6.170 Recitation 5; More on Express, Mongoose

Prerequisites

- Installation instructions completed for Node.js, Express, and MongoDB
- Familiarity with the Express framework (HTTP methods, routes, views, etc.)

Goals for this recitation

1. Understand the motivation for ORMs, the benefits and drawbacks
2. Get lots of hands on experience with Mongoose

Note: This recitation is very hands-on; the code used throughout can be found at https://github.com/kongming92/6170-mongoose-demo. The master branch contains the completed code, while you can trace through each of the branches to see the progression of the code.

1. Why ORMs?

Consider the code that you wrote using Monk for P2.1. MongoDB gives us a fairly clean way to make calls into the database. However, just using MongoDB queries gives us very little control over what *could* be stored in the database. For example, there is nothing that constrains every object in a collection to have the same fields, or even the same number of type of fields.

In contrast, if we were using a relational database, like any flavor of SQL, it wouldn't be so bad. In SQL, we have a *schema*, a way of enforcing the type of data that goes into each column of the table. (MongoDB doesn't have rows/columns and tables, but collections and fields are similar). Last week, it was recommended that you carefully keep track of the fields stored in the database, perhaps by documenting the contents of each collections. As you can imagine, that isn't a perfect solution. Today, we will introduce schemas to the way we interact with MongoDB.

Another issue with just writing MongoDB queries is that it doesn't give us a way to write functions to manipulate the data or return it in a format that we want it. We are constrained to getting documents from a collection and manipulating them later in our routing functions. This leads to lots of logic in our routes/controllers, something that makes it harder to reuse code and achieve modularity. Ideally, all of this logic would go into a model, and the router/controller would simply be the "glue" that takes data from the model and sends it to the view. Mongoose gives us the ability to write our own model functions, and provides built in mechanisms for common tasks like validation.

Finally, as discussed in lecture, we often want to reference one object in another. Mongoose gives us an easy way to do this while keeping storage efficient.

2. Getting Started with Mongoose

1) Install Mongoose in your project:

```
$ npm install mongoose
```

2) Ensure that Mongoose is in your dependencies by looking at the “dependencies” object in package.json.

3) In the main entry point to your Express application (typically your app.js), get the Mongoose module and open a connection to the database:

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');
```

4) Listen for when the connection is successfully established; register a callback for when that happens:

   Note: Remember that we are in the single-threaded world of Javascript. “Expensive” operations, such as connecting and performing operations on a database, should be done asynchronously, so that the main thread is not blocked and is able to accept additional incoming connections.

```
var db = mongoose.connections;
db.on('error', function() { console.log('connection error'); });
db.once('open', function() {
      // declare schemas and models here
});
```

5) Declare a schema:

```
var movieSchema = mongoose.Schema({
      _id: Number,
      title: String,
      starring: Array
});
```

6) Compile the schema into a Model:

```
var Movie = mongoose.model('Movie', movieSchema);
```

7) Perform CRUD operations on the schema; e.g.

```
var m = new Movie({
      title: "Guardians of the Galaxy",
      starring: ["Chris Pratt", "Bradley Cooper"]
});
```

3. Defining Schemas and Models

Go to the code, and checkout the first branch. This recitation is organized into 4 main parts, each of which has a few exercises. Checking out the next branch will reveal the solutions to the previous part and also the new exercise items.

For part 1, checkout the `part1-movieschema branch`.

```
$ git checkout part1-movieschema
```

Exercise 1: In movie-data-mongoose.js, declare a schema and make a model for a Movie

Exercise 2: in movies-mongoose.js, get the list of Movies, sorted by ascending order of time. Display the list of movies. A view is provided in views/movies/index.ejs.

When you are done, to see the answers and go to the next part, do:

```
$ git checkout part2-theaterschema
```

Notes:

In Mongoose, *schemas* define the structure of the data. Schemas are transformed into *models*, which represent classes that actually store the data. Remember that Mongoose sits between the actual database and the rest of the web application; all database reads and writes go through Mongoose. Thus, Mongoose will enforce that all writes through the model objects conform to the schema, and all database reads will give you an object as defined in the schema.

In order to define our objects in one file and use them in another, we must *export* the objects. When we do something like

```
var data = require('../data/movie-data-mongoose');
```

`data` becomes the `exports` object from the file that is passed to the `require` function. So in our

case, we want to export both the Movie and Theater Models.

Inside our router, notice the call to `data.Movie.find`. We pass it an empty object to get all Movies. Remember that database calls in Javascript are asynchronous. Thus, we can chair further operations, such as `sort`. Our query is run when the call to `exec` is made, and as expected, `exec` takes a callback function. The order of the arguments in the callbacks is always error first, followed by the response from the database. The response from the database is simply a Javascript object, which we can send to the view.

4. Defining the Theater Schema; References and Populating (10 minutes)

Exercise 3: Declare a schema and make a model for a Theater

Exercise 4: Extend the Movie schema to include a reference to a Theater.

Exercise 5: Extend the find query to populate each Movie with its Theater information.

Exercise 6: Extend the object returned by the formatMovie function to include the Theater's name.

When you are done, to see the answers and get ready for the next part, do

```
$ git checkout part3-instancemethod
```

Notes:

We want to associate each Movie with a Theater. Recall from lecture there are two ways of referring to one object inside another. The first is *embedding*, where one object is copy-pasted into the other every time it's used. In our case, we would copy the Theater object into each Movie object. This gives us easy access to all data, at the cost of storing lots of duplicates. (See data/movie-data-embed.js).

The other way is to simulate *joins*, in which only a single reference (often an id) is stored in place of the data. (See data/movie-data-join.js). We would simply store the Theater's id in the Movie object. While space-saving, one must remember to do two queries, once to get the Movie, and once more to get the Theater using the id in the Movie object.

What Mongoose lets us do it combine the two. In the Movie schema, we declare a theater, of type Number, referring to the Theater model. Then, in our query to get the Movies, we simply call the `populate` function, which does the queries necessary to fill in each Movie with the relevant Theater object. This lets us store the objects in a compact way but still have direct access to all information without having to do a second query explicitly.

5. Defining Custom Instance Methods (5 minutes)

One benefit to using the Mongoose framework is the ability to write custom methods on our model classes. This allows us to keep the vast majority of the functionality in the model, and simply make calls to the model instead of writing lots of custom logic in our routes.

Exercise 7: Define an instance method on `theaterSchema` called `getDescription`, which should return the concatenation of the name and location, separated by a dash

Exercise 8: Use the getDescription method created in Exercise 7 in place of movie.theater.name

When you are done, to see the answers and get ready for the next part, do

```
$ git checkout part4-validators
```

6. Defining Validators (10 minutes)

Validators are important whenever you're writing to the database, to ensure that all of you data makes sense, is correctly formatted, etc. Mongoose will check that your data corresponds to the schema, but you must generally do additional checks, like making sure fields aren't empty, etc.

Exercise 9 (mistakenly labeled Exercise 8 in the code): Write validators for both Theater and Movie, ensuring Strings are not empty and Numbers are positive.

Notes:

Validators are always functions that return a boolean value. If a validator returns false, saving the object will cause an error to be issued, which is passed to the callback as the `error` argument. This gives you a chance to inform the user of the error or to handle it in some special way.

To see the answers and the code up to this point, cleaned up, do

```
$ git checkout final
```

7. Inserting Objects, Forms, etc. (time permitting)

Extend the application so that the user can insert their own movies, via a form perhaps. Some things to think about:

- How is the form wired up to the backend? What route will it make a request to, and what should happen after the form is processed?
- A movie needs to have a theater associated with it. How should we specify that on the form? What is actually inserted into the database?
- How do we handle errors?

To see one possible solution, do

```
$ git checkout final+insert
```

Notes:

Note the /new route, which corresponds to the URI /movies/new. This renders a form. When rendering this form, we get a list of theaters from the database, including both their id and name. In the form itself (the view), when we make the dropdown, we set the value to be sent to the server as the theater's id.

Also note the way that we handle errors (line 91). In our simple example, we redirect back to the main page, and if necessary, send a key-value pair over the URL to specify an error message to display to the user.