Basic Usage

Redux-ORM comes with excellent documentation. The main Redux-ORM README, Redux-ORM Primer tutorial, and the API documentation cover the basics very well, but here's a quick recap.

Defining Model Classes

First, you need to determine your different data types, and how they relate to each other. Think of this in the same way you would set up a database schema, with tables, foreign keys, and so on. Then, declare ES6 classes that extend from Redux-ORM's `Model` class. Like other file types in a Redux app, there's no specific requirement for where these declarations should live, but you might want to put them into a `models.js` file, or a `/models` folder in your project

As part of those declarations, add a static `fields` section to the class itself that uses Redux-ORM's relational operators to define what relations this class has. This can be done with three different forms of JS syntax: attaching the declaration to the class variable after it's been declared; using a `static get` function on the class; or using the Class Properties syntax to declare a `static` field. You should also declare a `"modelName"` property so that table names are correctly generated even if the code is minified.

In Redux-ORM 0.9, the API was updated to allow you to declare what per-instance fields are expected for each model type in addition to the relational fields. It's not required, but it allows the library to more efficiently generate properties that look up the right values for those fields from the store.

Here's some basic examples of how to declare Redux-ORM Model classes. I'll demonstrate each of the three approaches for adding the different relations and fields to the classes, but all three of them should behave the same way.

```
import {Model, attr, fk, oneToOne, many} from "redux-orm";
```

```
// Basic JS class syntax, with field declarations added to the class later
export class Pilot extends Model{
  // Other class methods would go here
}

Pilot.modelName = "Pilot";

Pilot.fields = {
  // An instance field on the model
  id : attr(),
  name : attr(),
  // A foreign key relation
  mech : fk("Battlemech"),
  // A one-to-one relation
  lance : oneToOne("Lance")
};



// Using static getter functions
export class Battlemech extends Model{
  static get modelName() {
    return "Battlemech";
  }

  static get fields() {
    return {
      id : attr(),
      name : attr(),
      pilot : fk("Pilot"),
      lance : oneToOne("Lance"),
    }
  }
}


// Using the Stage 3 Class Properties syntax
export class Lance extends Model{
  static modelName = "Lance"

  static fields = {
    id : attr(),
    name : attr(),
    // A many-to-many relation
    mechs : many("Battlemech"),
    pilots : many("Pilot")


  }
}
```

Creating an ORM Instance

Once you've defined your models, you need to create an instance of the Redux-ORM `ORM` class, and pass the model classes to its `register` method. This `ORM` instance will be a singleton in your application:

```
import {ORM} from "redux-orm";
import {Pilot, Battlemech, Lance} from "./models";

const orm = new ORM();
orm.register(Pilot, Battlemech, Lance);
export default orm;
```

Setting Up the Store and Reducers

Next, you need to decide how to integrate Redux-ORM into your reducer structure. Per the docs, there's two ways to define reducer logic with Redux-ORM.

The primary approach is to write normal reducer functions that have access to the "database state" portion of your Redux store. In those reducers, you import the `ORM` instance, and use it to create a `Session` instance based on the initial DB state. After executing update commands via the `Session`, you would specifically return the updated DB state object generated by the `Session` instance. Here's what that might look like:

```
// entitiesReducer.js
import orm from "models/schema";

// This gives us a set of "tables" for our data, with the right structure
const initialState = orm.getEmptyState();

export default function entitiesReducer(state = initialState, action) {
    switch(action.type) {
        case "PILOT_CREATE": {
            const session = orm.session(state);
            const {Pilot} = session;
            // Creates a Pilot class instance, and updates session.state i
mmutably
            const pilot = Pilot.create(action.payload.pilotDetails);

            // Returns the updated "database tables" object
            return session.state;
        }
        // Other actual action cases would go here
        default : return state;
    }
}

// rootReducer.js
import {combineReducers} from "redux";
import entitiesReducer from "./entitiesReducer";

const rootReducer = combineReducers({
    entities: entitiesReducer
});

export default rootReducer;
```

The second approach is to define a `reducer()` function on your Model classes, then use the `createReducer()` utility method provided by Redux-ORM to create a reducer function you can add to your Redux store. The Model reducer methods will be called with the current action, a version of the current Model class that's tied to the `Session`, and the `Session` itself:

```
// Pilot.js
class Pilot extends Model {
    static reducer(action, Pilot, session) {
        case "PILOT_CREATE": {
            Pilot.create(action.payload.pilotDetails);
            // Don't need to return anything. The Session was updated auto
matically, and
            // its current state will be returned by the generated paren
t reducer
            break;
        }
    }
}

// rootReducer.js
import {combineReducers} from "redux";
import {createReducer} from "redux-orm";
import {createReducer} from "redux-orm";
import orm from "models/schema";

const rootReducer = combineReducers({
    entities : createReducer(orm)
});
export default rootReducer;
```

I personally prefer the first approach of “using Redux-ORM in my own reducers”, and actually have done that since I started using Redux-ORM, even though earlier versions of the docs suggested the “model-attached reducers” approach.

Selecting Data

Finally, the ORM instance can be used to look up data and relationships in selectors and `mapState` functions:

```
import React, {Component} from "react";
import orm from "./schema";
import {selectEntities} from "./selectors";

export function mapState(state, ownProps) {
    // Create a Redux-ORM Session instance based on the "tables" in our en
tities slice
    const entities = selectEntities(state);
    const session = orm.session(entities);
    const {Pilot} = session;

    const pilotModel = Pilot.withId(ownProps.pilotId);

    // Retrieve a reference to the real underlying object in the store
    const pilot = pilotModel.ref;

    // Dereference a relation and get the real object for it as well
    const battlemech = pilotModel.mech.ref;

    // Dereference another relation and read a field from that model
    const lanceName = pilotModel.lance.name;

    return {pilot, battlemech, lanceName};
}

export class PilotAndMechDetails extends Component { ....... }

export default connect(mapState)(PilotAndMechDetails);
export default connect(mapState)(PilotAndMechDetails);
```

That covers the basic concepts. Next, we'll look at Redux-ORM's APi and concepts in more detail.