SAMPLE CHAPTER

MANNING

Getting MEAN with Mongo, Express Angular, and Node

Simon Holmes

Getting MEAN with Mongo, Express, Angular, and Node

by Simon Holmes

Chapter 5

brief contents

5 Building a data model with MongoDB and Mongoose

This chapter covers

- How Mongoose helps bridge an Express/Node application to a MongoDB database
- Defining schemas for a data model using Mongoose
- Connecting an application to a database
- Managing databases using the MongoDB shell
- Pushing a database into a live environment
- Using the correct database depending on the environment, distinguishing between local and live versions of an application

In chapter 4 we ended up by moving our data out of the views and backward down the MVC path into the controllers. Ultimately, the controllers will pass data to the views, but they shouldn't store it. Figure 5.1 recaps the data flow in an MVC pattern.

For storing the data we'll need a database, specifically MongoDB. So this is our next step in the process: creating a database and data model.

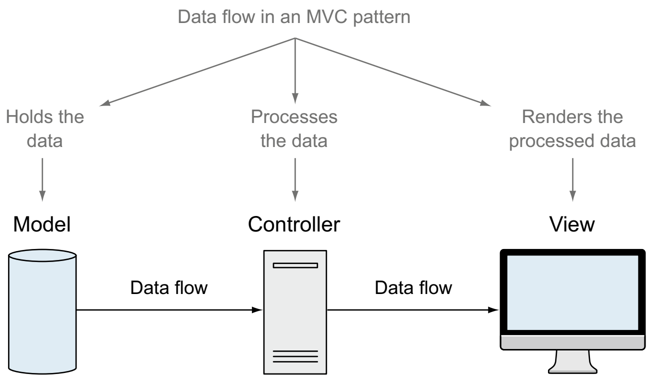


Figure 5.1 In an MVC pattern, data is held in the model, processed by a controller, and then rendered by a view.

NOTE If you haven't yet built the application from chapter 4, you can get the code from GitHub on the chapter-04 branch at github.com/simonholmes/getting-MEAN. In a fresh folder in terminal the following command will clone it:

```
$ git clone -b chapter-04 https://github.com/simonholmes/getting-MEAN.git
```

We'll start by connecting our application to a database before using Mongoose to define schemas and models. When we're happy with the structure we can add some test data directly to the MongoDB database. The final step will be making sure that this also works when pushed up to Heroku. Figure 5.2 shows the flow of these four steps.

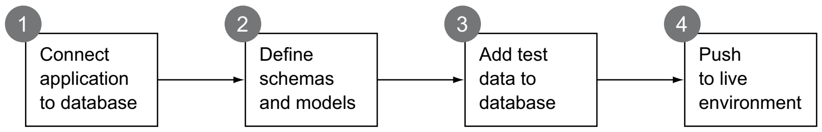

1 Connect application to database
2 Define schemas and models
3 Add test data to database
4 Push to live environment

Figure 5.2 Four main steps in this chapter, from connecting our application to a database to pushing the whole thing into a live environment

For those of you who are worried that you've missed a section or two, don't worry—we haven't created a database yet. And we don't need to. In various other technology stacks this can present an issue and throw errors. But with MongoDB we don't need to create a database before connecting to it. MongoDB will create a database when we first try to use it.

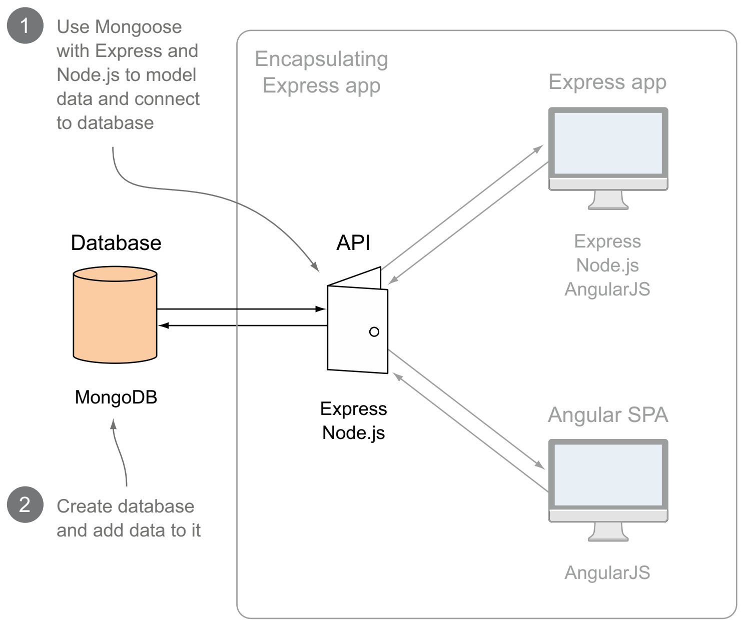

1 Use Mongoose with Express and Node.js to model data and connect to database
Encapsulating Express app
Express app
Database
API
Express
Node.js
AngularJS
MongoDB
Express
Node.js
Angular SPA
2 Create database and add data to it
AngularJS

Figure 5.3 The MongoDB database and using Mongoose inside Express to model the data and manage the connection to the database

Figure 5.3 shows where this chapter will focus in terms of the overall architecture.

We'll, of course, be working with a MongoDB database, but most of the work will be in Express and Node. In chapter 2 we discussed the benefits of decoupling the data integration by creating an API rather than tightly integrating it into the main Express app. So although we'll be working in Express and Node, and still within the same encapsulating application, we'll actually be starting the foundations of our API layer.

NOTE To follow through this chapter you'll need to have MongoDB installed. If you haven't done so already, you can find the instructions for this in appendix A.

The source code of the application as it will be at the end of this chapter is available from GitHub on the chapter-05 branch. In a fresh folder in terminal the following commands will clone it and install the npm module dependencies:

```
$ git clone -b chapter-05 https://github.com/simonholmes/getting-MEAN.git
$ cd getting-MEAN
$ npm install
```

5.1 Connecting the Express application to MongoDB using Mongoose

We could connect our application directly to MongoDB and have the two interact with each other using the native driver. While the native MongoDB driver is very powerful it isn't particularly easy to work with. It also doesn't offer a built-in way of defining and maintaining data structures. Mongoose exposes most of the functionality of the native driver, but in a more convenient way, designed to fit into the flows of application development.

Where Mongoose really excels is in the way it enables us to define data structures and models, maintain them, and use them to interact with our database. All from the comfort of our application code. As part of this approach Mongoose includes the ability to add validation to our data definitions, meaning that we don't have to write validation code into every place in our application where we send data back to the database.

So Mongoose fits into the stack inside the Express application by being the liaison between the application and the database, as shown in figure 5.4.

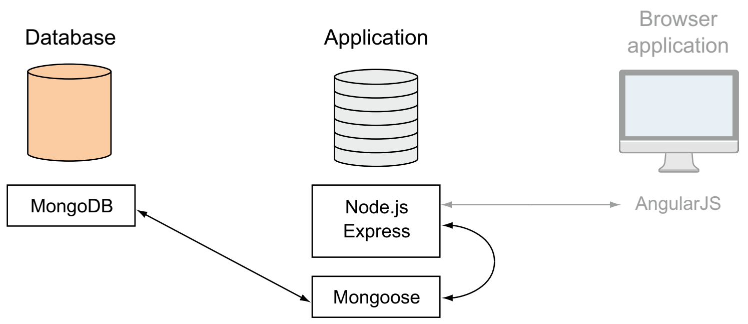


Figure 5.4 The data interactions in the MEAN stack and where Mongoose fits in. The Node/Express application interacts with MongoDB through Mongoose, and Node and Express can then also talk to Angular.

MongoDB only talks to Mongoose, and Mongoose in turn talks to Node and Express. Angular will not talk directly to MongoDB or Mongoose, but only to the Express application.

You should already have MongoDB installed on your system (covered in appendix A), but not Mongoose. Mongoose isn't installed globally, but is instead added directly to our application. We'll do that now.

5.1.1 Adding Mongoose to our application

Mongoose is available as an npm module. As you saw in chapter 3, the quickest and easiest way to install an npm module is through the command line. We can install Mongoose and add it to our list of dependencies in package.json with one command.

So head over to terminal and make sure the prompt is at the root folder of the application, where the package.json file is, and run the following command:

```
$ npm install --save mongoose
```

The `--save` flag is what tells npm to add Mongoose to the dependency list in package.json. When that command has finished running you'll be able to see a new *mongoose* folder inside the *node_modules* folder of the application, and the dependencies section of the package.json file should look like the following code snippet:

```
"dependencies": {
  "express": "~4.9.0",
  "body-parser": "~1.8.1",
  "cookie-parser": "~1.3.3",
  "morgan": "~1.3.0",
  "serve-favicon": "~2.1.3",
  "debug": "~2.0.0",
  "jade": "~1.6.0",
  "mongoose": "~3.8.20"
}
```

You may have slightly different version numbers, of course, but at the time of writing the latest stable version of Mongoose is 3.8.20. Now that Mongoose is installed, let's get it connected.

5.1.2 *Adding a Mongoose connection to our application*

At this stage we're going to connect our application to a database. We haven't created a database yet, but that doesn't matter because MongoDB will create a database when we first try to use it. This can seem a little odd, but for putting an application together it's a great advantage: we don't need to leave our application code to mess around in a different environment.

MONGODB AND MONGOOSE CONNECTION

Mongoose opens a pool of five reusable connections when it connects to a MongoDB database. This pool of connections is shared between all requests. Five is just the default number and can be increased or decreased in the connection options if you need to.

BEST-PRACTICE TIP Opening and closing connections to databases can take a little bit of time, especially if your database is on a separate server or service. So it's best to only run these operations when you need to. The best practice is to open the connection when your application starts up, and to leave it open until your application restarts or shuts down. This is the approach we're going to take.

SETTING UP THE CONNECTION FILE

When we first sorted out the file structure for the application we created three folders inside the app_server folder: *models*, *views*, and *controllers*. For working with data and models, we'll be predominantly based in the app_server/models folder.

Setting up the connection file is a two-part process: creating the file and requiring it into the application so that it can be used.

Step one: create a file called db.js in app_server/models and save it. For now we'll just `require` Mongoose in this file, with the following single command line:

```
var mongoose = require( 'mongoose' );
```

Step two: bring this file into the application by requiring it in app.js. As the actual process of creating a connection between the application and the database can take a little while, we want to do this early on in the setup. So amend the top part of app.js to look like the following code snippet (modifications in bold):

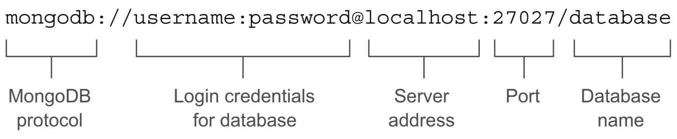
```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
require('./app_server/models/db');
```

We're not going to export any functions from db.js, so we don't need to assign it to a variable when we `require` it. We need it to be there in the application, but we're not going to need to hook into any methods of it from within app.js.

If you restart the application it should run just as before, but now you have Mongoose in the application. If you get an error, check that the path in the `require` statement matches the path to the new file, that your package.json includes the Mongoose dependency, and that you've run `npm install` from terminal in the root folder of the application.

Creating the Mongoose connection

Creating a Mongoose connection can be as simple as declaring the URI for your database and passing it to Mongoose's `connect` method. A database URI is a string following this construct:



```
mongodb://username:password@localhost:27027/database
```

The username, password, and port are all optional. So on your local machine your database URI is going to be quite simple. For now, assuming that you have MongoDB installed on your local machine, adding the following code snippet to db.js will be all you need to create a connection:

```
var dbURI = 'mongodb://localhost/Loc8r';
mongoose.connect(dbURI);
```

If you run the application with this addition to db.js it should still start and function just as before. So how do you know your connection is working correctly? The answer lies in connection events.

Monitoring the connection with Mongoose connection events

Mongoose will publish events based on the status of the connection, and these are really easy to hook into so that you can see what's going on. We're going to use events to see when the connection is made, when there's an error, and when the connection is disconnected. When any one of these events occurs we'll log a message to the console. The following code snippet shows the code required to do this:

```
mongoose.connection.on('connected', function () {
  console.log('Mongoose connected to ' + dbURI);
});
mongoose.connection.on('error',function (err) {
  console.log('Mongoose connection error: ' + err);
});
mongoose.connection.on('disconnected', function () {
  console.log('Mongoose disconnected');
});
```

Monitoring for successful connection through Mongoose

Checking for connection error

Checking for disconnection event

With this added to db.js, when you restart the application you should see the following confirmations logged to the terminal window:

```
Express server listening on port 3000
Mongoose connected to mongodb://localhost/Loc8r
```

If you restart the application again, however, you'll notice that you don't get any disconnection messages. This is because the Mongoose connection doesn't automatically close when the application stops or restarts. We need to listen for changes in the Node process to deal with this.

Closing a Mongoose connection

Closing the Mongoose connection when the application stops is as much a part of the best practice as opening the connection when it starts. The connection has two ends: one in your application and one in MongoDB. MongoDB needs to know when you want to close the connection so that it doesn't keep redundant connections open.

To monitor when the application stops we need to listen to the Node.js process, listening for an event called SIGINT.

Listening for SIGINT on Windows

SIGINT is an operating system–level signal that fires on Unix-based systems like Linux and Mac OS X. It also fires on some later versions of Windows. If you're running on Windows and the disconnection events don't fire, you can emulate them. If you need to emulate this behavior on Windows you first add a new npm package to your application, `readline`. So in your package.json file update the dependencies section like this:

```
"dependencies": {
  "express": "3.4.x",
  "jade": "*",
  "mongoose": "3.8.x",
  "readline": "0.0.x"
}
```

When that's done, install it into the application by running `npm install` from the command line, based in the same folder as your package.json file.

In the db.js file, above the event listener code, add the following:

```
var readLine = require ("readline");
if (process.platform === "win32"){
    var rl = readLine.createInterface ({
        input: process.stdin,
        output: process.stdout
    });
    rl.on ("SIGINT", function (){
        process.emit ("SIGINT");
    });
}
```

This will emit the SIGINT signal on Windows machines, allowing you to capture it and gracefully close down anything else you need to before the process ends.

If you're using nodemon to automatically restart the application then you'll also have to listen to a second event on the Node process called SIGUSR2. Heroku uses another different event, SIGTERM, so we'll need to listen for that as well.

Capturing the process termination events

With all of these events, once we've captured them we prevent the default behavior from happening, so we need to make sure that we manually restart the behavior required. After closing the Mongoose connection, of course.

To do this, we need three event listeners and one function to close the database connection. Closing the database is an asynchronous activity, so we're going to need to pass through whatever function is required to restart or end the Node process as a callback. While we're at it, we can output a message to the console confirming that the connection is closed, and the reason why. We can wrap this all in a function called `gracefulShutdown` in db.js, as in the following code snippet:

```
var gracefulShutdown = function (msg, callback) {
  mongoose.connection.close(function () {
    console.log('Mongoose disconnected through ' + msg);
    callback();
  });
};
```

Define function to accept message and callback function

Close Mongoose connection, passing through an anonymous function to run when closed

Output message and call callback when Mongoose connection is closed

Now we need to call this function when the application terminates, or when nodemon restarts it. The following code snippet shows the two event listeners we need to add to db.js for this to happen:

```
process.once('SIGUSR2', function () {
  gracefulShutdown('nodemon restart', function () {
    process.kill(process.pid, 'SIGUSR2');
  });
});

process.on('SIGINT', function () {
  gracefulShutdown('app termination', function () {
    process.exit(0);
  });
});

process.on('SIGTERM', function() {
  gracefulShutdown('Heroku app shutdown', function () {
    process.exit(0);
  });
});
```

Listen for SIGUSR2, which is what nodemon uses

Send message to gracefulShutdown and callback to kill process, emitting SIGUSR2 again

Listen for SIGINT emitted on application termination

Send message to gracefulShutdown and callback to exit Node process

Listen for SIGTERM emitted when Heroku shuts down process

Now when the application terminates, it gracefully closes the Mongoose connection before it actually ends. Similarly, when nodemon restarts the application due to changes in the source files, the application closes the current Mongoose connection first. The nodemon listener is using `process.once` as opposed to `process.on`, as we only want to listen for the SIGUSR2 event once. nodemon also listens for the same event and we don't want to capture it each time, preventing nodemon from working.

TIP It's important to manage opening and closing your database connections properly in every application you create. If you use an environment with different process termination signals you should ensure that you listen to them all.

COMPLETE CONNECTION FILE

That's quite a lot of stuff we've added to the db.js file, so let's take a moment to recap. So far we have

- Defined a database connection string
- Opened a Mongoose connection at application startup
- Monitored the Mongoose connection events
- Monitored some Node process events so that we can close the Mongoose connection when the application ends

All together the db.js file should look like the following listing. Note that this doesn't include the extra code required by Windows to emit the SIGINT event.

Listing 5.1 Complete database connection file db.js in app_server/models

```
var mongoose = require( 'mongoose' );
var gracefulShutdown;
var dbURI = 'mongodb://localhost/Loc8r';
mongoose.connect(dbURI);
```

Define database connection string and use it to open Mongoose connection

```
mongoose.connection.on('connected', function () {
  console.log('Mongoose connected to ' + dbURI);
});
mongoose.connection.on('error',function (err) {
  console.log('Mongoose connection error: ' + err);
});
mongoose.connection.on('disconnected', function () {
  console.log('Mongoose disconnected');
});
```

Listen for Mongoose connection events and output statuses to console

```
gracefulShutdown = function (msg, callback) {
  mongoose.connection.close(function () {
    console.log('Mongoose disconnected through ' + msg);
    callback();
  });
};
```

Reusable function to close Mongoose connection

```
// For nodemon restarts
process.once('SIGUSR2', function () {
  gracefulShutdown('nodemon restart', function () {
    process.kill(process.pid, 'SIGUSR2');
  });
});
// For app termination
process.on('SIGINT', function() {
  gracefulShutdown('app termination', function () {
    process.exit(0);
  });
});
// For Heroku app termination
process.on('SIGTERM', function() {
  gracefulShutdown('Heroku app shutdown', function () {
    process.exit(0);
  });
});
```

Listen to Node processes for termination or restart signals, and call gracefulShutdown function when appropriate, passing a continuation callback

Once you have a file like this you can easily copy it from application to application, because the events you're listening for are always the same. All you'll have to do each time is change the database connection string. Remember that we also *required* this file into app.js, right near the top, so that the connection opens up early on in the application's life.

Using multiple databases

What you've seen so far is known as the default connection, and is well suited to keeping a single connection open throughout the uptime of an application. But if you want to connect to a second database, say for logging or managing user sessions, then you can use a named connection. In place of the `mongoose.connect` method you'd use a different method called `mongoose.createConnection`, and assign this to a variable. You can see this in the following code snippet:

```
var dbURIlog = 'mongodb://localhost/Loc8rLog';
var logDB = mongoose.createConnection(dbURIlog);
```

This creates a new Mongoose connection object called `logDB`. You can interact with this in the same ways as you would with `mongoose.connection` for the default connection. Here are a couple of examples:

```
logDB.on('connected', function () {
  console.log('Mongoose connected to ' + dbURIlog);
});
logDB.close(function () {
  console.log('Mongoose log disconnected');
});
```

Monitoring connection event for named connection

Closing named connection

5.2 Why model the data?

In chapter 1 we talked about how MongoDB is a document store, rather than a traditional table-based database using rows and columns. This allows MongoDB great freedom and flexibility, but sometimes we want—that is, we *need*—structure to our data.

Take the Loc8r homepage, for example. The listing section shown in figure 5.5 contains a specific data set that's common to all locations.

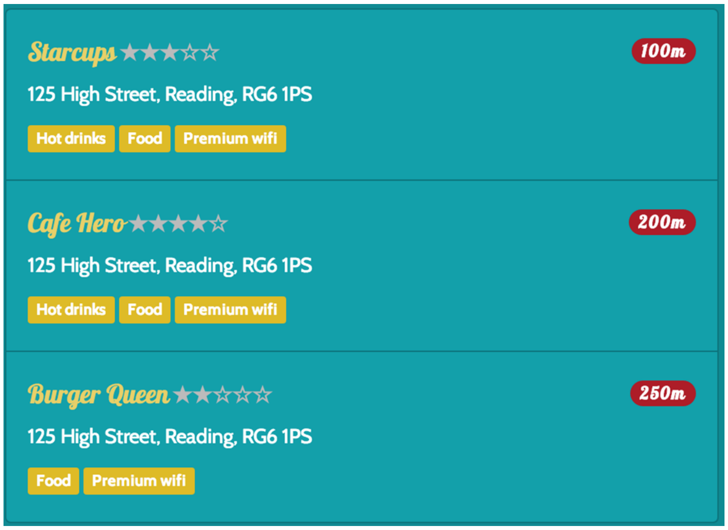


Figure 5.5 Listing section of the homepage has very defined data requirements and structure

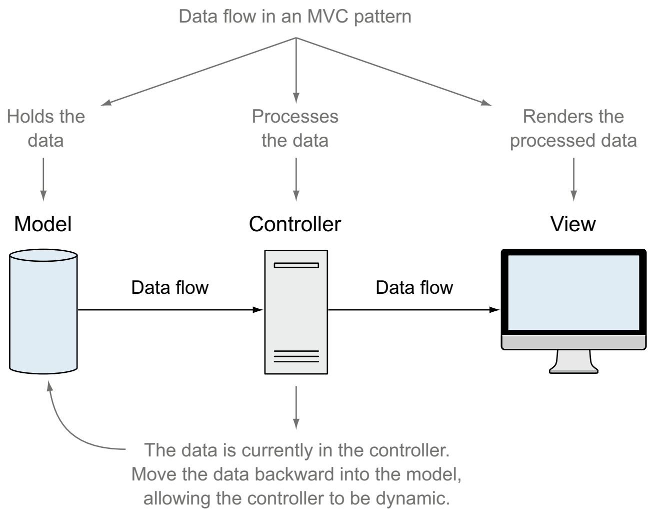

Data flow in an MVC pattern
Holds the data
Processes the data
Renders the processed data
Model
Controller
View
Data flow
Data flow
The data is currently in the controller. Move the data backward into the model, allowing the controller to be dynamic.

Figure 5.6 How data should flow in an MVC pattern, from the model, through the controller, into the view. At this point in our prototype our data is in the controller, so we want to move it a step back into the model.

The page needs these data items for all locations, and the data record for each location must have a consistent naming structure. Without this, the application wouldn't be able to find the data and use it. At this point in the development the data is held in the controller and being passed into the view. In terms of MVC architecture, we started off with the data in the *view* and then moved it back a step to the *controller*. Now what we need to do is move it back one final step to where it should belong, in the *model*. Figure 5.6 illustrates our current position, highlighting the end goal.

One of the outcomes of moving the data back through the MVC flow step-by-step as we've done so far is that it helps solidify the requirements of the data structure. This ensures the data structure accurately reflects the needs of our application. If you try to define your model first you end up second-guessing what the application will look like and how it will work.

So when we talk about modeling data, what we're really doing is describing how we want the data to be structured. In our application we could create and manage the definitions manually and do the heavy lifting ourselves, or we could use Mongoose and let it do the hard work for us.

5.2.1 What is Mongoose and how does it work?

Mongoose was built specifically as a MongoDB Object-Document Modeler (ODM) for Node applications. One of the key principles is that you can manage your data model from within your application. You don't have to mess around directly with databases or external frameworks or relational mappers; you can just define your data model in the comfort of your application.

First off, let's get some naming conventions out of the way:

- In MongoDB each entry in a database is called a *document.*
- In MongoDB a collection of documents is called a *collection* (think "table" if you're used to relational databases).
- In Mongoose the definition of a document is called a *schema.*
- Each individual data entity defined in a schema is called a *path.*

Using the example of a stack of business cards, figure 5.7 illustrates these naming conventions, and how each is related to the other.

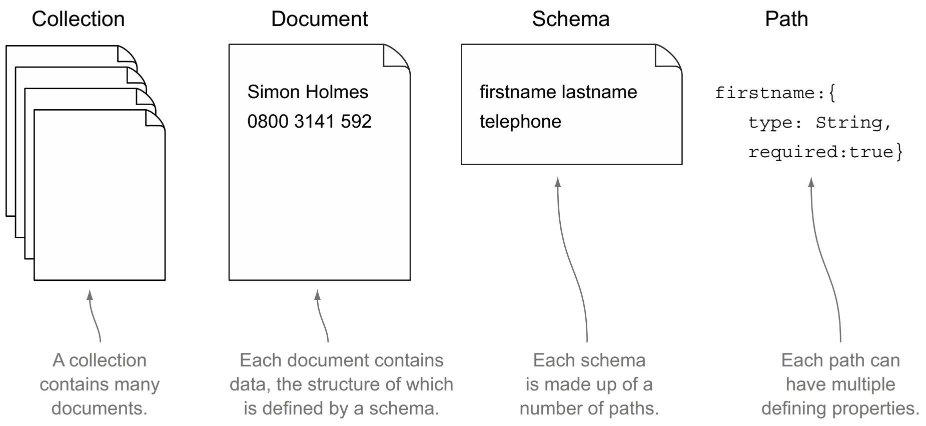

Collection
Document
Schema
Path
Simon Holmes
0800 3141 592
firstname lastname
telephone
firstname:{
type: String,
required:true}
A collection contains many documents.
Each document contains data, the structure of which is defined by a schema.
Each schema is made up of a number of paths.
Each path can have multiple defining properties.

Figure 5.7 Relationships among collections, documents, schemas, and paths in MongoDB and Mongoose, using a business card metaphor

One final definition is for models. A *model* is the compiled version of a schema. All data interactions using Mongoose go through the model. We'll work with models more in chapter 6, but for now we're focusing on building them.

How does Mongoose model data?

If we're defining our data in the application, how are we going to do it? In JavaScript, of course! JavaScript objects to be precise. We've already had a sneak peak in figure 5.7, but let's take a look at a simple MongoDB document and see what the Mongoose schema for it might look like. The following code snippet shows a MongoDB document, followed by the Mongoose schema:

```
{
  "firstname" : "Simon",
  "surname" : "Holmes",
  _id : ObjectId("52279effc62ca8b0c1000007")
}
```

Example MongoDB document

```
{
  firstname : String,
  surname : String
}
```

Corresponding Mongoose schema

As you can see, the schema bears a very strong resemblance to the data itself. The schema defines the name for each data path, and the data type it will contain. In this example we've simply declared the paths `firstname` and `surname` as strings.

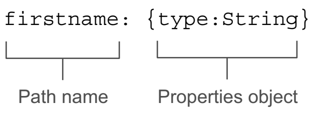
About the _id path

You may have noticed that we haven't declared the `id` path in the schema. `id` is the unique identifier—the primary key if you like—for each document. MongoDB automatically creates this path when each document is created and assigns it a unique `ObjectId` value. The value is designed to always be unique by combining the time since the Unix epoch with machine and process identifiers and a counter.

It's possible to use your own unique key system if you prefer, if you have a preexisting database, for example. In this book and the Loc8r application we're going to stick with the default `ObjectId`.

Breaking down a schema path

The basic construct for an individual path definition is the path name followed by a properties object. What we've just looked at is actually shorthand for when you just want to define the data type for that particular path. So a schema path is constructed of two parts, the path name and the properties object, like this:



```
firstname: {type:String}
```

Allowed schema types

The schema type is the property that defines the data type for a given path. It's required for all paths. If the only property of a path is the type, then the shorthand definition can be used. There are eight schema types that you can use:

- `String`—Any string, UTF-8 encoded
- `Number`—Mongoose doesn't support long or double numbers, but it can be extended to do so using Mongoose plugins; the default support is enough for most cases
- `Date`—Typically returned from MongoDB as an `ISODate` object
- `Boolean`—True or false
- `Buffer`—For binary information such as images

(continued)

- `Mixed`—Any data type
- `Array`—Can either be an array of the same data type, or an array of nested subdocuments
- `ObjectId`—For a unique ID in a path other than `_id`; typically used to reference `_id` paths in other documents

If you do need to use a different schema type it's possible to write your own custom schema types or to use an existing Mongoose plugin from http://plugins.mongoosejs.com.

The path name follows JavaScript object definition conventions and requirements. So there are no spaces or special characters and you should try to avoid reserved words. My convention is to use camelCase for path names. If you're using an existing database use the names of the paths already in the documents. If you're creating a new database, the path names in the schema will be used in the documents, so think carefully.

The properties object is essentially another JavaScript object. This one defines the characteristics of the data held in the path. At a minimum this contains the data type, but it can include validation characteristics, boundaries, default values, and more. We'll explore and use some of these options over the next few chapters as we turn Loc8r into a data-driven application.

But let's get moving and start defining the schemas we want in the application.

5.3 Defining simple Mongoose schemas

We've just discussed that a Mongoose schema is essentially a JavaScript object, which we define from within the application. Let's start by setting up and including the file so that it's done and out of the way, leaving us to concentrate on the schema.

As you'd expect we're going to define the schema in the model folder alongside db.js. In fact, we're going to `require` it into db.js to expose it to the application. So inside the models folder in app_server create a new empty file called locations.js. You need Mongoose to define a Mongoose schema, naturally, so enter the following line into locations.js:

```
var mongoose = require( 'mongoose' );
```

We're going to bring this file into the application by requiring it in db.js, so at the very end of db.js add the following line:

```
require('./locations');
```

And with that, we're set up and ready to go.

5.3.1 The basics of setting up a schema

Mongoose gives you a constructor function for defining new schemas, which you typically assign to a variable so that you can access it later. It looks like the following line:

```
var locationSchema = new mongoose.Schema({ });
```

In fact, that's exactly the construct we're going to use, so go ahead and add that to the locations.js model, below the line requiring Mongoose, of course. The empty object inside the `mongooseSchema({ })` brackets is where we'll define the schema.

DEFINING A SCHEMA FROM CONTROLLER DATA

One of the outcomes in moving the data back from the view to the controller is that the controller ends up giving us a good idea of the data structure we need. Let's start simple and take a look at the `homelist` controller in app_server/controllers/locations.js. The `homelist` controller passes the data to be shown on the homepage into the view. Figure 5.8 shows how one of the locations looks on the homepage.

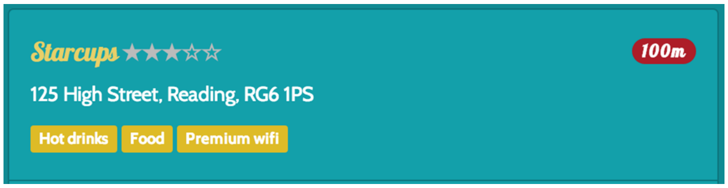


Figure 5.8 A single location as displayed on the homepage list

The following code snippet shows the data for this location, as found in the controller:

```
locations: [{
  name: 'Starcups',                                    <- name is a string
  address: '125 High Street, Reading, RG6 1PS',        <- address is another string
  rating: 3,                                           <- rating is a number
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],  <- facilities is an array of strings
  distance: '100m'
}]
```

We'll come back to the distance a bit later, as that will need to be calculated. The other four data items are fairly straightforward: two strings, one number, and one array of strings. Taking what you know so far you can use this information to define a basic schema, like in the following:

```
var locationSchema = new mongoose.Schema({
  name: String,
  address: String,
  rating: Number,
  facilities: [String]     <- (1) Declare an array of same schema type by declaring that type inside square brackets
});
```

Note the simple approach to declaring facilities as an array ❶. If your array will only contain one schema type, such as String, then you can simply define it by wrapping the schema type in square brackets.

ASSIGNING DEFAULT VALUES

In some cases it's useful to set a default value when a new MongoDB document is created based on your schema. In the locationSchema the rating path is a good candidate for this. When a new location is added to the database, it won't have had any reviews, so it won't have a rating. But our view expects a rating between zero and five stars, which is what the controller will need to pass through.

So what we'd like to do is set a default value of 0 for the rating on each new document. Mongoose lets you do this from within the schema. Remember how rating: Number is shorthand for rating: {type: Number}? Well you can add other options to the definition object, including a default value. This means that you can update the rating path in the schema as follows:

```
rating: {type: Number, "default": 0}
```

The word default doesn't *have* to be in quotes, but it's a reserved word in JavaScript so it's a good idea to do so.

ADDING SOME BASIC VALIDATION: REQUIRED FIELDS

Through Mongoose you can quickly add some basic validation at the schema level. This helps toward maintaining data integrity and can protect your database from problems of missing or malformed data. Mongoose's helpers make it really easy to add some of the most common validation tasks, meaning that you don't have to write or import the code each time.

The first example of this type of validation ensures that required fields aren't empty before saving the document to the database. Rather than writing the checks for each required field in code, you can simply add a required: true flag to the definition objects of each path that you decide should be mandatory. In the location-Schema, we certainly want to ensure that each location has a name, so we can update the name path like this:

```
name: {type: String, required: true}
```

If you try to save a location without a name, Mongoose will return a validation error that you can capture immediately in your code, without needing a roundtrip to the database.

ADDING SOME BASIC VALIDATION: NUMBER BOUNDARIES

You can also use a similar technique to define the maximum and minimum values you want for a number path. These validators are called max and min. Each location we have has a rating assigned to it, which we have just given a default value of 0. The value should never be less than 0 or greater than 5, so you can update the rating path as follows:

```
rating: {type: Number, "default": 0, min: 0, max: 5}
```

With this update Mongoose will not let you save a rating value of less than `0` or greater than `5`. It will return a validation error that you can handle in your code. One great thing about this approach is that the application doesn't have to make a roundtrip to the database to check the boundaries. Another bonus is that you don't have to write validation code into every place in the application where you might add, update, or calculate a rating value.

5.3.2 Using geographic data in MongoDB and Mongoose

When we first started to map our application's data from the controller into a Mongoose schema we left the question of distance until later. Now it's time to discuss how we're going to handle geographic information.

MongoDB can store geographic data as longitude and latitude coordinates, and can even create and manage an *index* based on this. This ability, in turn, enables users to do fast searches of places that are near to each other, or near a specific longitude and latitude. This is very helpful indeed for building a location-based application!

About MongoDB indexes

Indexes in any database system enable faster and more efficient queries, and MongoDB is no different. When a path is indexed, MongoDB can use this index to quickly grab subsets of data without having to scan through all documents in a collection.

Think of a filing system you might have at home, and imagine you need to find a particular credit card statement. You might keep all of your paperwork in one drawer or cabinet. If it's all just thrown in there randomly you'll have to sort through all types of irrelevant documents until you find what you're looking for. If you've "indexed" your paperwork into folders, you can quickly find your "credit card" folder. Once you've picked this out you just look through this one set of documents, making your search much more efficient.

This is akin to how indexing works in a database. In a database, though, you can have more than one index for each document, enabling you to search efficiently on different queries.

Indexes do take maintenance and database resources, though, just as it takes time to correctly file your paperwork. So for best overall performance, try to limit your database indexes to the paths that really need indexing and are used for most queries.

The data for a single geographical location is stored according to the GeoJSON format specification, which we'll see in action shortly. Mongoose supports this data type allowing you to define a geospatial path inside a schema. As Mongoose is an abstraction layer on top of MongoDB it strives to make things easier for you. All you have to do to add a GeoJSON path in your schema is

1. Define the path as an array of the `Number` type.
2. Define the path as having a `2dsphere` index.

To put this into action you can add a `coords` path to your location schema. If you follow the two preceding steps, your schema should be looking like the following code snippet:

```
var locationSchema = new mongoose.Schema({
  name: {type: String, required: true},
  address: String,
  rating: {type: Number, "default": 0, min: 0, max: 5},
  facilities: [String],
  coords: {type: [Number], index: '2dsphere'}
});
```

The `2dsphere` here is the critical part, as that's what enables MongoDB to do the correct calculations when running queries and returning results. It allows MongoDB to calculate geometries based on a spherical object. We'll work more with this in chapter 6 when we build our API and start to interact with the data.

TIP To meet the GeoJSON specification, a coordinate pair must be entered into the array in the correct order: longitude then latitude.

We've now got the basics covered and our schema for Loc8r currently holds everything needed to satisfy the homepage requirements. Next it's time to take a look at the Details page. This page has more complex data requirements, and we'll see how to handle them with Mongoose schemas.

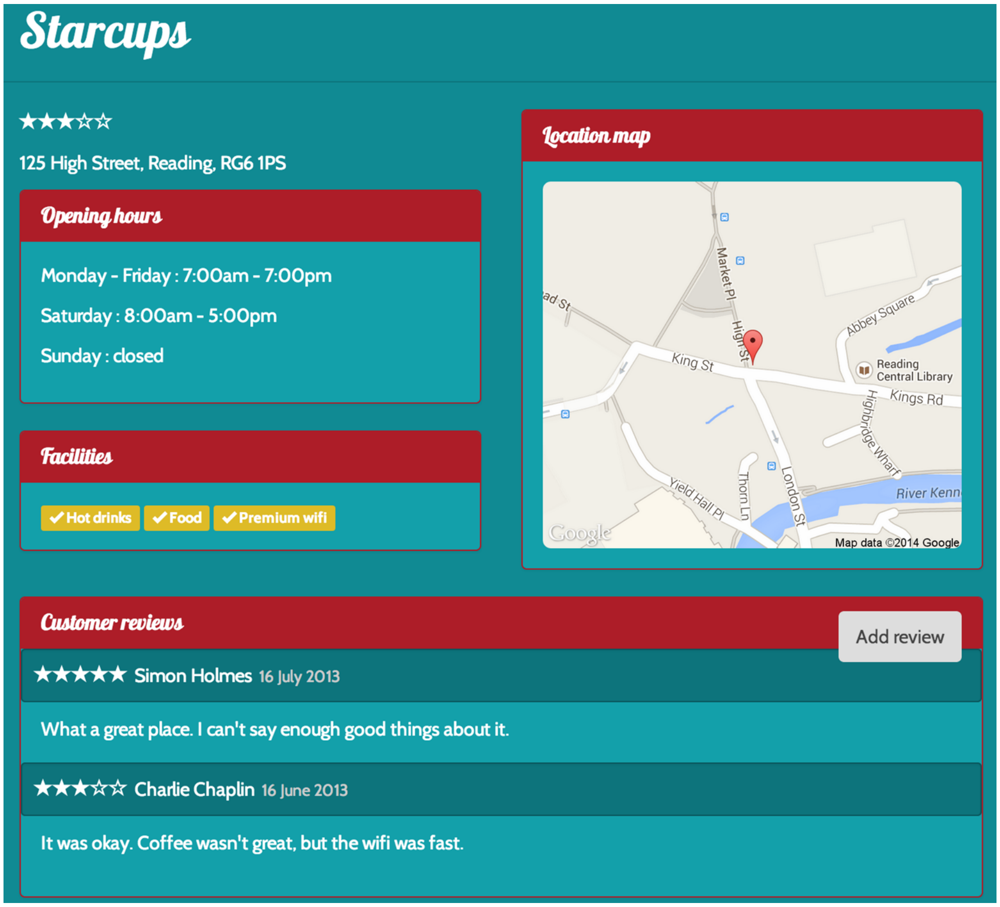
5.3.3 Creating more complex schemas with subdocuments

The data we've used so far has been pretty simple, and can be held in a fairly flat schema. We've used a couple of arrays for the facilities and location coordinates, but again those arrays are simple, containing just a single data type each. Now we're going to look at what happens when we have a slightly more complicated data set to work with.

Let's start by reacquainting ourselves with the Details page, and the data that it shows. Figure 5.9 shows a screenshot of the page and shows all the different areas of information.

The name, rating, and address are right at the top, and a little further down are the facilities. On the right side there's a map, based on the geographic coordinates. All of this we've covered already with the basic schema. The two areas that we don't have anything for are *opening hours* and *customer reviews.*

The data powering this view is currently held in the `locationInfo` controller in app_server/controllers/locations.js. The following listing shows the relevant portion of the data in this controller.



Starcups

★★★☆☆

125 High Street, Reading, RG6 1PS

Opening hours

Monday - Friday : 7:00am - 7:00pm

Saturday : 8:00am - 5:00pm

Sunday : closed

Facilities

✓Hot drinks ✓Food ✓Premium wifi

Location map

Customer reviews

Add review

★★★★★ Simon Holmes 16 July 2013

What a great place. I can't say enough good things about it.

★★★☆☆ Charlie Chaplin 16 June 2013

It was okay. Coffee wasn't great, but the wifi was fast.

Figure 5.9 The information displayed for a single location on the Details page

Listing 5.2 Data in the controller powering the Details page

```
location: {
  name: 'Starcups',
  address: '125 High Street, Reading, RG6 1PS',
  rating: 3,
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  coords: {lat: 51.455041, lng: -0.9690884},
  openingTimes: [{
    days: 'Monday - Friday',
    opening: '7:00am',
    closing: '7:00pm',
    closed: false
  },{
    days: 'Saturday',
    opening: '8:00am',
```

Already covered with existing schema

Data for opening hours is held as an array of objects

```
    closing: '5:00pm',
    closed: false
  },{
    days: 'Sunday',
    closed: true
  }],
  reviews: [{
    author: 'Simon Holmes',
    rating: 5,
    timestamp: '16 July 2013',
    reviewText: 'What a great place. I can\'t say enough good things about it.'
  },{
    author: 'Charlie Chaplin',
    rating: 3,
    timestamp: '16 June 2013',
    reviewText: 'It was okay. Coffee wasn\'t great, but the wifi was fast.'
  }]
}
```

Data for opening hours is held as an array of objects

Reviews are also passed to the view as array of objects

So here we have arrays of objects for the opening hours and for the reviews. In a relational database you'd create these as separate tables, and join them together in a query when you need the information. But that's not how document databases work, including MongoDB. In a document database anything that belongs specifically to a parent document should be contained *within* that document. Figure 5.10 illustrates the conceptual difference between the two approaches.

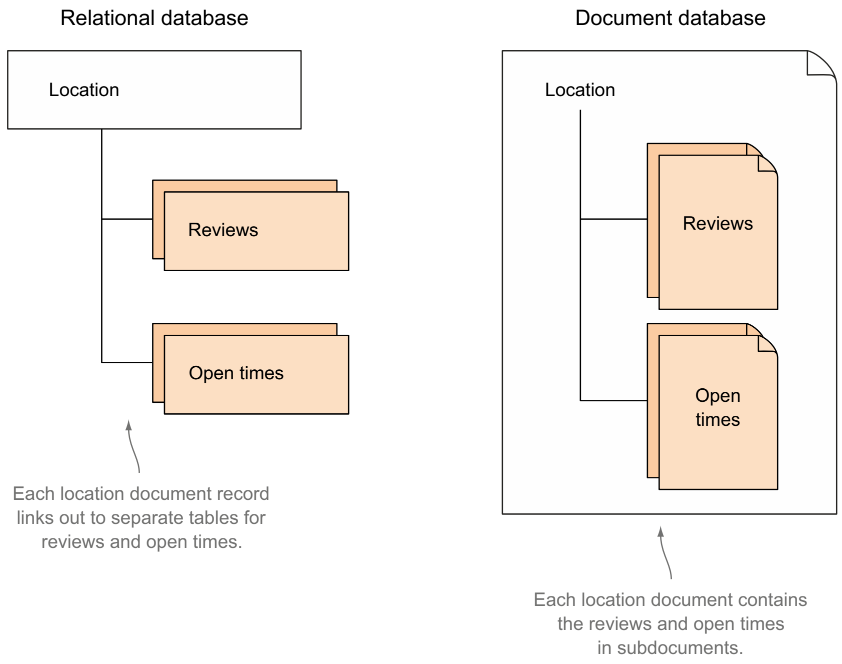

Relational database
Document database
Location
Reviews
Open times
Location
Reviews
Open times
Each location document record links out to separate tables for reviews and open times.
Each location document contains the reviews and open times in subdocuments.

Figure 5.10 Differences between how a relational database and document database store repeating information relating to a parent element

MongoDB offers the concept of *subdocuments* to store this repeating, nested data. Subdocuments are very much like documents in that they have their own schema and each is given a unique `_id` by MongoDB when created. But subdocuments are nested inside a document and they can only be accessed as a path of that parent document.

Using nested schemas in Mongoose to define subdocuments

Subdocuments are defined in Mongoose by using nested schemas. So that's one schema nested inside another. Let's create one to see how that works in code. The first step is to define a new schema for a subdocument. We'll start with the opening times and create the following schema. Note that this needs to be in the same file as the `locationSchema` definition, and, importantly, must be *before* the `locationSchema` definition.

```
var openingTimeSchema = new mongoose.Schema({
  days: {type: String, required: true},
  opening: String,
  closing: String,
  closed: {type: Boolean, required: true}
});
```

Options for storing time information

In the opening time schema we have an interesting situation where we want to save time information, such as 7:30 a.m., but without a date associated with it.

Here we're using a `String` method, as it doesn't require any processing before being put into the database or after being retrieved. It also makes each record easy to understand. The downside is that it would make it harder to do any computational processing with it.

One option is to create a date object with an arbitrary data value assigned to it, and manually set the hours and minutes, such as

```
var d = new Date();
d.setHours(15);
d.setMinutes(30);
```

d is now Wed Apr 09 2014 15:30:40 GMT+0100 (BST)

Using this method we could easily extract the time from the data. The downside is storing unnecessary data, and it's technically incorrect.

A second option is to store the number of minutes since midnight. So 7:30 a.m. is (7 × 60) + 30 = 450. This is a fairly simple computation to make when putting data into the database and pulling it back out again. But the data at a glance is meaningless.

But this second option would be my preference for making the dates smarter and could be a good extension if you want to try out something new. For the sake of readability and avoiding distractions we'll keep using the `String` method through the book.

This schema definition is again pretty simple, and maps over from the data in the controller. We have two required fields, the `closed` Boolean flag and the `days` each subdocument is referring to.

Nesting this schema inside the location schema is another straightforward task. We need to add a new path to the parent schema, and define it as an array of our subdocument schema. The following code snippet shows how to nest the `openingTimeSchema` inside the `locationSchema`:

```
var locationSchema = new mongoose.Schema({
  name: {type: String, required: true},
  address: String,
  rating: {type: Number, "default": 0, min: 0, max: 5},
  facilities: [String],
  coords: {type: [Number], index: '2dsphere'},
  openingTimes: [openingTimeSchema]
});
```

Add nested schema by referencing another schema object as an array

With this in place we could now add multiple opening time subdocuments to a given location, and they would be stored within that location document. An example document from MongoDB based on this schema is shown in the following code snippet, with the subdocuments for the opening times in bold:

```
{
  "_id": ObjectId("52ef3a9f79c44a86710fe7f5"),
  "name": "Starcups",
  "address": "125 High Street, Reading, RG6 1PS",
  "rating": 3,
  "facilities": ["Hot drinks", "Food", "Premium wifi"],
  "coords": [-0.9690884, 51.455041],
  "openingTimes": [{
    "_id": ObjectId("52ef3a9f79c44a86710fe7f6"),
    "days": "Monday - Friday",
    "opening": "7:00am",
    "closing": "7:00pm",
    "closed": false
  }, {
    "_id": ObjectId("52ef3a9f79c44a86710fe7f7"),
    "days": "Saturday",
    "opening": "8:00am",
    "closing": "5:00pm",
    "closed": false
  }, {
    "_id": ObjectId("52ef3a9f79c44a86710fe7f8"),
    "days": "Sunday",
    "closed": true
  }]
}
```

In a MongoDB document nested opening times subdocuments live inside location document

With the schema for the opening times taken care of, we'll move on and look at adding a schema for the review subdocuments.

Adding a second set of subdocuments

Neither MongoDB nor Mongoose limit the number of subdocument paths in a document. This means we're free to take what we've done for the opening times and replicate the process for the reviews.

Step one: take a look at the data used in a review, shown in the following code snippet:

```
{
  author: 'Simon Holmes',
  rating: 5,
  timestamp: '16 July 2013',
  reviewText: 'What a great place. I can\'t say enough good things about it.'
}
```

Step two: map this into a new `reviewSchema` in app_server/models/location.js:

```
var reviewSchema = new mongoose.Schema({
  author: String,
  rating: {type: Number, required: true, min: 0, max: 5},
  reviewText: String,
  createdOn: {type: Date, "default": Date.now}
});
```

Step three: add this `reviewSchema` as a new path to `locationSchema`:

```
var locationSchema = new mongoose.Schema({
  name: {type: String, required: true},
  address: String,
  rating: {type: Number, "default": 0, min: 0, max: 5},
  facilities: [String],
  coords: {type: [Number], index: '2dsphere'},
  openingTimes: [openingTimeSchema],
  reviews: [reviewSchema]
});
```

Once we've defined the schema for reviews and added it to our main location schema we have everything we need to hold the data for all locations in a structured way.

5.3.4 Final schema

Throughout this section we've done quite a bit in the file, so let's take a look at it all together and see what's what. The following listing shows the contents of the locations.js file in app_server/models, defining the schema for the location data.

Listing 5.3 Final location schema definition, including nested schemas

Require Mongoose so that we can use its methods

```
var mongoose = require( 'mongoose' );

var reviewSchema = new mongoose.Schema({
  author: String
  rating: {type: Number, required: true, min: 0, max: 5},
  reviewText: String,
  createdOn: {type: Date, default: Date.now}
});
```

Define a schema for reviews

```
var openingTimeSchema = new mongoose.Schema({
  days: {type: String, required: true},
  opening: String,
  closing: String,
  closed: {type: Boolean, required: true}
});
```

Define a schema for opening times

```
var locationSchema = new mongoose.Schema({
  name: {type: String, required: true},
  address: String,
  rating: {type: Number, "default": 0, min: 0, max: 5},
  facilities: [String],
  coords: {type: [Number], index: '2dsphere'},
  openingTimes: [openingTimeSchema],
  reviews: [reviewSchema]
});
```

Start main location schema definition

Use 2dsphere to add support for GeoJSON longitude and latitude coordinate pairs

Reference opening times and reviews schemas to add nested subdocuments

Documents and subdocuments all have a schema defining their structure, and we've also added in some default values and basic validation. To make this a bit more real, the following listing shows an example MongoDB document based on this schema.

Listing 5.4 Example MongoDB document based on the location schema

```
{
  "_id": ObjectId("52ef3a9f79c44a86710fe7f5"),
  "name": "Starcups",
  "address": "125 High Street, Reading, RG6 1PS",
  "rating": 3,
  "facilities": ["Hot drinks", "Food", "Premium wifi"],
  "coords": [-0.9690884, 51.455041],
  "openingTimes": [{
    "_id": ObjectId("52ef3a9f79c44a86710fe7f6"),
    "days": "Monday - Friday",
    "opening": "7:00am",
    "closing": "7:00pm",
    "closed": false
  }, {
    "_id": ObjectId("52ef3a9f79c44a86710fe7f7"),
    "days": "Saturday",
    "opening": "8:00am",
    "closing": "5:00pm",
    "closed": false
  }, {
    "_id": ObjectId("52ef3a9f79c44a86710fe7f8"),
    "days": "Sunday",
    "closed": true
  }],
```

Coordinates are stored as a GeoJSON pair [longitude, latitude]

Opening times are stored as nested array of objects—these are subdocuments

```
  "reviews": [{
    "_id": ObjectId("52ef3a9f79c44a86710fe7f9"),
    "author": "Simon Holmes",
    "rating": 5,
    "createdOn": ISODate("2013-07-15T23:00:00Z"),
    "reviewText": "What a great place. I can't say enough good
     things about it."
  }, {
    "_id": ObjectId("52ef3a9f79c44a86710fe7fa"),
    "author": "Charlie Chaplin",
    "rating": 3,
    "createdOn": ISODate("2013-06-15T23:00:00Z"),
    "reviewText": "It was okay. Coffee wasn't great, but the wifi was fast."
  }]
}
```

Reviews are also array of subdocuments

That should give you an idea of what a MongoDB document looks like, including subdocuments, when based on a known schema. In a readable form like this it's a JSON object, although technically MongoDB stores it as BSON, which is Binary JSON.

5.3.5 Compiling Mongoose schemas into models

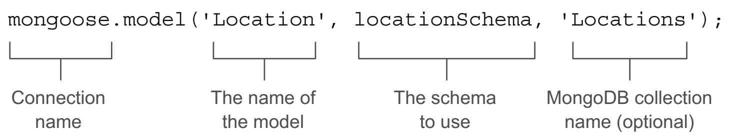
An application doesn't interact with the schema directly when working with data; data interaction is done through models.

In Mongoose, a model is a compiled version of the schema. Once compiled, a single instance of the model maps directly to a single document in your database. It's through this direct one-to-one relationship that the model can create, read, save, and delete data. Figure 5.11 illustrates this arrangement.

This approach makes Mongoose a breeze to work with and we'll really get our teeth into it in chapter 6 when we build the internal API for the application.

Compiling a model from a schema

Anything with the word "compiling" in it tends to sound a bit complicated. In reality, compiling a Mongoose model from a schema is a really simple one-line task. You just need to ensure that the schema is complete before you invoke the `model` command. The `model` command follows this construct:



```
mongoose.model('Location', locationSchema, 'Locations');
```

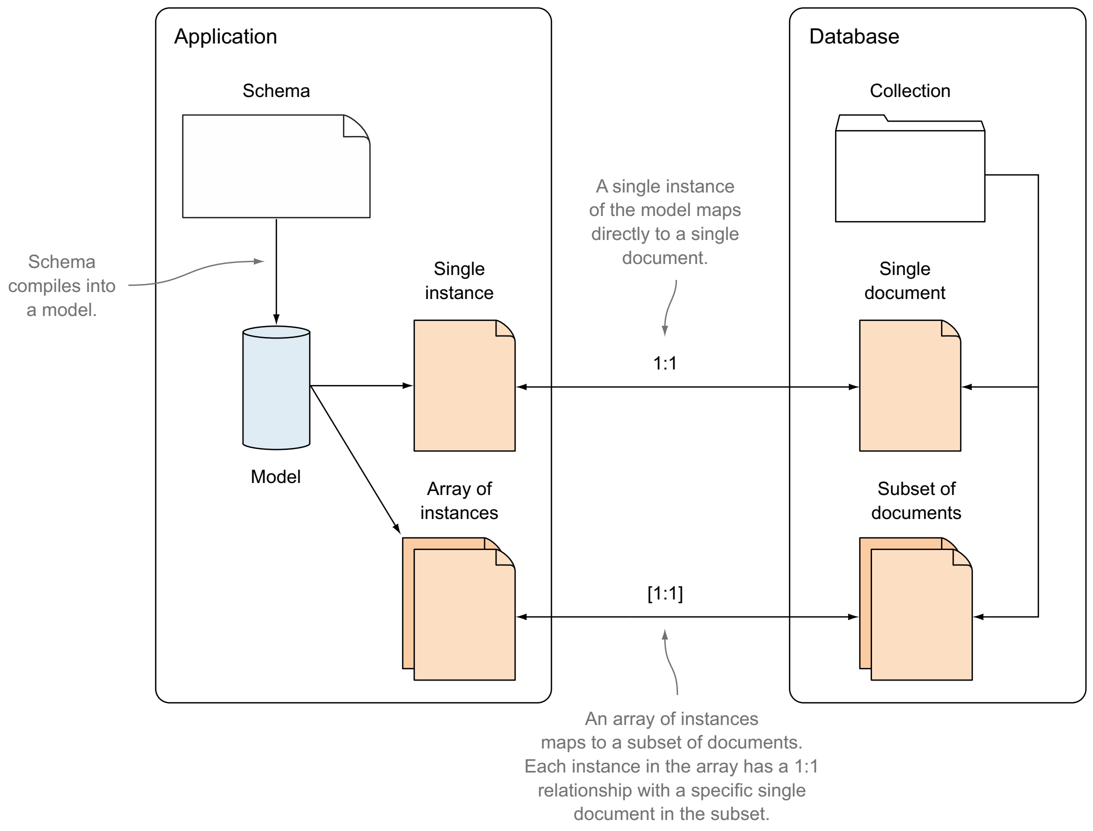

Application
Schema
Schema compiles into a model.
Model
Single instance
Array of instances
A single instance of the model maps directly to a single document.
1:1
[1:1]
An array of instances maps to a subset of documents. Each instance in the array has a 1:1 relationship with a specific single document in the subset.
Database
Collection
Single document
Subset of documents

Figure 5.11 The application and database talk to each other through models. A single instance of a model has a one-to-one relationship with a single document in the database. It's through this relationship that the creating, reading, updating, and deleting of data is managed.

TIP The MongoDB collection name is optional. If you exclude it Mongoose will use a lowercase pluralized version of the model name. For example, a model name of `Location` would look for a collection name of `locations` unless you specify something different.

As we're creating a database and not hooking into an existing data source we can use a default collection name, so we don't need to include that parameter into the `model` command. So to build a model of our location schema we can add the following line to the code, just below the `locationSchema` definition:

```
mongoose.model('Location', locationSchema);
```

That's all there is to it. We've defined a data schema for the locations, and complied the schema into a model that we can use in the application. What we need now is some data.

5.4 Using the MongoDB shell to create a MongoDB database and add data

For building the Loc8r app we're going to create a new database and manually add some test data. This means that you get to create your own personal version of Loc8r for testing, and at the same time get to play directly with MongoDB.

5.4.1 MongoDB shell basics

The MongoDB shell is a command-line utility that gets installed with MongoDB, and allows you to interact with any MongoDB databases on your system. It's quite powerful and can do a lot—we're just going to dip our toes in with the basics to get up and running.

Starting the MongoDB shell

Drop into the shell by running the following line in terminal:

```
$ mongo
```

This should respond in terminal with a couple of lines like these next two, confirming the shell version and that it's connecting to a test database:

```
MongoDB shell version: 2.4.6
connecting to: test
```

TIP When you're in the shell new lines start with a > to differentiate from the standard command-line entry point. The shell commands printed in this section will start with > instead of $ to make it clear that we're using the shell, but like the $ you don't need to type it in.

Listing all local databases

Next is a simple command that will show you a list of all of the local MongoDB databases. Enter the following line into the shell:

```
> show dbs
```

This will return a list of the local MongoDB database names and their sizes. If you haven't created any databases at this point you'll still see the two default ones, something like this:

```
local    0.078125GB
test     (empty)
```

Using a specific database

When starting the MongoDB shell it automatically connects to the empty test database. If you want to switch to a different database, such as the default one called local, you can use the `use` command, like this:

```
> use local
```

The shell will respond with a message, along these lines:

```
switched to db local
```

This message confirms the name of the database the shell has connected to.

LISTING THE COLLECTIONS IN A DATABASE

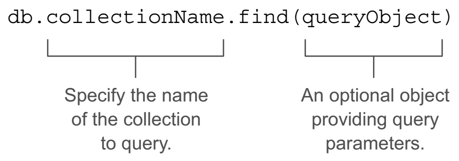
Once you're using a particular database, it's really easy to output a list of the collections using the following command:

```
> show collections
```

If you're using the local database you'll probably see a single collection name output to terminal: `startup_log`.

SEEING THE CONTENTS OF A COLLECTION

The MongoDB shell also lets you query the collections in a database. The construct for a query or find operation is as follows:



```
db.collectionName.find(queryObject)
```

Specify the name of the collection to query.
An optional object providing query parameters.

The `query` object is used to specify what you're trying to find in the collection, and we'll look at examples of this `query` object later in chapter 6 (Mongoose also uses the `query` object). The simplest query is an empty query, which will return all of the documents in a collection. Don't worry if your collection is large, as MongoDB will return a subset of documents that you can page through. Using the `startup_log` collection as an example, you can run the following command:

```
> db.startup_log.find()
```

This will return a number of documents from the MongoDB startup log, the content of which isn't interesting enough to show here. This command is useful for when you're getting your database up and running, and making sure that things are being saved as you expect.

5.4.2 Creating a MongoDB database

You don't actually have to *create* a MongoDB database; you just start to use it. For the Loc8r application it makes sense to have a database called Loc8r. So in the shell, you use it with the following command:

```
> use Loc8r
```

If you run the `show collections` command it won't return anything yet, but if you run `show dbs` you should see that it has been added to the list of databases, and is currently empty:

```
Loc8r    (empty)
local    0.078125GB
test     (empty)
```

This message shows it has been added to the list of databases.

Creating a collection and documents

Similarly, you don't have to explicitly create a collection as MongoDB will create it for you when you first save data into it.

Location data more personal to you

Loc8r is all about location-based data, and the examples are all fictitious places, geographically close to where I live in the United Kingdom. You can make your version more personal to you by changing the names, addresses, and coordinates.

To get your current coordinates you can visit http://whatsmylatlng.com. There's a button on the page to find your location using JavaScript, which will give you a much more accurate location than the first attempt. Note that the coordinates are shown to you in latitude–longitude order, and you need to flip them round for the database, so that longitude is first.

To get the coordinates of any address you can use http://mygeoposition.com. This will let you enter an address or drag and drop a pointer to give you the geographic coordinates. Again, remember that the pairs in MongoDB must be longitude then latitude.

To match the `Location` model you'll want a `locations` collection; remember that the default collection name is a lowercase pluralized version of the model name. You can create and save a new document by passing a data object into the `save` command of a collection, like in the following code snippet:

```
> db.locations.save({                                       <- Note collection name
  name: 'Starcups',                                            specified as part of
  address: '125 High Street, Reading, RG6 1PS',                save command
  rating: 3,
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  coords: [-0.9690884, 51.455041],
  openingTimes: [{
    days: 'Monday - Friday',
    opening: '7:00am',
    closing: '7:00pm',
    closed: false
  }, {
    days: 'Saturday',
    opening: '8:00am',
    closing: '5:00pm',
    closed: false
  }, {
    days: 'Sunday',
    closed: true
  }]
})
```

In one step this will have created a new `locations` collection, and also the first document within the collection. If you run `show collections` in the MongoDB shell now you should see the new `locations` collection being returned, alongside an automatically generated `system.indexes` collection. For example

```
> show collections
locations
system.indexes
```

You can now query the collection to find all of the documents—there's only one in there right now, so the returned information will be quite small. You can use the `find` command on the collection as well:

```
> db.locations.find()
{
  "_id": ObjectId("530efe98d382e7fa4345f173"),
  "address": "125 High Street, Reading, RG6 1PS",
  "coords": [-0.9690884, 51.455041],
  "facilities": ["Hot drinks", "Food", "Premium wifi"],
  "name": "Starcups",
  "openingTimes": [{
    "days": "Monday - Friday",
    "opening": "7:00am",
    "closing": "7:00pm",
    "closed": false
  }, {
    "days": "Saturday",
    "opening": "8:00am",
    "closing": "5:00pm",
    "closed": false
  }, {
    "days": "Sunday",
    "closed": true
  }],
  "rating": 3,
}
```

Remember to run the find operation on collection itself

MongoDB has automatically added a unique identifier for this document

This code snippet has been formatted for readability; the document that MongoDB returns to the shell won't have the line breaks and indentation. But the MongoDB shell can prettify it for you if you add `.pretty()` to the end of the command like this:

```
> db.locations.find().pretty()
```

Notice that the order of the data in the returned document doesn't match the order of the data in the object you supplied. As the data structure isn't column-based it doesn't matter how MongoDB stores the individual paths within a document. The data is always still there in the correct paths, and data held inside arrays always maintains the same order.

Adding subdocuments

You've probably noticed that our first document doesn't have the full data set—there are no review subdocuments. You can actually add these to the initial `save` command like we've done with the opening times, or you can update an existing document and push them in.

MongoDB has an `update` command that accepts two arguments, the first being a query so that it knows which document to update, and the second contains the instructions on what to do when it has found the document. At this point we can do a really simple query and look for the location by name (Starcups), as we know that there aren't any duplicates. For the instruction object we can use a `$push` command to add a new object to the reviews path; it doesn't matter if the reviews path doesn't exist yet, MongoDB will add it as part of the push operation.

Putting it all together shows something like the following code snippet:

```
> db.locations.update({
  name: 'Starcups'
}, {
  $push: {
    reviews: {
      author: 'Simon Holmes',
      id: ObjectId(),
      rating: 5,
      timestamp: new Date("Jul 16, 2013"),
      reviewText: "What a great place. I can't say enough good
      things about it."
    }
  }
})
```

Start with query object to find correct document

When document is found, push a subdocument into the reviews path

Subdocument contains this data

If you run that command in the MongoDB shell while using the `Loc8r` database, it will add a review to the document. You can repeat it as often as you like, changing the data to add multiple reviews.

Note the `new Date` command for setting the timestamp of the review. Using this ensures that MongoDB stores the date as an ISO date object, not a string—this is what our schema expects and allows greater manipulation of dates data.

Repeat the process

These few commands have given us one location to test the application with, but ideally we need a couple more. So go ahead and add some more locations to your database.

When you're done with that and your data is set, you're just about at the point where you can start using it from the application—in this case we'll be building an API. But before we jump into that in chapter 6, there's just one more piece of housekeeping. We want to keep pushing regular updates into Heroku, and now that we've added a database connection and data models to our application we need to make sure that these are supported in Heroku.

5.5 Getting our database live

If you've got your application out in the wild it's no good having your database on your local host. Your database also needs to be externally accessible. In this section we're going to push our database into a live environment, and update our Loc8r application so that it uses the published database from the published site, and the local host database from the development site. We'll start by using the free tier of a service called MongoLab, which can be used as an add-on to Heroku. If you have a different preferred provider or your own database server, that's no problem. The first part of this section runs through setting up on MongoLab, but the following parts—migrating the data and setting the connection strings in the Node application—aren't platform-specific.

5.5.1 Setting up MongoLab and getting the database URI

The first goal is to get an externally accessible database URI so that we can push data to it and add it to the application. We're going to use MongoLab here as it has a good free tier, excellent online documentation, and a very responsive support team.

There are a couple of ways to set up a database on MongoLab. The quickest and easiest way is to use an add-on via Heroku. This is what we'll run through here, but this does require you to register a valid credit card with Heroku. Heroku makes you do this when using add-ons through their ecosystem to protect themselves from abusive behavior. Using the free sandbox tier of MongoLab will not incur any charges. If you're not comfortable doing this, check out the following sidebar for setting up MongoLab manually.

Setting up MongoLab manually

You don't have to use the Heroku add-on system if you don't want to. What you really want to do is to set up a MongoDB database in the cloud and get a connection string for it.

You can follow through the MongoLab documentation to guide you through this: http://docs.mongolab.com/.

In short, the steps are

1. Sign up for a free account.
2. Create a new database (select Single Node, Sandbox for the free tier).
3. Add a user.
4. Get the database URI (connection string).

The connection string will look something like this:

```
mongodb://dbuser:dbpassword@ds059957.mongolab.com:59957/loc8r-dev
```

All of the parts will be different for you, of course, and you'll have to swap out the username and password with what you specified in step 3.

Once you have your full connection string you should save it as part of your Heroku configuration. With a terminal prompt in the root folder of your application you can do this with the following command:

```
$ heroku config:set MONGOLAB_URI=your_db_uri
```

Replace `your_db_uri` with your full connection string, including the `mongodb://` protocol. The quick and easy way automatically creates the `MONGOLAB_URI` setting in your Heroku configuration. These manual steps bring you to the same point as the quick way, and you can now jump back to the main text.

Adding MongoLab to the Heroku application

The quickest way to add MongoLab as a Heroku add-on is through terminal. Make sure you're in the root folder of your application and run the following command:

```
$ heroku addons:add mongolab
```

Unbelievably, that's it! You now have a MongoDB database ready and waiting for you out in the cloud. You can prove this to yourself and open up a web interface to this new database using the following command:

```
$ heroku addons:open mongolab
```

To use the database, you'll need to know its URI.

Getting the database URI

You can get the full database URI also using the command line. This will give you the full connection string that you can use in the application, and also show you the various components that you'll need to push data up to the database.

The command to get the database URI is

```
$ heroku config:get MONGOLAB_URI
```

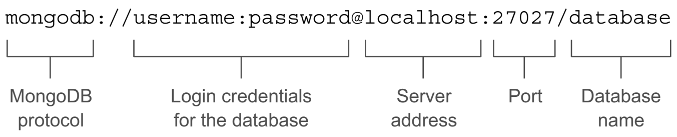
This will output the full connection string, which looks something like this:

```
mongodb://heroku_app20110907:4rqhlidfdqq6vgdi06c15jrlpf@ds033669
.mongolab.com:33669/heroku_app20110907
```

Keep your version handy, as you'll use it in the application soon. First we need to break it down into the components.

Breaking down the URI into its components

This looks like quite a random mess of characters, but we can break it down to make sense of it. From section 5.2.2 we know that this is how a database URI is constructed:



```
mongodb://username:password@localhost:27027/database
```

So taking the URI that MongoLab has given you, you can break it down into something like the following:

- Username: `heroku_app20110907`
- Password: `4rqhlidfdqq6vgdi06c15jrlpf`
- Server address: `ds033669.mongolab.com`
- Port: `33669`
- Database name: `heroku_app20110907`

These are from the example URI, so yours will be different, of course, but make a note of them and they'll be useful.

5.5.2 Pushing up the data

Now that you have an externally accessible database set up, and know all of the details for connecting to it, you can push up data to it. The steps to do this are as follows:

1. Create a temporary directory to hold the data dump.
2. Dump the data from your development Loc8r database.
3. Restore the data to your live database.
4. Test the live database.

All of these steps can be achieved quickly through terminal, so that's what we'll do. It saves jumping around between environments.

CREATING A TEMPORARY FOLDER

A really simple first step, which you can do in your operating system interface if you prefer, is to create a temporary folder into which you can dump your data. The following command does it on Mac or Linux:

```
$ mkdir -p ~/tmp/mongodump
```

Now you have a place for the data dump.

DUMPING THE DATA FROM THE DEVELOPMENT DATABASE

Dumping the data sounds like you're deleting everything from your local development version, but this isn't the case. The process is more of an export than a trashing.

The command used is `mongodump`, which accepts the following three parameters:

- `-h`—The host server (and port)
- `-d`—The database name
- `-o`—The output destination folder

Putting it all together, and using the default MongoDB port of `27017`, you should end up with a command like the following:

```
$ mongodump -h localhost:27017 -d Loc8r -o ~/tmp/mongodump
```

Run that and you have a temporary dump of the data.

Restoring the data to your live database

The process of pushing up the data to your live database is similar, this time using the `mongorestore` command. This command expects the following parameters:

- `-h`—Live host and port
- `-d`—Live database name
- `-u`—Username for the live database
- `-p`—Password for the live database
- Path to the dump directory and database name (this comes at the end of the command and doesn't have a corresponding flag like the other parameters)

Putting all of this together, using the information you have about the database URI, you should have a command like the following:

```
$ mongorestore -h ds033669.mongolab.com:33669 -d heroku_app20110907 -u
heroku_app20110907 -p 4rqhlidfdqq6vgdi06c15jrlpf ~/tmp/mongodump/Loc8r
```

Yours will look a bit different, of course, because you'll have a different host, live database name, username, and password. When you run your `mongorestore` command it will push up the data from the data dump into your live database.

Testing the live database

The MongoDB shell isn't restricted to only accessing databases on your local machine. You can also use the shell to connect to external databases, if you have the right credentials, of course.

To connect the MongoDB shell to an external database you use the same `mongo` command, but add information about the database you want to connect to. You need to include the hostname, port, and database names, and you can supply a username and password if required. This is put together in the following construct:

```
$ mongo hostname:port/database_name -u username -p password
```

For example, using the setup we've been looking at in this section would give you this command:

```
$ mongo ds033669.mongolab.com:33669/heroku_app20110907 -u heroku_app20110907
-p 4rqhlidfdqq6vgdi06c15jrlpf
```

This will connect you to the database through the MongoDB shell. When the connection is established you can use the commands you've already been using to interrogate it, such as

```
> show collections
> db.locations.find()
```

Now you've got two databases and two connection strings; it's important to use the right one at the right time.

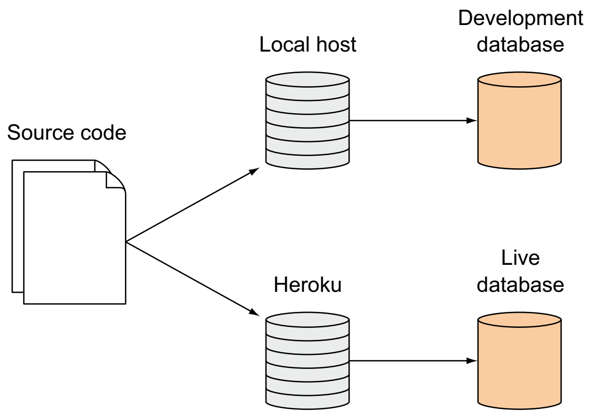


Figure 5.12 The source code runs in two locations, each of which needs to connect to a different database.

5.5.3 Making the application use the right database

So you have your original development database on your local machine plus your new live database up on MongoLab (or elsewhere). We want to keep using the development database while we're developing our application, and we want the live version of our application to use the live database. Yet they both use the same source code. Figure 5.12 shows the issue.

So we have one set of source code running in two environments, each of which should use a different database. The way to handle this is through using a Node environment variable, `NODE_ENV`.

THE NODE_ENV ENVIRONMENT VARIABLE

Environment variables affect the way the core process runs, and the one we're going to look at and use here is `NODE_ENV`. The application already uses `NODE_ENV`; you just don't see it exposed anywhere. By default, Heroku should set `NODE_ENV` to `production` so that the application will run in production mode on their server.

> **Ensuring Heroku is using production mode**
>
> In certain instances, depending on how the application was set up, the Heroku application might not be running in production mode. You can ensure that the Heroku environment variable is set correctly with the following terminal command:
>
> ```
> $ heroku config:set NODE_ENV=production
> ```

You can read `NODE_ENV` from anywhere in the application by using the following statement:

```
process.env.NODE_ENV
```

Unless specified in your environment this will come back as `undefined`. You can specify different environment variables when starting the Node application by prepending the assignment to the launch command. For example

```
$ NODE_ENV=production nodemon
```

This command will start up the application in production mode, and the value of `process.env.NODE_ENV` will be set to `production`.

TIP Don't set `NODE_ENV` from inside the application, only read it.

Setting the database URI based on the environment

The database connection for our application is held in the db.js file in app_server/models. The connection portion of this file currently looks like the following code snippet:

```
var dbURI = 'mongodb://localhost/Loc8r';
mongoose.connect(dbURI);
```

Changing the value of `dbURI` based on the current environment is as simple as using an `if` statement to check `NODE_ENV`. The next code snippet shows how you can do this to pass in your live MongoDB connection. Remember to use your own MongoDB connection string rather than the one in this example.

```
var dbURI = 'mongodb://localhost/Loc8r';
if (process.env.NODE_ENV === 'production') {
  dbURI = 'mongodb://
     heroku_app20110907:4rqhlidfdqq6vgdi06c15jrlpf@ds033669.mongolab.com:3366
     9/heroku_app20110907';
}
mongoose.connect(dbURI);
```

If the source code is going to be in a public repository then you probably don't want to be giving everybody the login credentials to your database. A way around this is to use an environment variable. With MongoLab on Heroku you automatically have one set up—it's how we originally got access to the connection string (if you set your MongoLab account up manually, this is the Heroku configuration variable that you set). If you're using a different provider that hasn't added anything to the Heroku configuration, you can add in your URI with the `heroku config:set` command that we used to ensure Heroku is running in production mode.

The following code snippet shows how you can use the connection string set in the environment variables:

```
var dbURI = 'mongodb://localhost/Loc8r';
if (process.env.NODE_ENV === 'production') {
  dbURI = process.env.MONGOLAB_URI;
}
mongoose.connect(dbURI);
```

This now means that you can share your code, but only you retain access to your database credentials.

TESTING BEFORE LAUNCHING

You can test this update to the code locally before pushing the code to Heroku by setting the environment variable as you start up the application from terminal. The Mongoose connection events we set up earlier output a log to the console when the database connection is made, verifying the URI used.

Starting the application normally from terminal looks like this:

```
$ nodemon
Express server listening on port 3000
Mongoose connected to mongodb://localhost/Loc8r
```

In comparison, starting the application in production mode looks like this:

```
$ NODE_ENV=production nodemon
Express server listening on port 3000
Mongoose connected to mongodb://
heroku_app20110907:4rqhlidfdqq6vgdi06c15jrlpf@ds033669.mongolab.com:33669/
heroku_app20110907
```

When running these commands you'll probably notice that the Mongoose connection confirmation takes longer to appear in the production environment. This is due to the latency of using a separate database server and is why it's a good idea to open the database connection at application startup and leave it open.

Note that the preceding production test may fail on some versions of Windows and the occasional flavor of Linux. This happens when your system is unable to pull down the Heroku configuration variables. You can still test against the production database by prepending the `MONGOLAB_URI` to the application `start` command, which looks like the following code snippet (note that this should all be entered as one line):

```
$ NODE_ENV=production MONGOLAB_URI=mongodb://
<username>:<password>@<hostname>:<port>/<database> nodemon start
```

Whatever OS you're running you should now be able to run the application locally and connect to the production database.

TESTING ON HEROKU

If your local tests are successful, and you can connect to your remote database by temporarily starting the application in production mode, then you're ready to push it up to Heroku. Use the same commands as normal to push the latest version of the code up:

```
$ git add .
$ git commit -m "Commit message here"
$ git push heroku master
```

Heroku lets you easily look at the latest 100 lines of logs by running a terminal command. You can check in those logs to see the output of your console log messages, one of which will be your "Mongoose connected to …" logs. To view the logs run the following command in terminal:

```
$ heroku logs
```

This will output the latest 100 rows to the terminal window, with the very latest messages at the bottom. Scroll up until you find the "Mongoose connected to …" message that looks something like this:

```
2014-03-08T08:19:42.269603+00:00 app[web.1]: Mongoose connected to mongodb://
heroku_app20110907:4rqhlidfdqq6vgdi06c15jrlpf@ds033669.mongolab.com:33669/
heroku_app20110907
```

When you see this, you know that the live application on Heroku is connecting to your live database.

So that's the data defined and modeled, and our Loc8r application is now connected to the database. But we're not interacting with the database at all yet—that comes next!

Get the source code

The source code of the application so far is available from GitHub on the chapter-05 branch of the getting-MEAN repository. In a fresh folder in terminal the following commands will clone it and install the npm module dependencies:

```
$ git clone -b chapter-05 https://github.com/simonholmes/getting-MEAN.git
$ cd getting-MEAN
$ npm install
```

5.6 Summary

In this chapter we've covered

- Using Mongoose to connect an Express application to MongoDB
- Best practices for managing Mongoose connections
- How to model data using Mongoose schemas
- Compiling schemas into models
- Using the MongoDB shell to work directly with the database
- Pushing your database to a live URI
- Connecting to different databases from different environments

Coming up next in chapter 6 we're going to use Express to create a REST API, so that we can then access the database through web services.

WEB DEVELOPMENT

Getting MEAN
with Mongo, Express, Angular, and Node

Simon Holmes

Traditional web dev stacks use a different programming language in every layer, resulting in a complex mashup of code and frameworks. Together, the MongoDB database, the Express and AngularJS frameworks, and Node.js constitute the MEAN stack—a powerful platform that uses only one language, top to bottom: JavaScript. Developers and businesses love it because it's scalable and cost-effective. End users love it because the apps created with it are fast and responsive. It's a win-win-win!

Getting MEAN teaches you how to develop web applications using the MEAN stack. First, you'll create the skeleton of a static site in Express and Node, and then push it up to a live web server. Next, add a MongoDB database and build an API before using Angular to handle data manipulation and application logic in the browser. Finally you'll add an authentication system to the application, using the whole stack. When you finish, you'll have all the skills you need to build a dynamic data-driven web application.

What's Inside

- Full-stack development using JavaScript
- Responsive web techniques
- Everything you need to get started with MEAN
- Best practices for efficiency and reusability

Readers should have some web development experience. This book is based on MongoDB 2, Express 4, Angular 1, and Node.js 4.

Simon Holmes has been a full-stack developer since the late 1990s and runs Full Stack Training Ltd.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/getting-mean-with-mongo-express-angular-and-node

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]

"Looking to go full stack? *Getting MEAN* will take you there."
—Matt Merkes, MyNeighbor

"Fantastic explanations and up-to-date, real-world examples."
—Rambabu Posa
LGL Assessment

"From novice to experienced developer, all who want to use the MEAN stack will get useful advice here."
—Davide Molin
CodingShack.com

"A ground-up explanation of MEAN stack layers."
—Andrea Tarocchi, Red Hat

Free eBook
SEE INSERT

ISBN 13: 978-1-61729-203-3
ISBN 10: 1-61729-203-6
54499
9 781617 292033