Specialized Behavior

Redux-ORM provides some very useful tools for dealing with normalized data, but it only has so much functionality built-in. Fortunately, it also serves as a great starting point for building additional functionality.

Serializing and Deserializing Data

As mentioned earlier, the Normalizr library is the de-facto standard for normalizing data received from the server. I've found that Redux-ORM can be used to mostly build a replacement for Normalizr. I added static `parse()` methods to each of my classes, which know how to handle the incoming data based on the relations:

```
class Lance extends Model {
    static parse(lanceData) {
        // Because it's a static method, "this" refers to the class itsel
f.
        // In this case, we're running inside a subclass bound to the Sess
ion.
        const {Pilot, Battlemech, Officer} = this.session;

        // Assume our incoming data looks like:
        // {name, commander : {}, mechs : [], pilots : []}

        let clonedData = {
           ...lanceData,
           commander = Officer.parse(clonedData.commander),
           mechs : lanceData.mechs.map(mech => Battlemech.parse(mech)),
           pilots : lanceData.pilots.map(pilot => Pilot.parse(pilot))
        };

        return this.create(clonedData);
    }
}
```

This method could then be called in either a thunk or a reducer to help

process data from a response. If called in a reducer, the updates could be

applied directly to the existing state. If used in a thunk, you'd want to take the resulting normalized data and include it in a dispatched action for merging into the store.

There is one noticeable difference out of the box between this technique and using Normalizr. Normalizr will correctly de-duplicate and merge multiple entries with the same ID, whereas Redux-ORM's `Model.create()` does not. After noticing this, I filed an issue to discuss improvements to creating/merging duplicate entries. After discussion, the issue was eventually resolved by adding a `Model.upsert()` method, which first checks to see if a model with the same ID exists. If so, the existing model is updated, otherwise a new one is created.

On the flip side, it's often necessary to send a denormalized version of the data back to the server. I've added `toJSON()` methods to my models to support that:

```
class Lance extends Model {
    toJSON() {
        const data = {
            // Include all fields from the plain data object
            ...this.ref,
            // As well as serialized versions of all the relations we kno
w about
            commander : this.commander.toJSON(),
            pilots : this.pilots.withModels.map(pilot => pilot.toJSON()),
            mechs : this.mechs.withModels.map(mech => mech.toJSON())
        };

        return data;
    }
}
```

Creation and Deletion

I often need to generate initial data for a new entity in an action creator before either sending it to the server or adding it to the store. While I'm not sure it's the "best" approach, for now I've added `generate` methods that can help encapsulate the process:

```
const defaultAttributes = {
```

```
    name : "Unnamed Lance",
};

class Lance extends Model {
    static generate(specifiedAttributes = {}) {
        const id = generateUUID("lance");

        const mergedAttributes = {
            ...defaultAttributes,
            id,
            ...specifiedAttributes,
        }

        return this.create(mergedAttributes);
    }
}

function createNewLance(name) {
    return (dispatch, getState) => {
        const session = getUnsharedEntitiesSession(getState());
        const {Lance} = session;

        const newLance = Lance.generate({name : "Command Lance"});
        const itemAttributes = newLance.toJSON();

        dispatch(createEntity("Lance", newLance.getId(), itemAttributes));
    }
}
```

Meanwhile, Redux-ORM's `Model.delete` method doesn't fully cascade, so I've implemented a custom `deleteCascade` method where neded:

```
class Lance extends Model {
    deleteCascade() {
        this.mechs.toModelArray().forEach(mechModel => mechModel.deleteCas
cade());
        this.pilots.toModelArray().forEach(pilotModel => pilotModel.delete
Cascade());
        this.delete();
    }
}
```

I've also implemented a number of additions to better handle copying data back and forth between various versions of a model instance (such as a

"current" version vs a "WIP draft" version), which we'll go over in a later part of the course.