORM will create a Session instance internally. Otherwise, you can create Session instances by calling `orm.session(entities)` (which creates a Session that will apply updates immutably), or `orm.mutableSession(entities)` (which creates a Session that will directly mutate the provided data). Note that the "database tables" JS object you pass into `orm.session()` doesn't have to have come from the Redux store - you could easily use Redux-ORM outside of Redux, or pass in data that came from somewhere else.

A `Session` instance has a reference to the current state object it's wrapping as `session.state`.

When a Session instance is created from source data, Redux-ORM creates temporary subclasses of the Model types available in the Schema, "binds" them to that session, and exposes the subclasses as fields on the Session instance. This means **it's important that you always extract the Model classes you want to use from the Session instance and interact with those**, rather than using the versions you might have directly imported at the module level. If you're writing your reducers as part of your Model classes, Redux-ORM passes the bound version of the current class in as the second argument, and the current Session instance as the third argument.

Models

The Model instances returned by a Session really just **act as facades over the plain Javascript objects inside the store**. When a Model instance is requested, Redux-ORM generates property fields on the Model instances based on the keys in the underlying object, as well as the declared relations. These property fields define getters and setters that encapsulate the real behavior.

Depending on what that field is, the getters will return the plain value from the underlying object, a new Model instance for a single relation, or a QuerySet instance for a collection relation. The underlying object can be accessed directly using `someModelInstance.ref`.

The use of properties and getters also means that **relations are not denormalized until you actually access those properties**. So, even an entity has a lot of relations, there shouldn't be any additional expense to just retrieving a Model instance for that entity.

Updates

Up through version 0.8.x, Redux-ORM used an internal queue of actions that would only be applied when you ran `session.reduce()`. In version 0.9 and later, a `Session` instance will automatically apply updates immediately, and the updated "database state" is immediately available as `session.state`. Here's an example:

```
// Assume we've done: orm.register(Pilot);
const initialState = orm.getEmptyState();
console.log(state);
/*
{
  Pilot : {
    itemsById : {},
    items : []
  }
}
*/

const session = orm.session(initialState);
const {Pilot} = session;

const pilotModel = Pilot.create({id : 1, name : "Jaime Wolf", gunner
y : 3});
const secondState = session.state;
console.log(initialState === secondState);
// false

console.log(secondState);
/*
{
```

```
  Pilot : {
    itemsById : {
      1 : {id : 1, name : "Jaime Wolf", gunnery : 3}
    },
    items : [1]
  }
}
*/

pilotModel.gunnery = 2;

const thirdState = session.state;
console.log(secondState === thirdState);
// false

console.log(thirdState);
/*
{
  Pilot : {
    itemsById : {
      1 : {id : 1, name : "Jaime Wolf", gunnery : 2}
    },
    items : [1]
  }
}
*/
```

As discussed in the Redux docs section on Immutable Update Patterns, updating nested data can be complicated, which is one of the reasons why we advise that you normalize your data in the first place. But, even with normalization, there's still a few levels of nesting, which need to be updated each time there's a change.

Notice that in each of these cases, **Redux-ORM correctly applied immutable updates to the actual plain JS object for that item, the `byId` and `items` sections, the `Pilot` object, and the root state object itself, but the code we've written looks like pretty normal mutating code**. That means that the code we're writing is simpler, and we're still getting all the benefits of using Redux the way it was meant to be used.

It's worth noting that neither Redux nor Redux-ORM will stop you from directly reaching into the store and mutating objects if you have references to them. But, by using Redux-ORM models for most of the access, it's less likely you'd be poking around at the plain objects to begin with.

Managing Relations

Redux-ORM relies on a `QuerySet` class as an abstraction for managing collections and many-to-many relations. A `QuerySet` knows what `Model` type it relates to. `QuerySet`s are created when you access a `many()`-type field on a model instance, or do queries against a Model class type. Operations such as `filter()` return a new `QuerySet` instance with additional filtering logic in place.

A `QuerySet` is lazy, and does not apply its filters until you call `querySet.toRefArray()` or `querySet.toModelArray()`, which return arrays containing either the plain underlying JS objects from the store, or Model class instances, respectively.

For `many`-type relations, Redux-ORM will auto-generate "through model" classes, which store the IDs for both items in the relation. For example, a `Lance` with a field `pilots : many("Pilot")` would result in a `LancePilot` class and table. You can also pass a custom through model class as an argument to the `many()` relational attribute function instead, which can be useful for tracking additional data about the relation.

Synchronization

It's also important to understand that Redux-ORM ***does not*** include any capabilities for synchronizing data with a server (such as the methods included with Backbone.Model or Ember Data). It is *only* about managing relations stored locally as plain JS data. (In fact, despite the name, it doesn't even depend on Redux at all.) You are responsible for handling data syncing yourself.