Loading Pilot Data

With the Pilot model hooked up to the ORM instance, we can start using it for something useful. We already have a sample data file and a `DATA_LOADED` action being dispatched containing that data. Let's add some pilot entries to that list, and load them into memory.

For sample entries, we're going to use the characters from the legendary Black Widow Company of the Wolf's Dragoons mercenary unit, as they existed in the 3025 game era. This group was led by the notorious Black Widow herself, Natasha Kerensky.

We'll define several attributes for each pilot: an ID, their name, rank, age, in-game Gunnery and Piloting skills (lower is better), and a short description of what type of Battlemech they pilot. An example looks like:

```
{
    pilots: [
        {
            id : 1,
            name : "Natasha Kerensky",
            rank : "Captain",
            gunnery : 2,
            piloting : 2,
            age : 52,
            mechType : "WHM-6R",
        },
        {
            id : 2,
            name : "Colin Maclaren",
            rank : "Sergeant",
            gunnery : 3,
            piloting : 4,
            age : 43,
            mechType : "MAD-3R",
        },
```

```
    ]
}
```

Once those entries have been added to the sample data, they wil be included in the `DATA_LOADED` action we've been dispatching. So, we should update our `entitiesReducer` to respond to that action.

Redux-ORM provides methods on Model instances to query, update, and delete instances. It also provides a `create` static method on the class itself to create new instances. I like to write `parse` methods on my Model classes that encapsulate the logic for handling all the relations that might be involved in nested JSON data. Since this is our first Model type, there's actually nothing special we need to do in terms of loading - we just pass the data straight to `Model.create()`.

Commit 1825918: Add Pilot parsing, and load Pilots from sample data

features/pilots/Pilot.js

```
import {Model} from "redux-orm";

export default class Pilot extends Model {
    static modelName = "Pilot";

    static parse(pilotData) {
        // We could do useful stuff in here with relations,
        // but since we have no relations yet, all we need
        // to do is pass the pilot data on to create()

        // Note that in a static class method, `this` is the
        // class itself, not an instance
        return this.create(pilotData);
    }
}
```

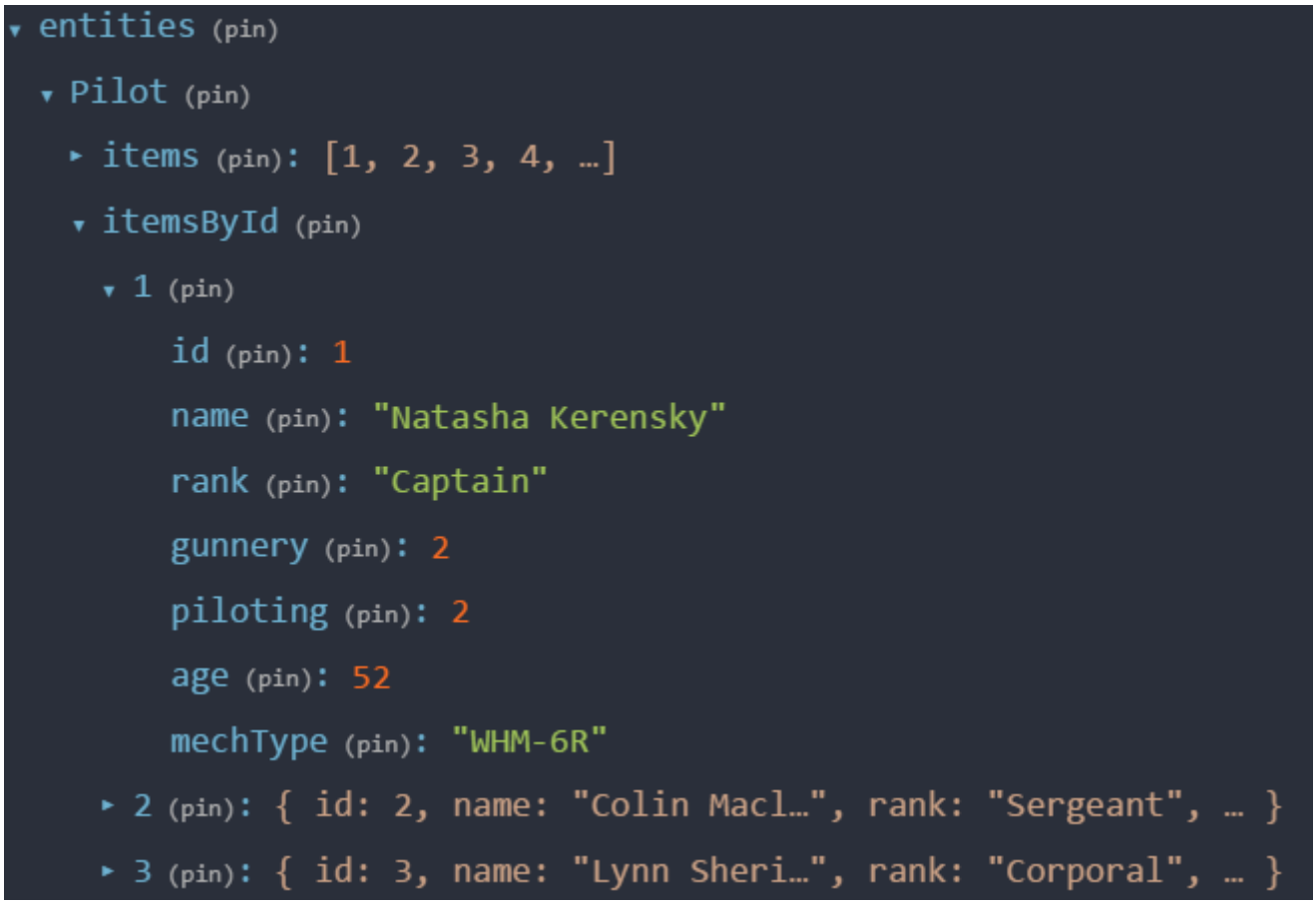
app/reducers/entitiesReducer.js

```
export function loadData(state, payload) {
    // Create a Redux-ORM session from our entities "database tables" obje
ct
    const session = orm.session(state);
    // Get a reference to the correct version of the Pilots class for thi
s Session
    const {Pilot} = session;
```

```
    const {pilots} = payload;

    // Insert the Pilot entries into the Session
    pilots.forEach(pilot => Pilot.parse(pilot));

    // return a new version of the entities state object with the inserte
d entries
    return session.state;
}

export default createReducer(initialState, {
    [DATA_LOADED] : loadData
});
```

Notice that we actually have two different slice reducers responding to the same action! Both our `unitInfoReducer` and our `entitiesReducer` are responding to the `DATA_LOADED` action. **This is a key concept for Redux usage, which is often misunderstood or ignored**. It doesn't happen by accident, or completely automatically - the `combineReducers` function we're using in our `rootReducer` is specifically calling both slice reducers, and giving them a chance to respond to that action by updating their own slice of data. We could implement the same behavior ourselves, but `combineReducers` does it for us. **(We could also choose *not* to use `combineReducers` if we wanted to, and maybe handle things a different way if it made sense.)**

If we hit the "Reload Unit Data" button in our "Tools" tab and check out the results in the DevTools, our `entities` slice should now contain entries for each of the pilot entries in our sample data:



Also notice that Redux-ORM automatically stored all the pilots in "normalized" form, by creating an `items` array for the Pilot type that stores a list of all item IDs, and an `itemsById` lookup table that stores the actual objects keyed by their IDs.