UNIT-III

Building a data model with MongoDB and Mongoose: Connecting the Express application to MongoDB by using Mongoose, Benefits of modeling the data, defining simple mongoose schemas, using the MongoDB shell to create a MongoDB database and add data, getting database live.

Writing a REST API: Exposing the MongoDB database to the application: The rules of a REST API, setting up the API in Express, GET methods: Reading data from MongoDB, POST methods: Adding data to MongoDB, PUT methods: Updating data in MongoDB, DELETE method: Deleting data from MongoDB.

Building a data model with MongoDB and Mongoose

Installing MongoDB

- MongoDB is also available for Windows, macOS, and Linux. Detailed instructions about all the following options are available in the documentation at https://docs.mongodb.com/manual/administration/install-community.

INSTALLING MONGODB ON WINDOWS

- Some direct downloads for Windows are available at
- https://docs.mongodb.org/manual/installation, depending on which version of Windows you're running.

Installing MongoDB

CHECKING THE MONGODB VERSION NUMBER

- MongoDB installs not only itself, but also a Mongo shell so that you can interact with your MongoDB databases through the command line.

- To check the **shell version**, run the following command in terminal:

  $ mongo --version

- To check the **version of MongoDB**, run this command:

  $ mongod --version

MVC pattern

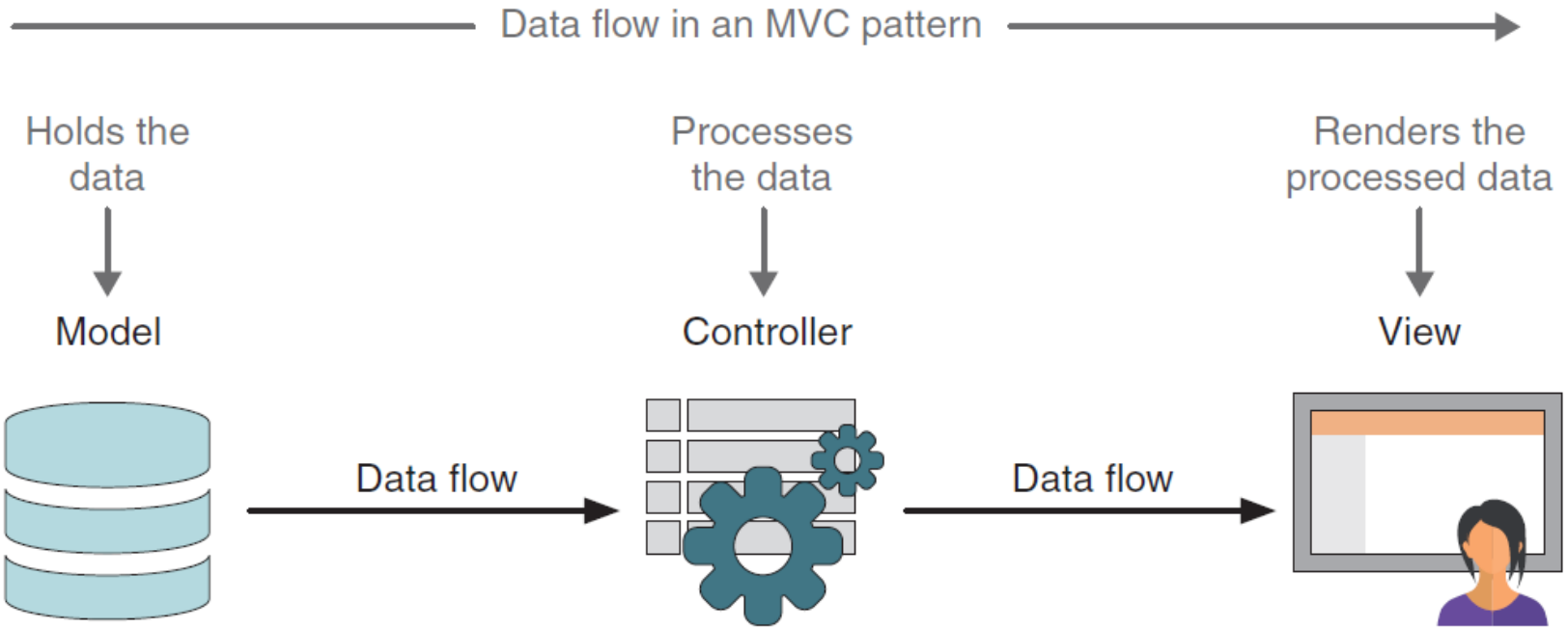


Figure 5.1 In an MVC pattern, data is held in the model, processed by a controller, and then rendered by a view.

4 main steps

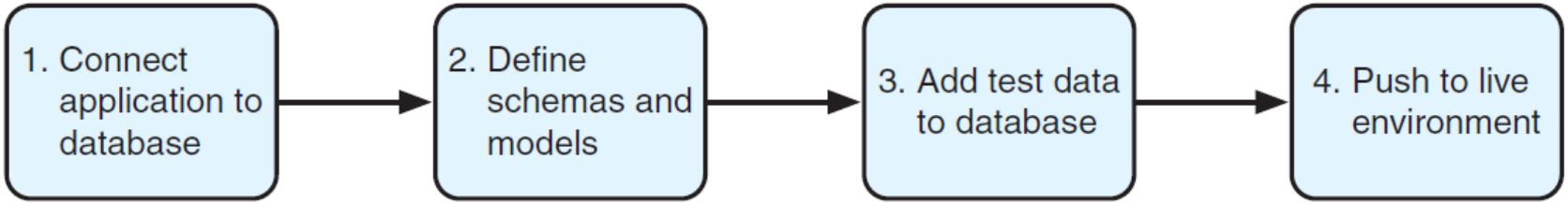


Figure 5.2 Four main steps in this chapter, from connecting your application to a database to pushing the whole thing into a live environment

- You'll start by connecting your application to a database before using Mongoose to define schemas and models.
- When you're happy with the structure, you can add some test data directly to the MongoDB database.
- The final step is making sure that access to the data store also works when pushed up to live.

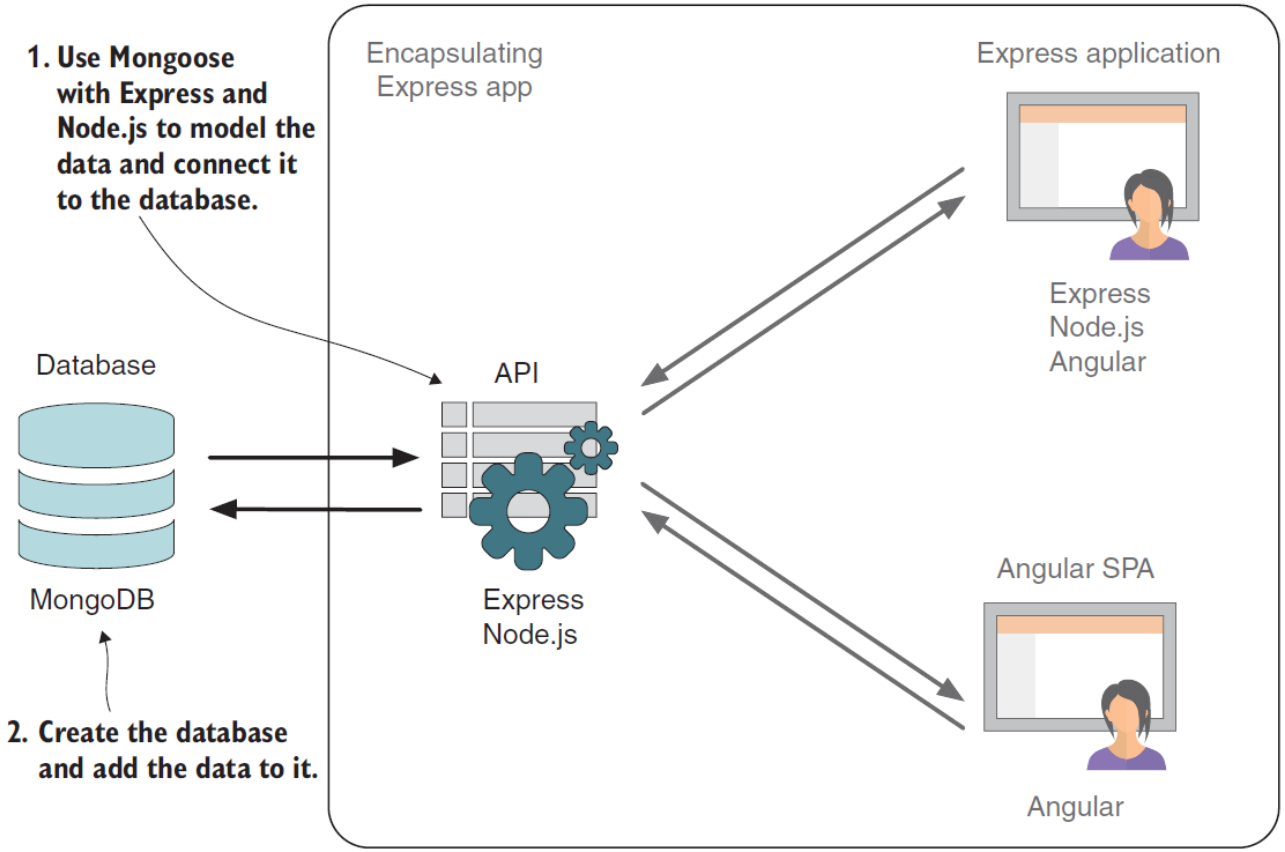


Figure 5.3 Viewing the MongoDB database and using Mongoose inside Express to model the data and manage the connection to the database

1. Connecting the Express application to MongoDB by using Mongoose

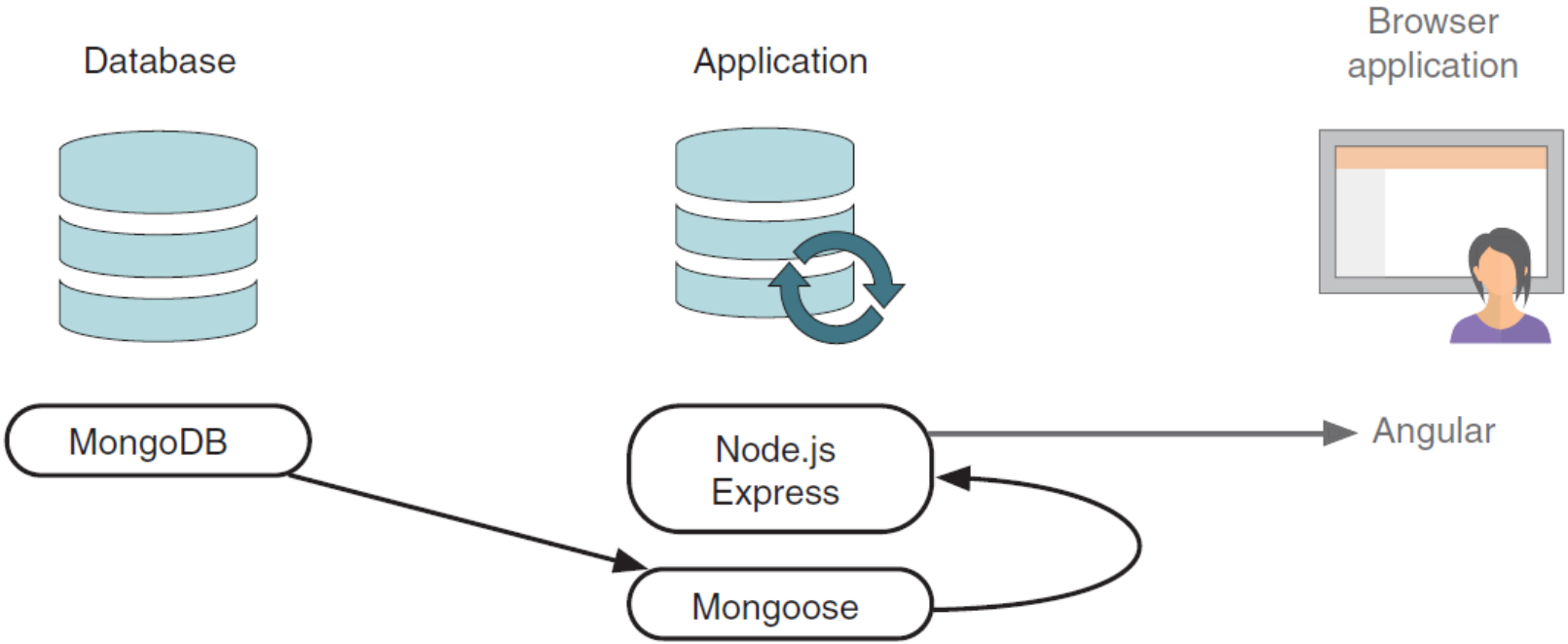


Figure 5.4 The data interactions in the MEAN stack and where Mongoose fits in. The Node/Express application interacts with MongoDB through Mongoose; Node and Express can also talk to Angular.

Adding Mongoose to your application

Mongoose is an **Object Data Modeling (ODM)** library for MongoDB and Node.js. It provides a schema-based solution to model your application data, making it easier to work with MongoDB by enforcing structure and offering powerful features like validation, middleware, and population.

- **$ npm i mongoose**

```
"dependencies": {
  "body-parser": "~1.18.3",
  "cookie-parser": "~1.4.3",
  "debug": "~4.1.0",
  "express": "~4.16.4",
  "mongoose": "^5.3.11",
  "morgan": "~1.9.1",
  "pug": "~2.0.3",
  "serve-favicon": "~2.5.0"
}
```

Adding a Mongoose connection to your application

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 among all requests.

SETTING UP THE CONNECTION FILE

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 1: Create a file called **db.js** in **app_server/models**, and save it. For now, you'll `require` Mongoose in this file with the following single command line:

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

Adding a Mongoose connection to your application

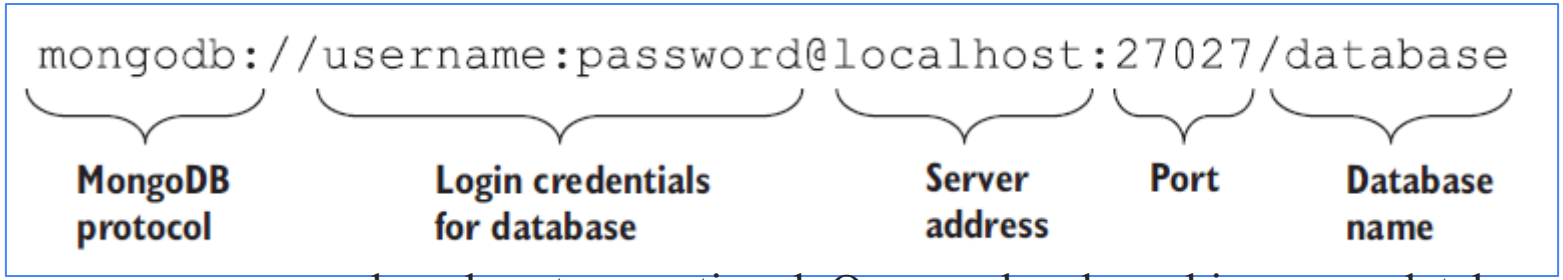
SETTING UP THE CONNECTION FILE

Step 2: 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, you want to do this early in the setup. Amend the top part of **app.js** to look like the following code snippet (modifications in bold):

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

Step-1: 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:



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

```
const dbURI = 'mongodb://localhost/Loc8r';
mongoose.connect(dbURI, {useNewUrlParser: true});
```

Step-2: Monitoring the connection with mongoose connection events

Mongoose publishes events based on the status of the connection, and these events are easy to hook into so that you can see what's going on.

```
mongoose.connection.on('connected', () => {
  console.log(`Mongoose connected to ${dbURI}`);
});
```

Monitors for a successful connection through Mongoose

```
mongoose.connection.on('error', err => {
  console.log('Mongoose connection error:', err);
});
```

Checks for a connection error

```
mongoose.connection.on('disconnected', () => {
  console.log('Mongoose disconnected');
});
```

Checks for a disconnection event

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

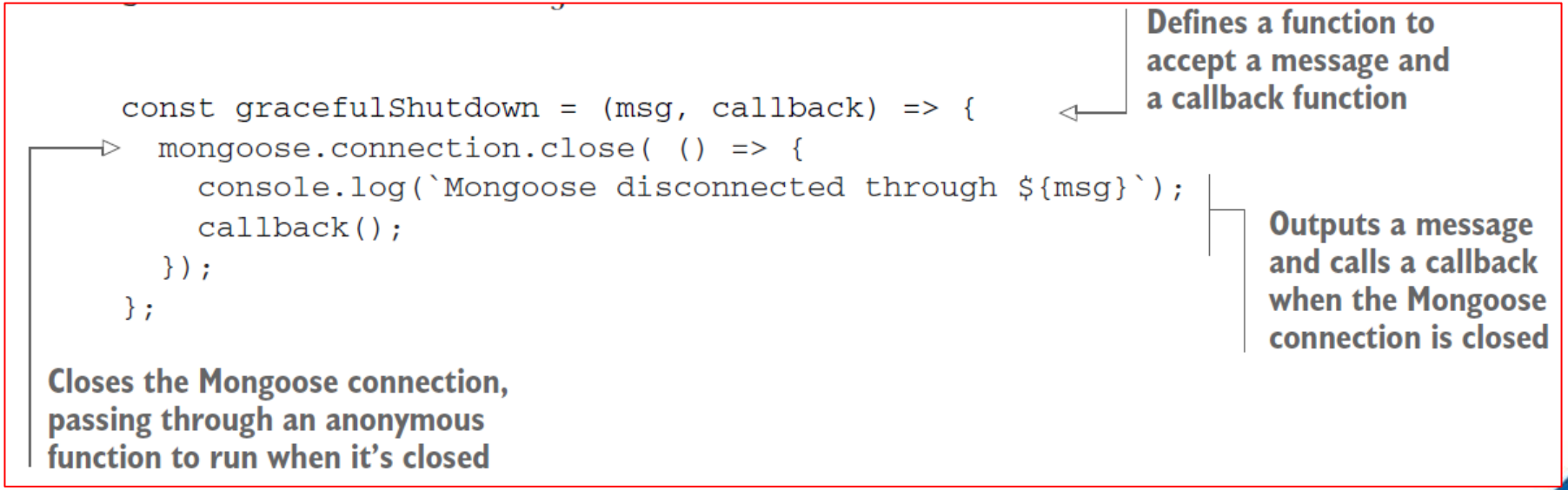
Step-3: CLOSING A MONGOOSE CONNECTION

- Closing the Mongoose connection when the application stops is as much a part of best practices 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, you need to listen to the Node.js process for an event called **SIGINT**.

Step-4: Capturing the process termination events

- Capturing these events prevents the default behavior from happening. You need to make sure that you manually restart the behavior required.



Step-4: Capturing the process termination events

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

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

Listens for SIGUSR2, which is what nodemon uses

Sends a message to graceful-Shutdown and a callback to kill the process, emitting SIGUSR2 again

Listens for SIGINT to be emitted upon application termination

Sends a message to gracefulShutdown and a callback to exit the Node process

Listens for SIGTERM to be emitted when Heroku shuts down the process

Sends a message to gracefulShutdown and a callback to exit the Node process

COMPLETE CONNECTION FILE

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

Altogether, the **db.js** file should look like the following listing. Note that it includes the extra code required by Windows to emit the SIGINT event.

models/db.js

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

```
const mongoose = require('mongoose');
const dbURI = 'mongodb://localhost/Loc8r';
mongoose.connect(dbURI, {useNewUrlParser: true});
mongoose.connection.on('connected', () => {
  console.log(`Mongoose connected to ${dbURI}`);
});
mongoose.connection.on('error', err => {
  console.log(`Mongoose connection error: ${err}`);
});
mongoose.connection.on('disconnected', () => {
  console.log('Mongoose disconnected');
});
```

Defines a database connection string and uses it to open a Mongoose connection

Listens for Mongoose connection events and outputs statuses to the console

models/db.js

```
const gracefulShutdown = (msg, callback) => {
  mongoose.connection.close( () => {
    console.log(`Mongoose disconnected through ${msg}`);
    callback();
  });
};
// For nodemon restarts
process.once('SIGUSR2', () => {
  gracefulShutdown('nodemon restart', () => {
    process.kill(process.pid, 'SIGUSR2');
  });
```

Reusable function to close the Mongoose connection

Listens to Node processes for termination or restart signals and calls the gracefulShutdown function when appropriate, passing a continuation callback

models/db.js

```
});
// For app termination
process.on('SIGINT', () => {
  gracefulShutdown('app termination', () => {
    process.exit(0);
  });
});
// For Heroku app termination
process.on('SIGTERM', () => {
  gracefulShutdown('Heroku app shutdown', () => {
    process.exit(0);
  });
});
```

Listens to Node processes for termination or restart signals and calls the gracefulShutdown function when appropriate, passing a continuation callback

2. Why model the data?

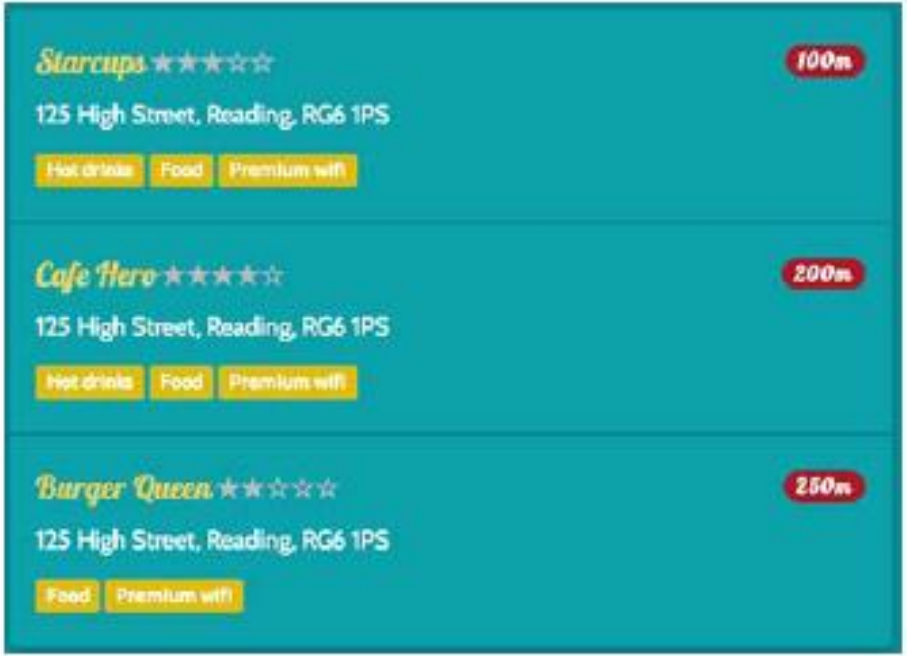


Figure 5.5 Listing section of the homepage has defined data requirements and structure

Move the data to model from controller

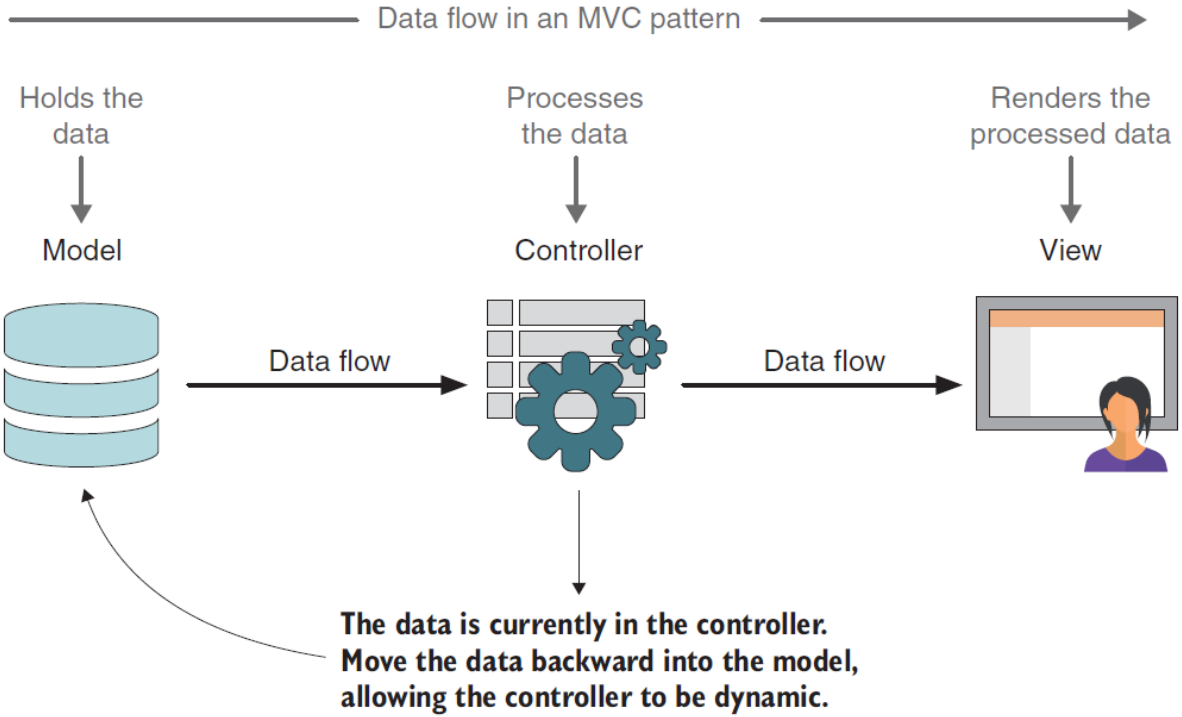


Figure 5.6 How data should flow in an MVC pattern, from the model through the controller and into the view. At this point in your prototype, your data is in the controller, so you want to move it a step back into the model.

What is Mongoose and how does it work?

- Mongoose was built specifically as a **MongoDB Object Document Modeler (ODM)** for Node applications.
- One key principle 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 define your data model in the comfort of your application.

What is Mongoose and how does it work?

First, we'll get some naming conventions out of the way:

- In MongoDB, each entry in a database is called a ***document.***
- In MongoDB, a group 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***.

A ***model*** is the compiled version of a schema. All data interactions using Mongoose go through the model.

What is Mongoose and how does it work?

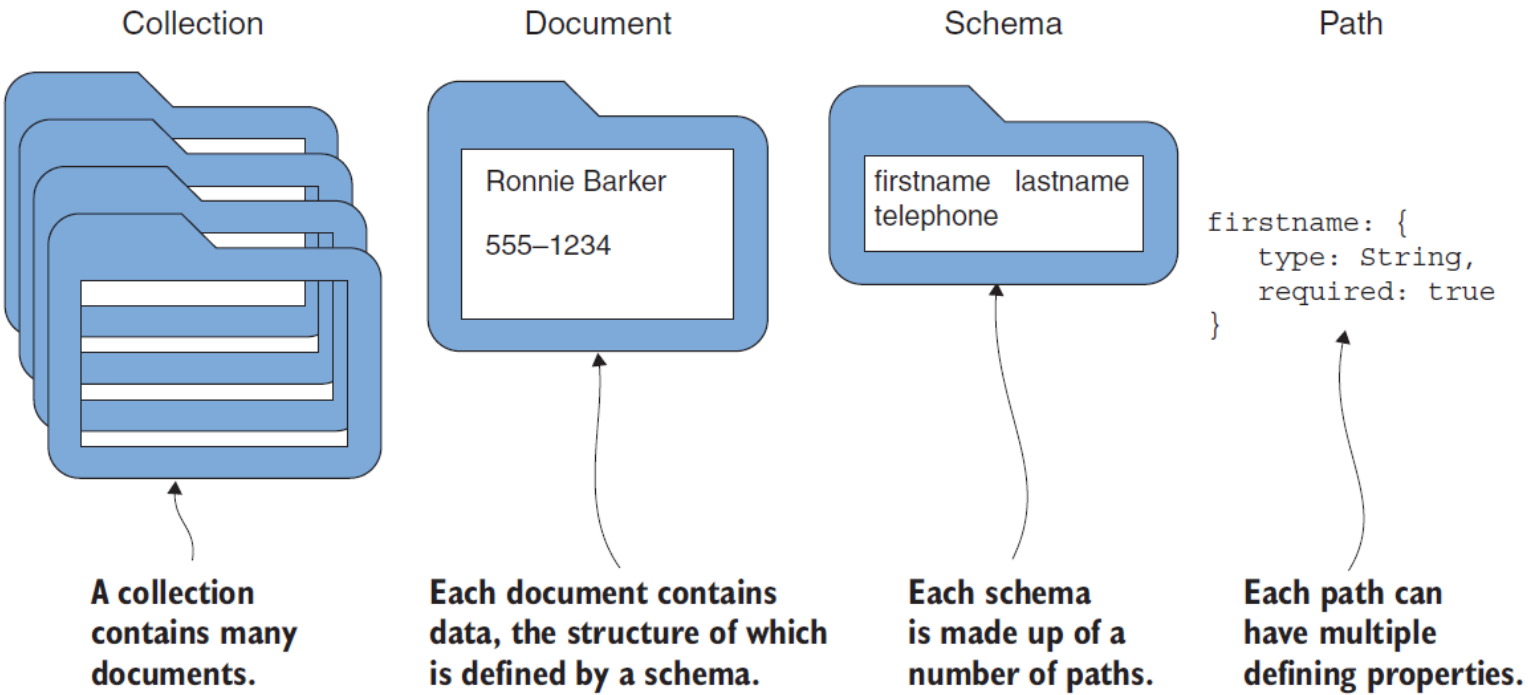


Figure 5.7 Relationships among collections, documents, schemas, and paths in MongoDB and Mongoose, using a business card metaphor

How does Mongoose model data?

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

Breaking down a schema path

- The basic construct for an individual path definition is the pathname followed by a properties object.

- A schema path is constructed of the pathname and the properties object

Allowed schema types

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 using Mongoose plugins; the default support is enough in most
- cases.
- **`Date`**—Typically returned from MongoDB as an **`ISODate`** object.
- **`Boolean`**—True or false.
- **`Buffer`**—For binary information such as images.
- **`Mixed`**—Any data type.
- **`Array`**—Can 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.

3. Defining simple Mongoose schemas

- 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 to **locations.js:**

  const mongoose = require('mongoose');

- You'll bring this file into the application by adding a require in **db.js** for it. At the end of db.js, add the following line:

  require('./locations');

1. Defining A Schema From Controller Data

- Mongoose gives you a constructor function for defining new schemas, which you typically assign to a variable so that you can access it later.
- The **homelist** controller in **app_server/controllers/locations.js**.
- The **homelist** controller passes the data to be shown on the homepage into the view.

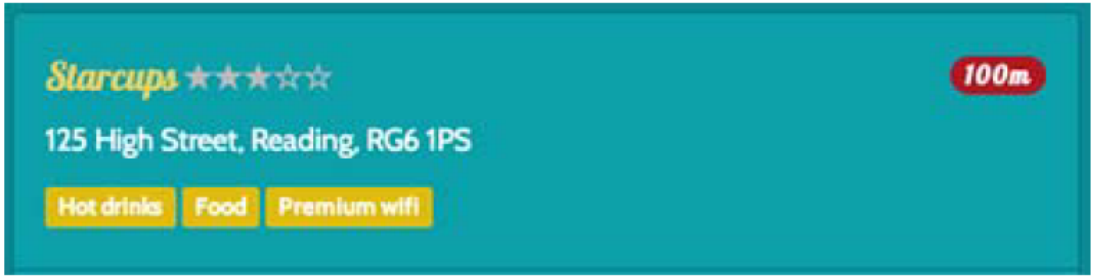


Figure 5.8 A single location as displayed in the homepage list

Defining A Schema From Controller Data

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

```
locations: [{
  name: 'Starcups',
  address: '125 High Street, Reading, RG6 1PS',
  rating: 3,
  facilities: ['Hot drinks', 'Food', 'Premium wifi'],
  distance: '100m'
}]
```

name is a string.

address is another string.

rating is a number.

facilities is an array of strings.

Basic schema

```
const locationSchema = new mongoose.Schema({
  name: String,
  address: String,
  rating: Number,
  facilities: [String]
});
```

1 Declares an array of the same schema type by declaring that type inside square brackets

The basics of setting up a schema

Assigning Default Values

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

- The word ***default*** doesn't *have* to be in quotes, but it's a reserved word in JavaScript; therefore, it's a good idea to use them.

Adding Some Basic Validation: Required Fields

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

Adding Some Basic Validation: Number Boundaries

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

Using geographic data in MongoDB and Mongoose

- MongoDB can store geographic data as longitude and latitude coordinates and can even create and manage an index based on this data.

- The data for a single geographical location is stored according to the GeoJSON format specification.

- add a GeoJSON path in your schema
  1. Define the path as an array of the **Number** type.
  2. Define the path as having a **2dsphere** index.

Using geographic data in MongoDB and Mongoose

add a **coords** path to your location schema

- The **2dsphere** here is the critical part because it enables MongoDB to do the correct calculations when running queries and returning results.
- It allows MongoDB to calculate geometries based on a spherical object.

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

Creating more complex schemas with subdocuments

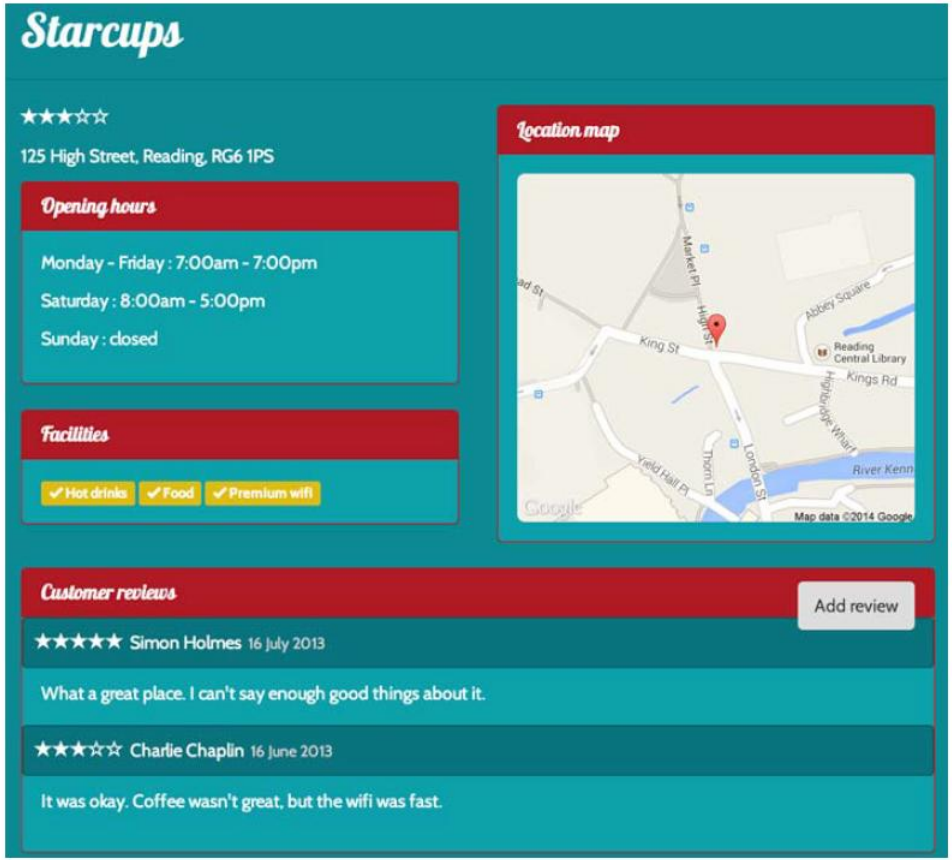


Figure 5.9 The information displayed for a single location on the Details page

Creating more complex schemas with subdocuments

app_server/controllers/locations.js

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},
    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
  }],
```

Already covered with the existing schema

Data for opening hours is held as an array of objects.

Creating more complex schemas with subdocuments

app_server/controllers/locations.js

```
  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.'
  }]
}
```

Reviews are also passed to the view as an array of objects.

Difference between RD and DD

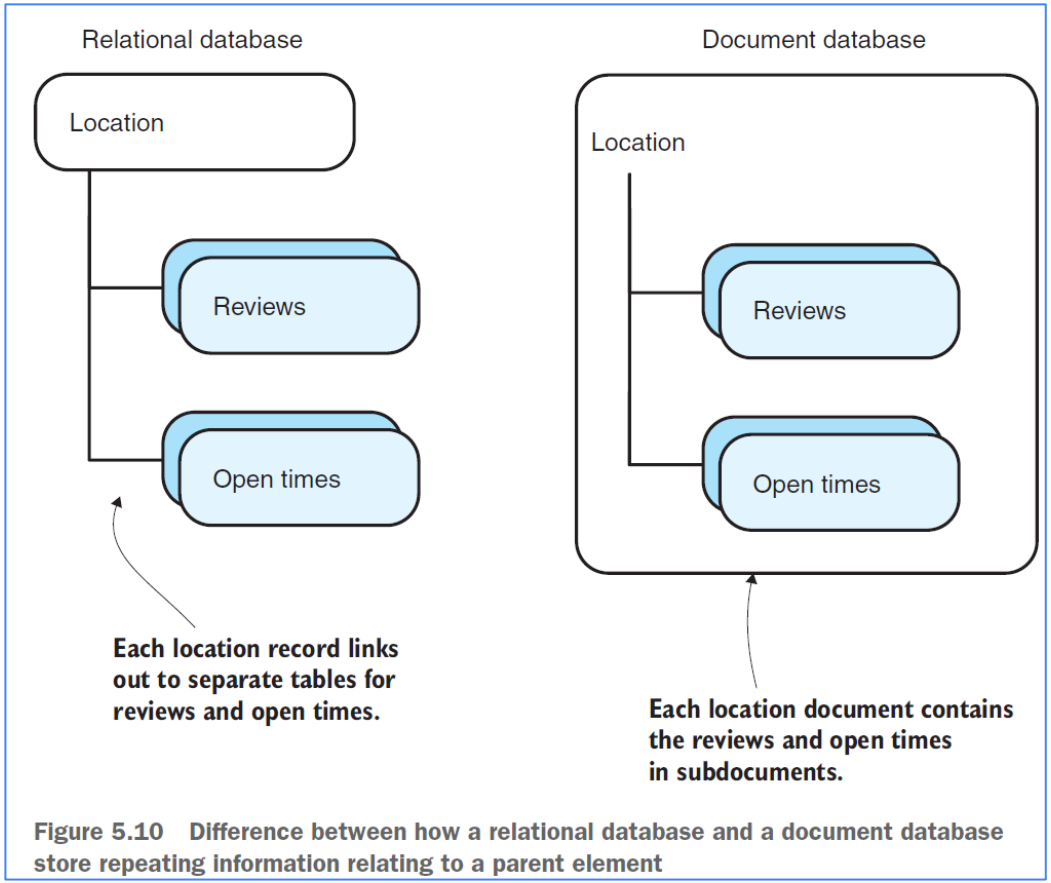


Figure 5.10 Difference between how a relational database and a document database store repeating information relating to a parent element

Creating more complex schemas with subdocuments

Using Nested Schemas In Mongoose To Define Subdocuments

- Subdocuments are defined in Mongoose by nested schemas—one schema nested inside another.
- The first step is defining a new schema for a subdocument.
- Start with the opening times, and create the following schema.
- Note that this schema needs to be in the same file as the ***locationSchema*** definition and (important) must be *before* the ***locationSchema*** definition:

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

Creating more complex schemas with subdocuments

openingTimeSchema inside the locationSchema:

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

Adds nested schema by referencing another schema object as an array

Adding A Second Set Of Subdocuments

Neither MongoDB nor Mongoose limits the number of subdocument paths in a document, so you're free to use what you've done for the opening times and replicate the process for the reviews:

- Step 1: Look at the data used in a review:

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

Adding A Second Set Of Subdocuments

- Step 2: Map this code into a new `reviewSchema` in app_server/models/location.js:

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

Creating more complex schemas with subdocuments

Adding A Second Set Of Subdocuments

- Step 3: Add this `reviewSchema` as a new path to `locationSchema`:

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

When you've defined the schema for reviews and added it to your main location schema, you have everything you need to hold the data for all locations in a structured way.

Final schema

Listing 5.3 Final location schema definition, including nested schemas

```
const mongoose = require( 'mongoose' );
const openingTimeSchema = new
     mongoose.Schema({
  days: {type: String, required: true},
  opening: String,
  closing: String,
  closed: {
    type: Boolean,
    required: true
  }
});
const reviewSchema = new mongoose.Schema({
  author: String,
  rating: {
    type: Number,
    required: true,
    min: 0,
    max: 5
  },
  reviewText: String,
  createdOn: {type: Date, default: Date.now}
});
```

Requires Mongoose so that you can use its methods

Defines a schema for opening times

Defines a schema for reviews

Defines a schema for reviews

Final schema

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

Starts the main location schema definition

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

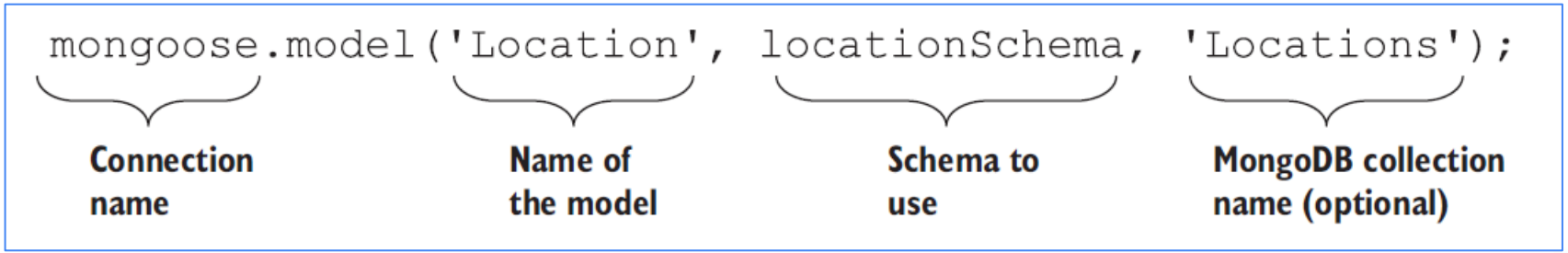
References the opening times and reviews schemas to add nested subdocuments

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.
- When it's 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.

Compiling A Model From A Schema

- In reality, compiling a Mongoose model from a schema is a simple one-line task.
- You need to ensure that the schema is complete before you invoke the model command.
- The model command follows this construct:



Compiling A Model From A Schema

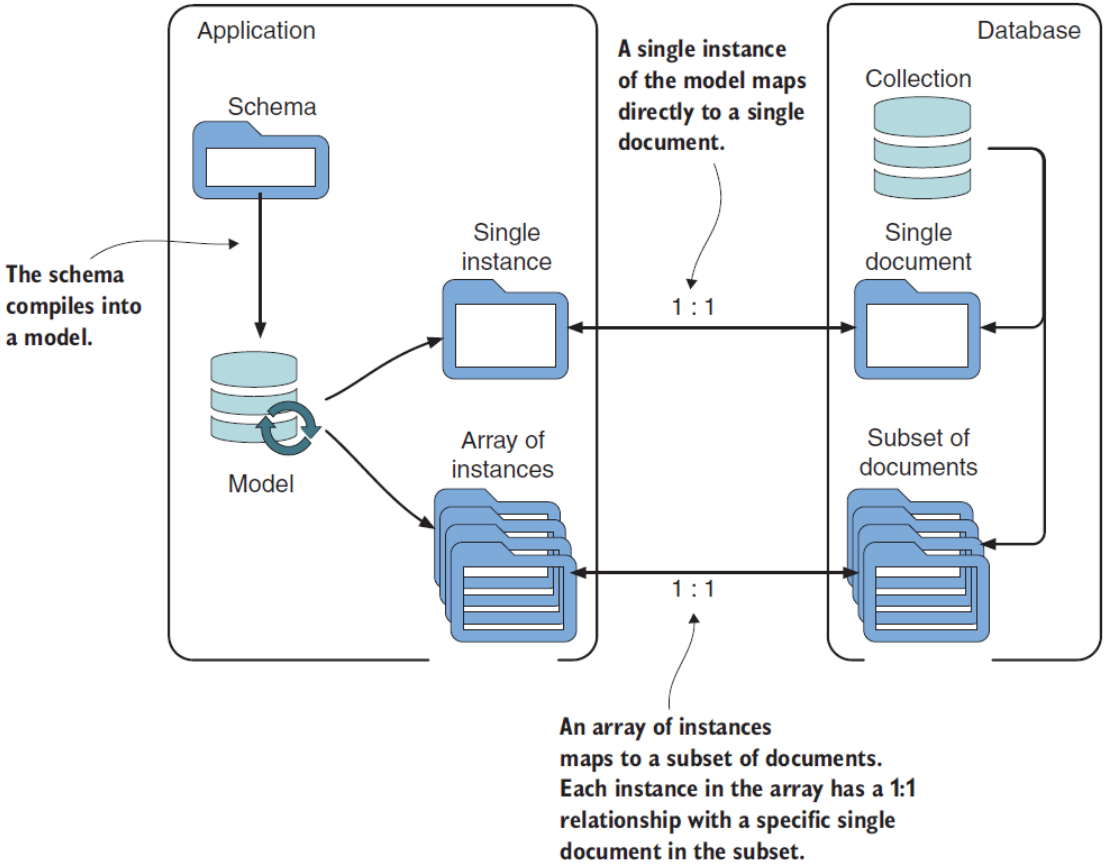


Figure 5.11 The application and the database talk to each other through models. A single instance of a model has a one-to-one relationship with a single document in the database. It's through this relationship that the creating, reading, updating ,and deleting of data are managed.

Compiling Mongoose schemas into models

- The MongoDB collection name is optional.
- If you exclude it, Mongoose uses a lowercase pluralized version of the model name.
- A model name of Location, for example, would look for a collection name of locations unless you specify something different.

Compiling Mongoose schemas into models

- To build a model of your location schema, you can add the following line to the code below the **locationSchema definition**:

  mongoose.model('Location', locationSchema);

- That's all there is to it. You've defined a data schema for the locations and compiled the schema into a model that you can use in the application.

4. Using the MongoDB shell to create a MongoDB database and add data

i. 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.

STARTING THE MONGODB SHELL

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

  $ mongosh

- This command should respond in the terminal with a few lines confirming
  - The shell version
  - The server and port that it's connecting to
  - The server version it has connected to

```
MongoDB shell version 4.0.0
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 4.0.0
```

4. Using the MongoDB shell to create a MongoDB database and add data

ii. Listing All Local Databases

- Next is a simple command that shows a list of all the local MongoDB databases. Enter the following line in the shell:

  > show dbs

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

  admin 0.000GB

  local 0.000GB

4. Using the MongoDB shell to create a MongoDB database and add data

iii. Using A Specific Database

- If you want to use a specific database, such as the default one called local, you can use the use command, like this:

  > use local

- The shell responds with a message along these lines:

  switched to db local

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

iv. Using A Specific Database

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

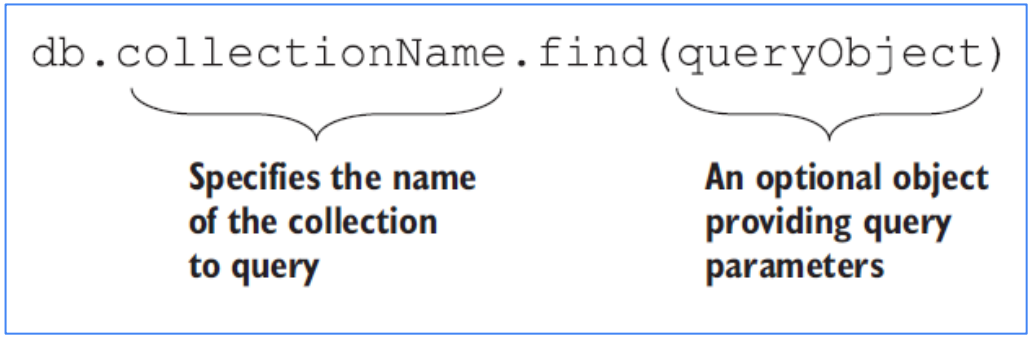
```
> show collections
```

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

4. Using the MongoDB shell to create a MongoDB database and add data

v. 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:



- Using the **startup_log** collection as an example, you can run the following command:

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

- This command returns several documents from the MongoDB startup log, the content of which isn't interesting enough to show here.

Basic Commands

MongoDB Operations – Case Study on Student Information

This demonstrates MongoDB operations using a case study of student information in the database 'drauk'. The collection considered is 'students', with details such as student_id, name, branch, year, email, and cgpa.

1. Create / Switch Database

use drauk

```
> use drauk
< switched to db drauk
```

2. Sample Document

```
{
  "student_id": 101,
  "name": "Alice",
  "branch": "AI",
  "year": 3,
  "email": "alice@example.com",
  "cgpa": 8.9
}
```

ANURAG UNIVERSITY

Basic Commands

3. Insert Operations

1. **Insert a single student record:**

```
db.students.insertOne({
  "student_id": 101,
  "name": "Alice",
  "branch": "AI",
  "year": 3,
  "email": "alice@example.com",
  "cgpa": 8.9
})
```

```
{
  acknowledged: true,
  insertedId: ObjectId('68b6184b5bfc177d5a8d5ec0')
}
```

2. Insert multiple student records:

```
db.students.insertMany([
  {
    "student_id": 102,
    "name": "Bob",
    "branch": "AI",
    "year": 2,
    "email": "bob@example.com",
    "cgpa": 8.2
  },
  {
    "student_id": 103,
    "name": "Charlie",
    "branch": "AI",
    "year": 1,
    "email": "charlie@example.com",
    "cgpa": 7.9
  }
])
```

```
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('68b6188b5bfc177d5a8d5ec1'),
    '1': ObjectId('68b6188b5bfc177d5a8d5ec2')
  }
}
```

Basic Commands

4. Query Operations

1. **Find all students:**
   db.students.find()

2. **Find student with student_id = 101:**
   db.students.findOne({"student_id": 101})

3. **Find students with cgpa greater than 8.0:**
   db.students.find({"cgpa": {$gt: 8.0}})

4. **Find students in branch 'AI' and year = 2:**
   db.students.find({"branch": "AI", "year": 2})

```
> db.students.find({"branch": "AI", "year": 2})
< {
    _id: ObjectId('68b6188b5bfc177d5a8d5ec1'),
    student_id: 102,
    name: 'Bob',
    branch: 'AI',
    year: 2,
    email: 'bob@example.com',
    cgpa: 8.2
  }
```

```
> db.students.findOne({"student_id": 101})
< {
    _id: ObjectId('68b6184b5bfc177d5a8d5ec0'),
    student_id: 101,
    name: 'Alice',
    branch: 'AI',
    year: 3,
    email: 'alice@example.com',
    cgpa: 8.9
  }
```

```
> db.students.find({"cgpa": {$gt: 8.0}})
< {
    _id: ObjectId('68b6184b5bfc177d5a8d5ec0'),
    student_id: 101,
    name: 'Alice',
    branch: 'AI',
    year: 3,
    email: 'alice@example.com',
    cgpa: 8.9
  }
  {
    _id: ObjectId('68b6188b5bfc177d5a8d5ec1'),
    student_id: 102,
    name: 'Bob',
    branch: 'AI',
    year: 2,
    email: 'bob@example.com',
    cgpa: 8.2
  }
```

```
> db.students.find()
< {
    _id: ObjectId('68b6184b5bfc177d5a8d5ec0'),
    student_id: 101,
    name: 'Alice',
    branch: 'AI',
    year: 3,
    email: 'alice@example.com',
    cgpa: 8.9
  }
  {
    _id: ObjectId('68b6188b5bfc177d5a8d5ec1'),
    student_id: 102,
    name: 'Bob',
    branch: 'AI',
    year: 2,
    email: 'bob@example.com',
    cgpa: 8.2
  }
  {
    _id: ObjectId('68b6188b5bfc177d5a8d5ec2'),
    student_id: 103,
    name: 'Charlie',
    branch: 'AI',
    year: 1,
    email: 'charlie@example.com',
    cgpa: 7.9
  }
```

Basic Commands

5. Update Operations

1. **Update email of a student:**
   db.students.updateOne({"student_id": 101}, {$set: {"email": "alice_new@example.com"}})

2. **Increase cgpa of all AI branch students by 0.2:**
   db.students.updateMany({"branch": "AI"}, {$inc: {"cgpa": 0.2}})

6. Delete Operations

1. **Delete one student record:**
   db.students.deleteOne({"student_id": 103})

2. **Delete all students with cgpa less than 6.0:**
   db.students.deleteMany({"cgpa": {$lt: 6.0}})

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

```
> db.students.deleteOne({"student_id": 103})
{
  acknowledged: true,
  deletedCount: 1
}
> db.students.deleteMany({"cgpa": {$lt: 6.0}})
{
  acknowledged: true,
  deletedCount: 0
}
```

Basic Commands

7. Indexing

1. **Create an index on branch:**
   db.students.createIndex({"branch": 1})

2. **Create a compound index on year and cgpa:**
   db.students.createIndex({"year": 1, "cgpa": -1})

3. **View existing indexes:**
   db.students.getIndexes()

```
> db.students.createIndex({"branch": 1})
< branch_1
> db.students.createIndex({"year": 1, "cgpa": -1})
< year_1_cgpa_-1
> db.students.getIndexes()
< [
    { v: 2, key: { _id: 1 }, name: '_id_' },
    { v: 2, key: { branch: 1 }, name: 'branch_1' },
    { v: 2, key: { year: 1, cgpa: -1 }, name: 'year_1_cgpa_-1' }
  ]
```

8. Aggregation

1. **Find average CGPA of AI students:**
   db.students.aggregate([
   { $match: {"branch": "AI"} },
   { $group: { _id: "$branch", avgCGPA: { $avg: "$cgpa" } } }
   ])

```
< {
    _id: 'AI',
    avgCGPA: 8.75
  }
```

2. **Count number of students in each year:**
   db.students.aggregate([
   { $group: { _id: "$year", count: { $sum: 1 } } }
   ])

```
< {
    _id: 2,
    count: 1
  }
  {
    _id: 3,
    count: 1
  }
```

Basic Commands

1. Drop a Collection

If you want to drop the **students'** collection from the drauk database:

```
use drauk
db.students.drop()
```

Output (if collection exists and is dropped):
true
Output (if collection doesn't exist):
false

2. Drop the Entire Database

If you want to drop the **drauk** database (including all its collections):

```
use drauk
db.dropDatabase()
```

Output:
{ "dropped" : "drauk", "ok" : 1 }

Creating a MongoDB database

- You don't have to *create* a MongoDB database; you only need to start using it. For the Loc8r application, it makes sense to have a database called Loc8r. In the shell, you use it with the following command:

  > use Loc8r

Creating A Collection And Documents

- Similarly, you don't have to explicitly create a collection, as MongoDB creates it for you when you first save data to it.
- 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 insert one command of a collection, as in the following code snippet:

CREATING A COLLECTION AND DOCUMENTS

```
db.locations.insertOne({
  name: 'Starcups',
  address: '125 High Street, Reading, RG6 1PS',
  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 }
  ]
})
```

CREATING A COLLECTION AND DOCUMENTS

- you've created the Loc8r database and a new locations collection, and added the first document to the collection.
- If you run show dbs in the MongoDB shell now, you should see the new Loc8r database being returned alongside the other databases, like so:

  > *show dbs*

  Loc8r 0.000GB

  admin 0.000GB

  local 0.000GB

CREATING A COLLECTION AND DOCUMENTS

- Now, when you run show collections in the MongoDB shell, you should see the new locations collection being returned:

> show collections

locations

```
> 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 the collection itself.

MongoDB has automatically added a unique identifier for this document.

ADDING SUBDOCUMENTS

- MongoDB has an *update* command that accepts two arguments: a query so that it knows which document to update, and the instructions on what to do when it finds the document.
- At this point, you can do a simple query and look for the location by name (Starcups), as you know that there aren't any duplicates.
- For the instruction object, you can use a *$push* command to add a new object to the reviews path.

ADDING SUBDOCUMENTS

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("Mar 12, 2017"),
      reviewText: "What a great place."
    }
  }
})
```

Starts with a query object to find correct document

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

Subdocument contains this data

REPEAT THE PROCESS

These few commands have given you one location to test the application with, but ideally, you need a couple more. Add some more locations to your database.

5. Getting your database live

- MongoDB Compass
- MongoDB Atlas

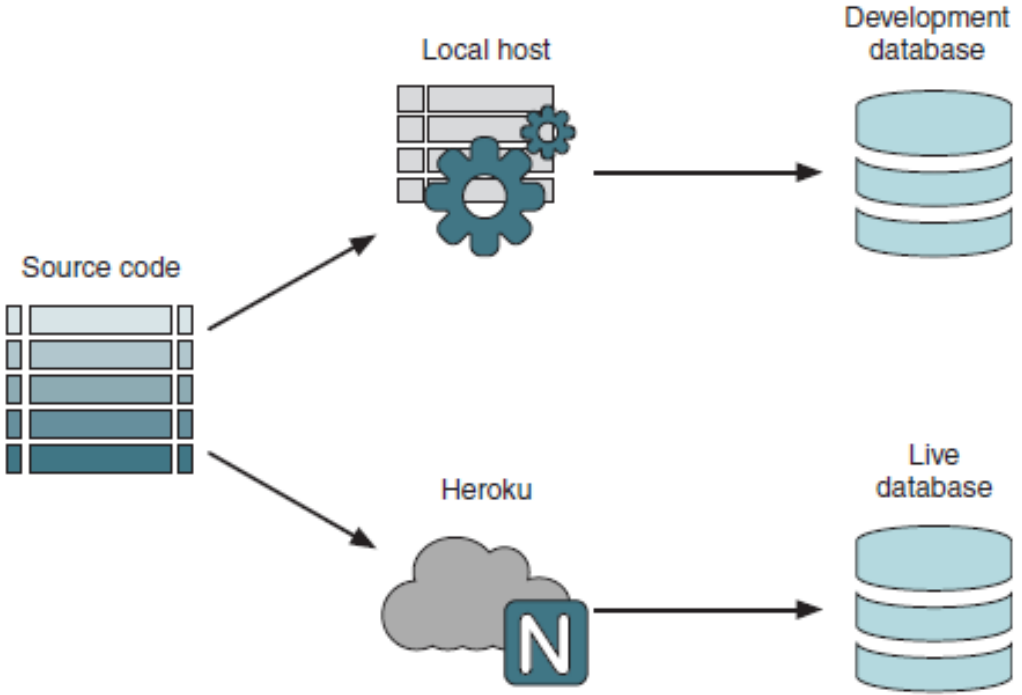


Figure 5.12 The source code runs in two locations, each of which needs to connect to a different database.

Setting up mLab and getting the database URI

- The first goal is getting an externally accessible database URI so that you can push data to it and add it to the application.

Steps

Step 1: Add mLab Add-on to Heroku

Make sure you are inside your application root folder. Run:

```
$ heroku addons:create mongolab
```

This creates a live MongoDB database (free tier).

Open the database web interface:

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

Setting up mLab and getting the database URI

Step 2: Get the Database URI

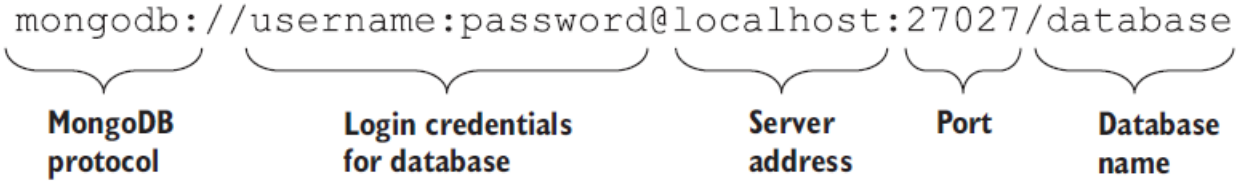
Retrieve the full connection string:

```
$ heroku config:get MONGODB_URI
```

Sample output:

```
mongodb://heroku_user:password@ds159330.mlab.com:59330/
heroku_db
```

BREAKING DOWN THE URI INTO ITS COMPONENTS



Pushing up the data

Step 3: Push Existing Data to Live Database

1.Go to a directory where you want the dump to be stored.

2.Dump data from your **local** database:

```
$ mongodump -h localhost:27017 -d Loc8r
```

3.Restore dump to **live** database:

```
$ mongorestore -h ds159330.mlab.com:59330 -d heroku_db \
-u heroku_user -p password dump/
```

Test the Live Database

Step 4: Test the Live Database

Connect via Mongo shell:

```
$ mongo ds159330.mlab.com:59330/heroku_db -u heroku_user
-p password
```

Then run:

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

Making the application use the right database

Step 5: Make App Use Correct Database

Modify `db.js` in `app_server/models`:

```
let dbURI = 'mongodb://localhost/Loc8r';

if (process.env.NODE_ENV === 'production') {
  dbURI = process.env.MONGODB_URI;
}

mongoose.connect(dbURI, { useNewUrlParser: true });
```

Step 6: Set Environment Variables in Heroku

Force Heroku into production mode:

```
$ heroku config:set NODE_ENV=production
```

Verify:

```
$ heroku config:get NODE_ENV
```

If you set up manually, also configure:

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

Step 7: Test Before Launch

Run locally in production mode:

```
$ NODE_ENV=production
MONGODB_URI=mongodb://<user>:<pass>@<host>:<port>/<db>
nodemon
```

Console output should confirm:

```
Mongoose connected to mongodb://<...>
```

Step 8: Push to Heroku

Commit and push code:

```
$ git add --all
$ git commit -m "Configured live database"
$ git push heroku master
```

Check logs for confirmation:

```
$ heroku logs
```

Look for:

```
Mongoose connected to
mongodb://heroku_user:password@ds159330.mlab.com:59330/h
eroku_db
```

Writing a REST API:

Exposing the MongoDB database to the application

UNIT-III

Building a data model with MongoDB and Mongoose: Connecting the Express application to MongoDB by using Mongoose, Benefits of modeling the data, defining simple mongoose schemas, using the MongoDB shell to create a MongoDB database and add data, getting database live.

Writing a REST API: Exposing the MongoDB database to the application: The rules of a REST API, setting up the API in Express, GET methods: Reading data from MongoDB, POST methods: Adding data to MongoDB, PUT methods: Updating data in MongoDB, DELETE method: Deleting data from MongoDB.

REST API

- The different request methods (GET, POST, PUT, and DELETE) that should be used for different actions, and how an API should respond with data and an appropriate HTTP status code.

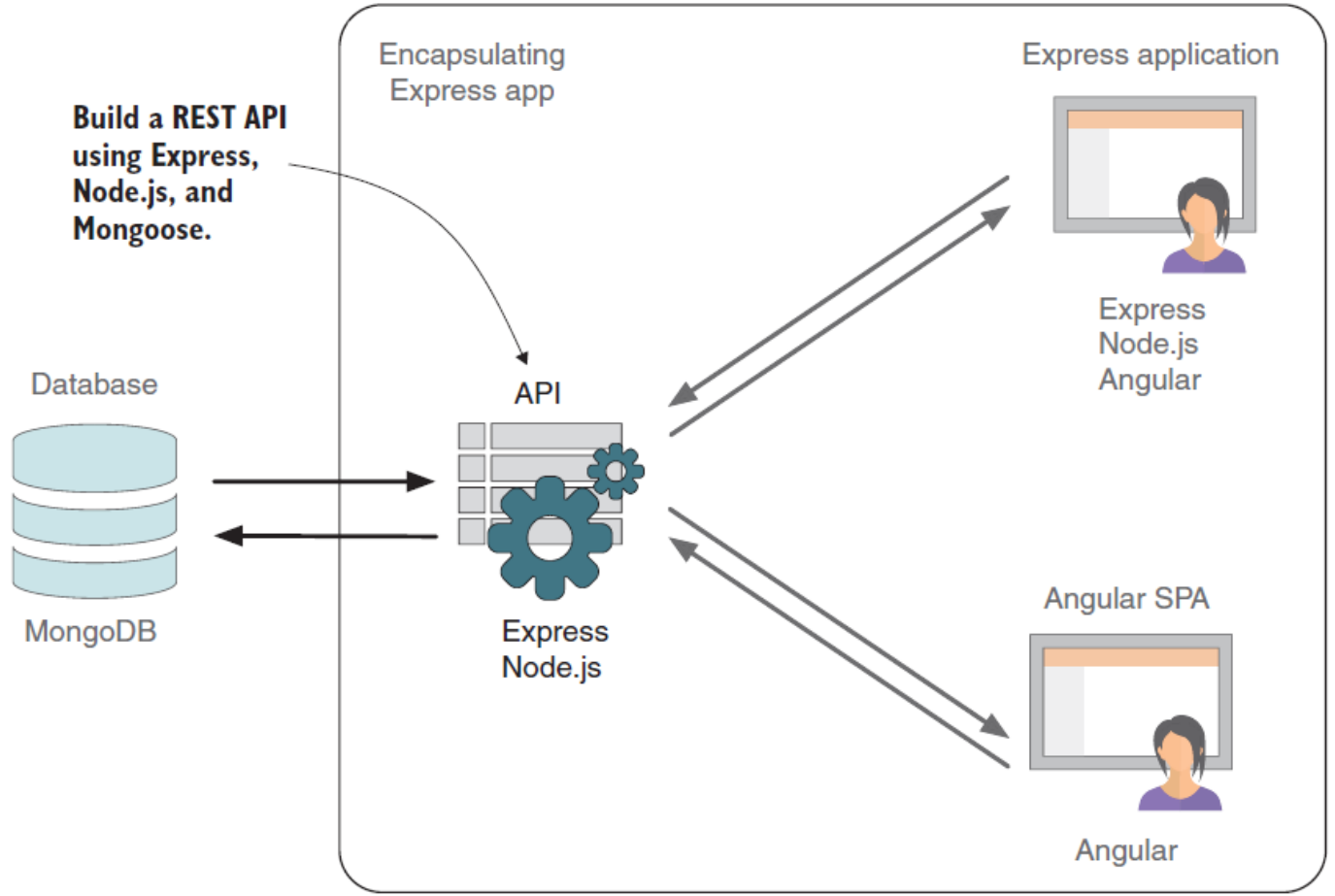


Figure 6.1 This chapter focuses on building the API that interacts with the database, exposing an interface for the applications to talk to.

The rules of a REST API

- ***REST*** stands for ***REpresentational State Transfer***, which is an architectural style rather than a strict protocol.
- **REST** is **stateless**; it has no idea of any current user state or history.
- ***API*** is an abbreviation for ***application program interface***, which enables applications to talk to one another.
- In the case of the MEAN stack, the **REST API** is used to create a stateless interface to your database, enabling a way for other applications to work with the data.

The rules of a REST API

- In basic terms, a REST API takes an incoming HTTP request, does some processing, and always sends back an HTTP response

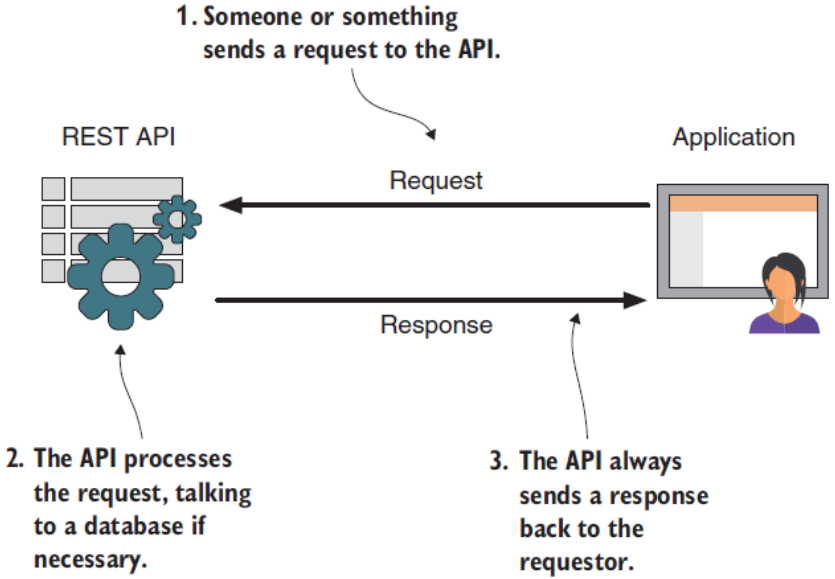


Figure 6.2 A REST API takes incoming HTTP requests, does some processing, and returns HTTP responses.

The rules of a REST API

1. Request URLs

Standard CRUD operations

- Create a new item
- Read a list of several items
- Read a specific item
- Update a specific item
- Delete a specific item

Table 6.1 URL paths and parameters for an API to the Locations collection

Action	URL path	Example
Create new location	`/locations`	http://loc8r.com/api/locations
Read list of locations	`/locations`	http://loc8r.com/api/locations
Read a specific location	`/locations/:locationid`	http://loc8r.com/api/locations/123
Update a specific location	`/locations/:locationid`	http://loc8r.com/api/locations/123
Delete a specific location	`/locations/:locationid`	http://loc8r.com/api/locations/123

2. Request methods

- HTTP requests can have different methods that essentially tell the server what type of action to take.
- The most common type of request is a GET request—the method used when you enter a URL in the address bar of your browser.
- Another common method is POST, often used for submitting form data.

Table 6.2 Four request methods used in a REST API

Request method	Use	Response
POST	Create new data in the database	New data object as seen in the database
GET	Read data from the database	Data object answering the request
PUT	Update a document in the database	Updated data object as seen in the database
DELETE	Delete an object from the database	Null

2. Request methods

Table 6.3 Request methods that link URLs to the desired actions, enabling the API to use the same URL for different actions

Action	Method	URL path	Example
Create new location	POST	/locations	http://loc8r.com/api/locations
Read list of locations	GET	/locations	http://loc8r.com/api/locations
Read a specific location	GET	/locations/:locationid	http://loc8r.com/api/locations/123
Update a specific location	PUT	/locations/:locationid	http://loc8r.com/api/locations/123
Delete a specific location	DELETE	/locations/:locationid	http://loc8r.com/api/locations/123

Table 6.4 URL specifications for interacting with subdocuments; each base URL path must contain the ID of the parent document

Action	Method	URL path	Example
Create new review	POST	/locations/:locationid/reviews	http://loc8r.com/api/locations/123/reviews
Read a specific review	GET	/locations/:locationid/reviews/:reviewid	http://loc8r.com/api/locations/123/reviews/abc
Update a specific review	PUT	/locations/:locationid/reviews/:reviewid	http://loc8r.com/api/locations/123/reviews/abc
Delete a specific review	DELETE	/locations/:locationid/reviews/:reviewid	http://loc8r.com/api/locations/123/reviews/abc

3. Responses and status codes

- For a successful REST API, standardising the responses is as important as standardising the request format. There are two key components to a response:
  - The returned data
  - The HTTP status code
- Combining the returned data with the appropriate status code should give the requester all the information required to continue.

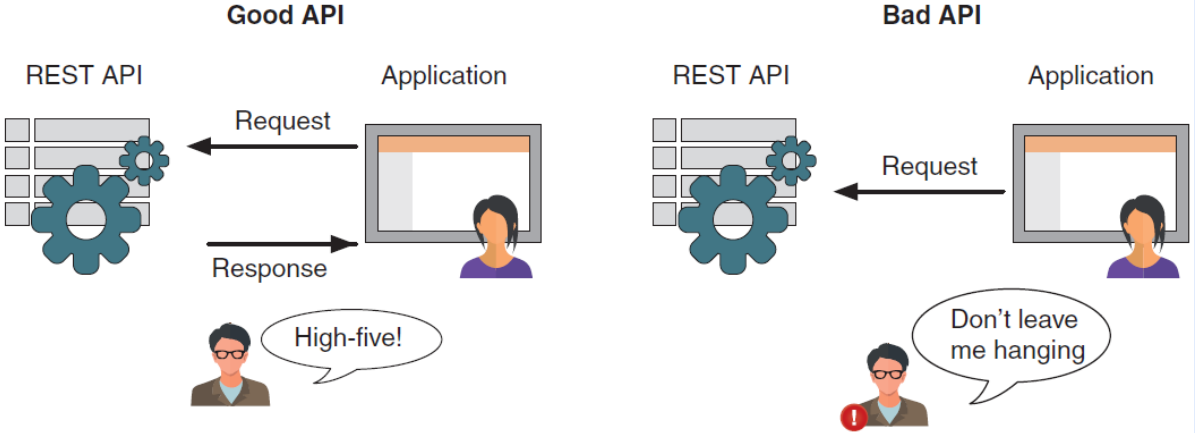


Figure 6.3 A good API always returns a response and shouldn't leave you hanging.

3. Responses and status codes

1. Returning data from an API

- Typical formats for a REST API are XML and/or JSON. You'll use JSON for your API, because it's the natural fit for the MEAN stack.
- MongoDB outputs JSON, which Node and Angular can both natively understand.
- JSON is, after all, the JavaScript way of transporting data.
- JSON is also more compact than XML, so it can help speed the response times and efficiency of an API by reducing the bandwidth required.

Your API will return one of three things for each request:

- A JSON object containing data answering the request query
- A JSON object containing error data
- A null response

3. Responses and status codes

2. USING HTTP STATUS CODES

Table 6.5 Most popular HTTP status codes and how they might be used to send responses to an API request

Status code	Name	Use case
200	OK	A successful GET or PUT request
201	Created	A successful POST request
204	No content	A successful DELETE request
400	Bad request	An unsuccessful GET, POST, or PUT request due to invalid content
401	Unauthorized	Requesting a restricted URL with incorrect credentials
403	Forbidden	Making a request that isn't allowed
404	Not found	Unsuccessful request due to an incorrect parameter in the URL
405	Method not allowed	Request method not allowed for the given URL
409	Conflict	Unsuccessful POST request when another object with the same data already exists
500	Internal server error	Problem with your server or the database server

2. Setting up the API in Express

- Create a structure directory called *app_api* similar to *app_server* as sibling
- Create subfolder called *routes*
- Create a file under routes as *index.js*

1 Creating the routes

INCLUDING THE ROUTES IN THE APPLICATION

The first step is telling your application that you're adding more routes to look out for and when it should use them. You can duplicate a line in app.js to `require` the server application routes, and set the path to the API routes as follows:

```
const indexRouter = require('./app_server/routes/index');
const apiRouter = require('./app_api/routes/index');
```

INCLUDING THE ROUTES IN THE APPLICATION

Modify in App.js file

const indexRouter = require('./app_server/routes/index');

const apiRouter = require('./app_api/routes/index');

app.use('/', indexRouter);

app.use('/api', apiRouter);

SPECIFYING THE REQUEST METHODS IN THE ROUTES

router.**get**('/location', ctrlLocations.locationInfo);
router.post('/locations', ctrlLocations.locationsCreate);

SPECIFYING REQUIRED URL PARAMETERS

- Suppose that you're trying to access a review with the ID abc that belongs to a location with the ID 123. You'd have a URL path like this:

  */api/locations/**123**/reviews/**abc***

- Swapping out the IDs for the parameter names (with a colon prefix) gives you a path like this:

  */api/locations/**:locationid**/reviews/**:reviewid***

DEFINING THE LOC8R API ROUTES

Listing 6.1 Routes defined in app_api/routes/index.js

```
const express = require('express');
const router = express.Router();
const ctrlLocations = require('../controllers/locations');
const ctrlReviews = require('../controllers/reviews');

// locations
router
  .route('/locations')
  .get(ctrlLocations.locationsListByDistance)
  .post(ctrlLocations.locationsCreate);

router
  .route('/locations/:locationid')
  .get(ctrlLocations.locationsReadOne)
  .put(ctrlLocations.locationsUpdateOne)
  .delete(ctrlLocations.locationsDeleteOne);

// reviews
router
  .route('/locations/:locationid/reviews')
  .post(ctrlReviews.reviewsCreate);

router
  .route('/locations/:locationid/reviews/:reviewid')
  .get(ctrlReviews.reviewsReadOne)
  .put(ctrlReviews.reviewsUpdateOne)
  .delete(ctrlReviews.reviewsDeleteOne);

module.exports = router;
```

Includes controller files. (You'll create these next.)

Defines routes for locations

Defines routes for reviews

Exports routes

2. Setting up the API in Express

2. Creating the controller placeholders

- The first step, of course, is creating the controller files.
- You know where these files should be and what they should be called because you've already declared them in the ***app_api/routes*** folder.
- You need two new files called ***locations.js*** and ***reviews.js*** in the ***app_api/controllers*** folder.
- You can create a placeholder for each of the controller functions as an empty function, as in the following code snippet:

```
const locationsCreate = (req, res) => { };
```

2. Setting up the API in Express

2. Creating the controller placeholders

- Remember to put each controller in the correct file, depending on whether it's for a location or a review, and export them at the bottom of the files, as in this example:

```
module.exports = {
    locationsListByDistance,
    locationsCreate,
    locationsReadOne,
    locationsUpdateOne,
    locationsDeleteOne
};
```

2. Setting up the API in Express

3. Returning JSON from an Express request

When building the Express application, you rendered a view template to send HTML to the browser, but with an API, you instead want to send a status code and some JSON data. Express makes this task easy with the following lines:

```
res                    <-- Uses the Express response object
  .status(status)      <-- Sends response status code, such as 200
  .json(content);      <-- Sends response data, such as {"status" : "success"}
```

You can use these two commands in the placeholder functions to test the success, as shown in the following code snippet:

```
const locationsCreate = (req, res) => {
  res
    .status(200)
    .json({"status" : "success"});
};
```

As you build up your API, you'll use this method a lot to send different status codes and data as the response.

2. Setting up the API in Express

4. Including the model

- As api deals with model not the main express application, **models** folder has been moved to **app_api** from **app_server**.
- The same is to be updated in **app.js** file

*require('./**app_server**/models/db');*
*require('./**app_api**/models/db');*

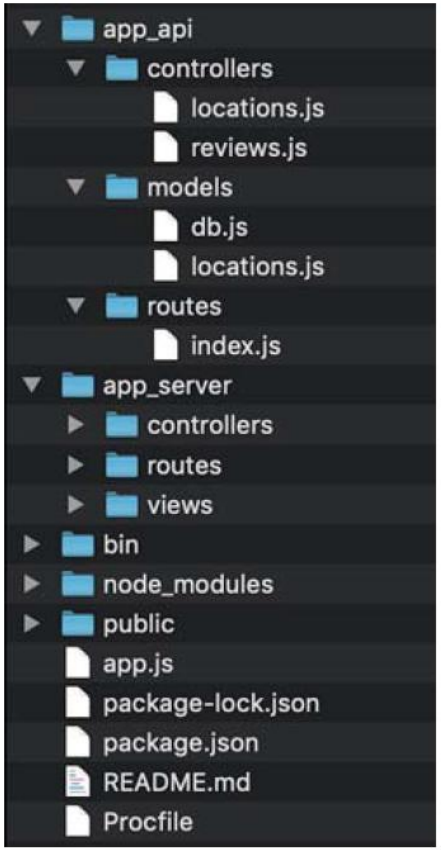


Figure 6.4 Folder structure of the application at this point. app_api has models, controllers, and routes, and app_server has views, controllers, and routes.

2. Setting up the API in Express

5. Testing the API

- You can test the GET routes in your browser quickly by heading to the appropriate URL, such as ***http://localhost:3000/api/locations/1234***. You should see the success response being delivered to the browser, as shown in figure 6.5.

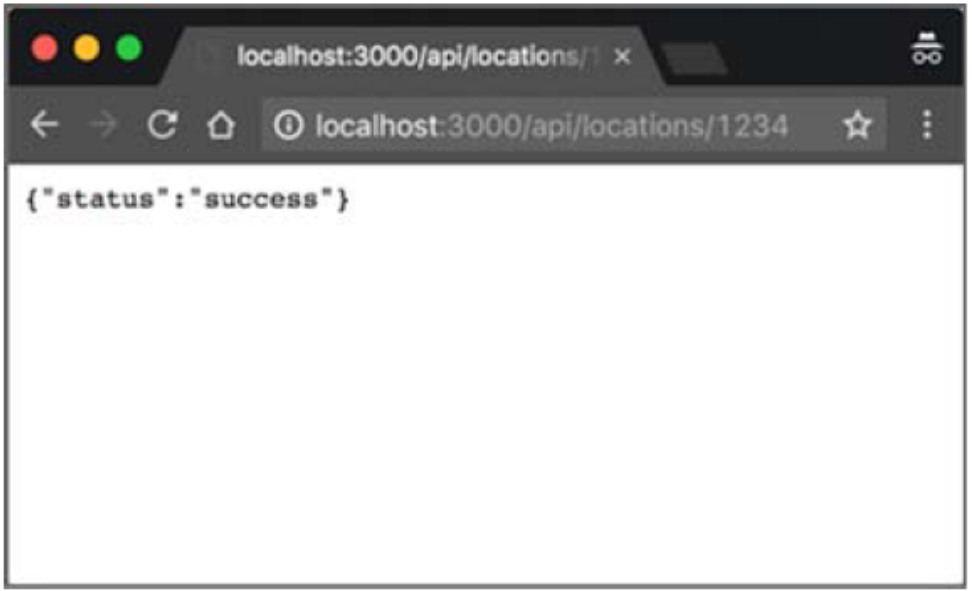


Figure 6.5 Testing a GET request of the API in the browser

2. Setting up the API in Express

5. Testing the API

Screenshot of Postman making a **PUT** request to the same URL

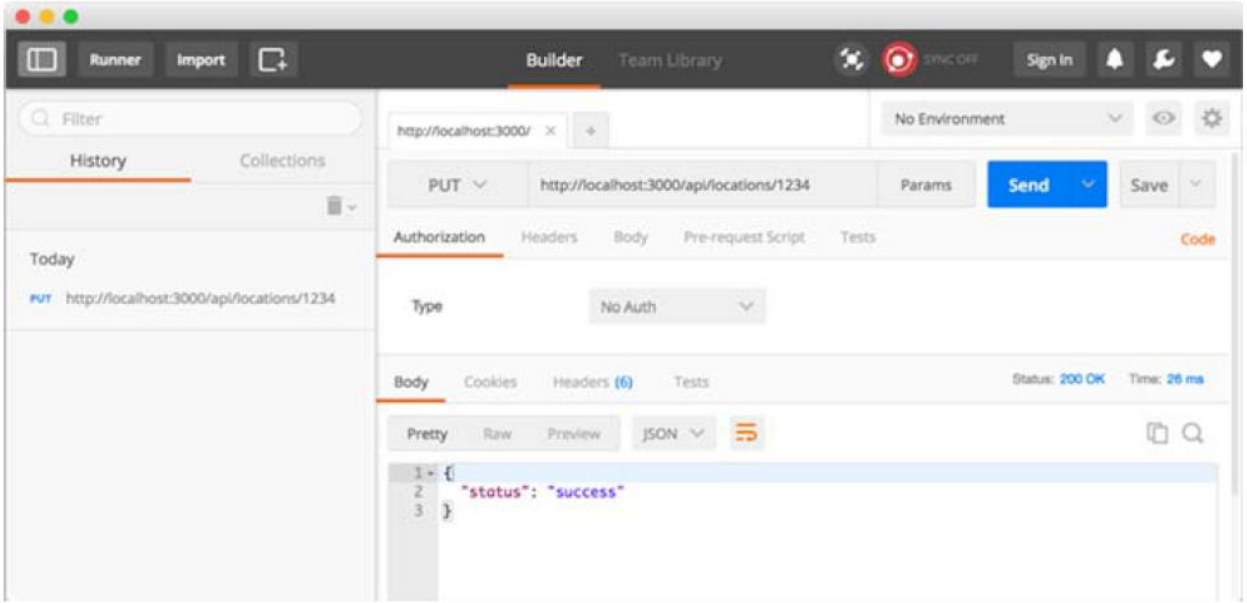


Figure 6.6 Using the Postman REST Client to test a `PUT` request to the API

3. GET methods: Reading data from MongoDB

`GET` methods are all about querying the database and returning some data.

Table 6.6 Three `GET` requests of the Loc8r API

Action	Method	URL path	Example
Read a list of locations	`GET`	`/locations`	http://loc8r.com/api/locations
Read a specific location	`GET`	`/locations/:locationid`	http://loc8r.com/api/locations/123
Read a specific review	`GET`	`/locations/:locationid/reviews/:reviewid`	http://loc8r.com/api/locations/123/reviews/abc

3. GET methods: Reading data from MongoDB

1. Finding a single document in MongoDB using Mongoose

Mongoose query methods

Mongoose models have several methods available to help with querying the database. Here are some of the key ones:

- `find`—General search based on a supplied query object
- `findById`—Looks for a specific ID
- `findOne`—Gets the first document to match the supplied query
- `geoNear`—Finds places geographically close to the provided latitude and longitude
- `geoSearch`—Adds query functionality to a `geoNear` operation

3. GET methods: Reading data from MongoDB

1. Finding a single document in MongoDB using Mongoose

Applying the findbyid method to the model

The **findById()** method is relatively straightforward, accepting a single parameter: the ID to look for.

```
Loc.findById(locationid)
```

Running the query with the exec method

- The **exec** method executes the query and passes a callback function that will run when the operation is complete.
- The callback function should accept two parameters: an error object and the instance of the found document.

The methods can be chained as follows:

```
Loc
  .findById(locationid)
  .exec((err, location) => {
    console.log("findById complete");
  });
```

Applies the findById method to the Location model, using Loc

Executes the query

Logs the message when complete

3. GET methods: Reading data from MongoDB

1. Finding a single document in MongoDB using Mongoose

Using the findById method in a controller

locations.js file in app_api/controllers

```
const locationsReadOne = (req, res) => {
  Loc
    .findById(req.params.locationid)
    .exec((err, location) => {
      res
        .status(200)
        .json(location);
    });
};
```

Gets a locationid from the URL parameters, and gives it to the findById method

Defines callback to accept possible parameters

Sends the document found as a JSON response with an HTTP status of 200

3. GET methods: Reading data from MongoDB

1. Finding a single document in MongoDB using Mongoose

Using the findById method in a controller

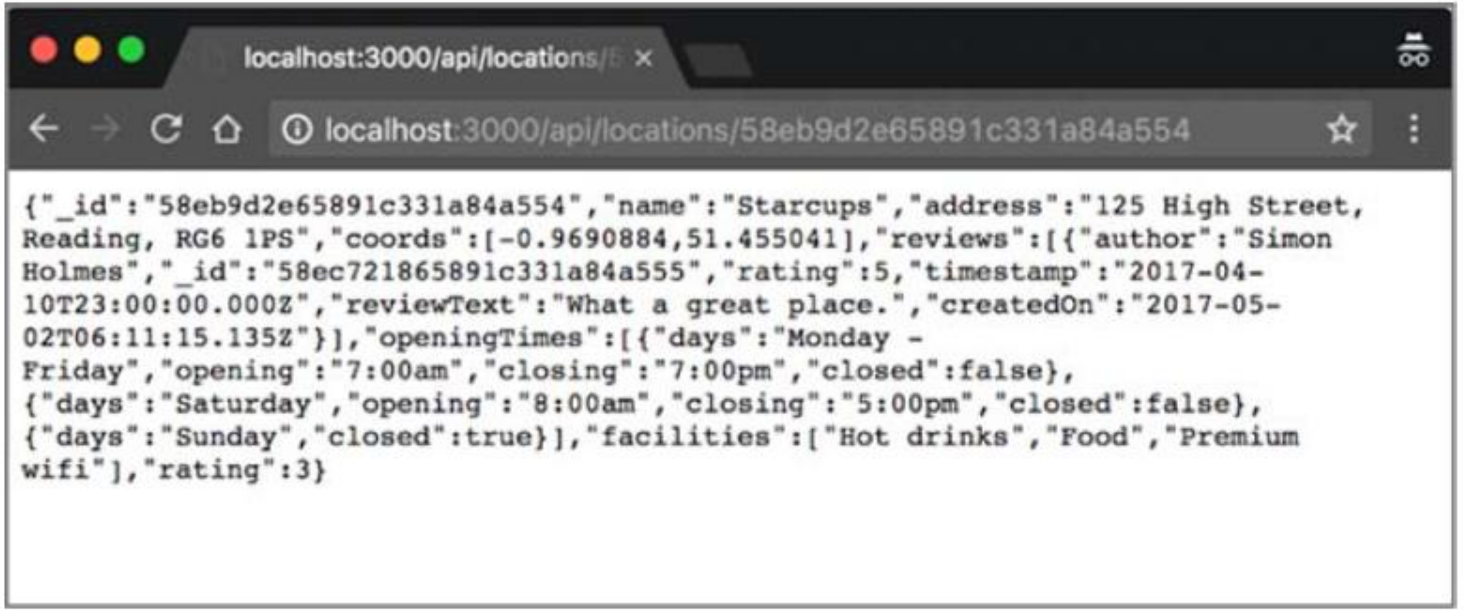


Figure 6.7 A basic controller for finding a single location by ID returns a JSON object to the browser if the ID is found.

3. GET methods: Reading data from MongoDB

1. Finding a single document in MongoDB using Mongoose

CATCHING ERRORS

With your basic controller, you need to trap three errors:

- The request parameters don't include `locationid`.
- The `findById()` method doesn't return a location.
- The `findById()` method returns an error.

The status code for an unsuccessful `GET` request is 404.

3. GET methods: Reading data from MongoDB

1. Finding a single document in MongoDB using Mongoose

CATCHING ERRORS

Listing 6.2 locationsReadOne controller

```
const locationsReadOne = (req, res) => {
    Loc
      .findById(req.params.locationid)
      .exec((err, location) => {
        if (!location) {
          return res
            .status(404)
            .json({
              "message": "location not found"
            });
        } else if (err) {
          return res
            .status(404)
            .json(err);
        }
        res
          .status(200)
          .json(location);
      });
};
```

1. Error trap 1: If Mongoose doesn't return a location, sends a 404 message and exits the function scope, using a return statement
2. Error trap 2: If Mongoose returns an error, sends it as a 404 response and exits the controller, using a return statement
3. If Mongoose doesn't error, continues as before, and sends a location object in a 200 response

3. GET methods: Reading data from MongoDB

1. Finding a single document in MongoDB using Mongoose

CATCHING ERRORS

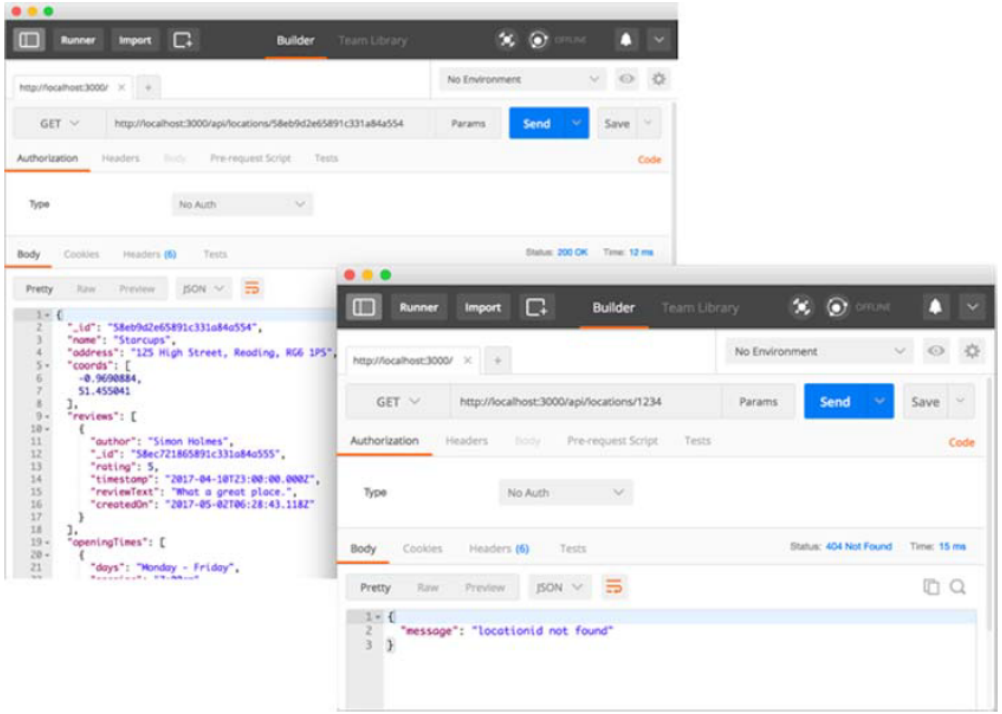


Figure 6.8 Testing successful (left) and failed (right) API responses using Postman

3. GET methods: Reading data from MongoDB

2. Finding a single subdocument based on IDs

The modifications are:

- Accept and use an additional **reviewid** URL parameter.
- Select only the name and reviews from the document rather than have MongoDB return the entire document.
- Look for a review with a matching ID.
- Return the appropriate JSON response.

To do these things, you can use a couple of new Mongoose methods.

LIMITING THE PATHS RETURNED FROM MONGODB

Limiting the data being passed around is also better for bandwidth consumption and speed.

```
Loc
   .findById(req.params.locationid)
   .select('name reviews')
   .exec();
```

3. GET methods: Reading data from MongoDB

2. Finding a single subdocument based on IDs

USING MONGOOSE TO FIND A SPECIFIC SUBDOCUMENT

Mongoose also offers a helper method for finding a subdocument by ID. Given an array of subdocuments, Mongoose has an id method that accepts the ID you want to find. The id method returns the single matching subdocument, and it can be used as follows:

```
Loc
  .findById(req.params.locationid)
  .select('name reviews')
  .exec((err, location) => {
    const review = location.reviews.id(req.params.reviewid);
  }
);
```

Passes reviewid from the parameters into the id method

3. GET methods: Reading data from MongoDB

2. Finding a single subdocument based on IDs

Adding some error trapping and putting it all together

Listing 6.3 Controller for finding a single review

```
const reviewsReadOne = (req, res) => {
    Loc
      .findById(req.params.locationid)
      .select('name reviews')
      .exec((err, location) => {
        if (!location) {
          return res
            .status(404)
            .json({
              "message": "location not found"
            });
        } else if (err) {
          return res
            .status(400)
            .json(err);
        }
        if (location.reviews && location.reviews.length > 0) {
```

Adds the Mongoose select method to the model query, stating that you want to get the name of a location and its reviews

Checks that the returned location has reviews

3. GET methods: Reading data from MongoDB

2. Finding a single subdocument based on IDs

Adding some error trapping and putting it all together

Uses the Mongoose subdocument .id method as a helper for searching for a matching ID

```
const review = location.reviews.id(req.params.reviewid);
if (!review) {
  return res
    .status(400)
    .json({
      "message": "review not found"
    });
} else {
  response = {
    location : {
      name : location.name,
      id : req.params.locationid
    },
    review
  };
  return res
    .status(200)
    .json(response);
}
} else {
  return res
    .status(404)
    .json({
      "message": "No reviews found"
    });
}
}
);
};
```

If a review isn't found, returns an appropriate response

If a review is found, builds a response object returning the review and location name and ID

If no reviews are found, returns an appropriate error message

3. GET methods: Reading data from MongoDB

3. Finding multiple documents with geospatial queries

- The homepage of Loc8r should display a list of locations based on the user's current geographical location.
- MongoDB and Mongoose have some special geospatial aggregation methods to help find nearby places.
- Mongoose aggregate **$geoNear** to find a list of locations close to a specified point, up to a specified maximum distance
- **$geoNear** is an aggregation method that accepts multiple configuration options, of which of the following are required:
  - `near` as a `geoJSON` geographical point
  - A `distanceField` object option
  - A `maxDistance` object option
- The following code snippet shows the basic construct:

```
Loc.aggregate([{$geoNear: {near: {}, distanceField:
"distance", maxDistance: 100}}]);
```

4. POST methods: Adding data to MongoDB

- **POST** methods are all about creating documents or subdocuments in the database and then returning the saved data as confirmation.

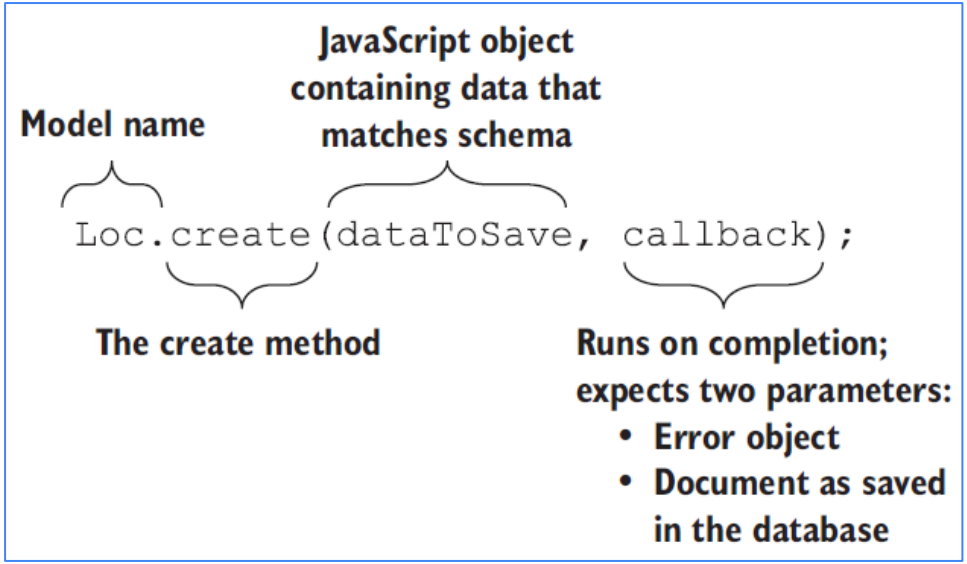
Table 6.7 Two POST requests of the Loc8r API

Action	Method	URL path	Example
Create new location	POST	/locations	http://api.loc8r.com/locations
Create new review	POST	/locations/:locationid/reviews	http://api.loc8r.com/locations/123/reviews

4. POST methods: Adding data to MongoDB

1. Creating new documents in MongoDB

- In the database for Loc8r, each location is a document.



The creation process has two main steps:

1. Use the posted form data to create a JavaScript object that matches the schema.
2. Send an appropriate response in the callback, depending on the success or failure of the create() operation.

4. POST methods: Adding data to MongoDB

1. Creating new documents in MongoDB

Listing 6.5 Complete controller for creating a new location

```
const locationsCreate = (req, res) => {
  Loc.create({                                  <-- Applies the create method to the model
    name: req.body.name,
    address: req.body.address,
    facilities:                                 <-- Creates an array of facilities by splitting a comma-separated list
     req.body.facilities.split(","),
    coords: {                                   <-- Parses coordinates from strings to numbers
     type: "Point",
     [
       parseFloat(req.body.lng),
       parseFloat(req.body.lat)
     ]
    }, {
      days: req.body.days2,
      opening: req.body.opening2,
      closing: req.body.closing2,
      closed: req.body.closed2,
    }]
  }, (err, location) => {                       <-- Supplies a callback function, containing appropriate responses for success and failure
    if (err) {
      res
        .status(400)
        .json(err);
    } else {
      res
        .status(201)
        .json(location);
    }
  });
};
```

4. POST methods: Adding data to MongoDB

2. Validating the data using Mongoose

- This controller has no validation code inside it, so what's to stop somebody from entering loads of empty or partial documents.

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

4. POST methods: Adding data to MongoDB

3. Creating new subdocuments in MongoDB

- In the context of Loc8r locations, reviews are subdocuments.
- Subdocuments are created and saved through their parent document.
- Put another way, to create and save a new subdocument, you have to
  1. Find the correct parent document.
  2. Add a new subdocument.
  3. Save the parent document.

4. POST methods: Adding data to MongoDB

3. Creating new subdocuments in MongoDB

Listing 6.6 Controller for creating a review

```
const reviewsCreate = (req, res) => {
  const locationId = req.params.locationid;
  if (locationId) {
    Loc
      .findById(locationId)
      .select('reviews')
      .exec((err, location) => {
        if (err) {
          res
            .status(400)
            .json(err);
        } else {
          doAddReview(req, res, location);
        }
      });
  } else {
    res
      .status(404)
      .json({"message": "Location not found"});
  }
};
```

Successful find operation will call a new function to add a review, passing request, response, and location object

4. POST methods: Adding data to MongoDB

3. Creating new subdocuments in MongoDB

Listing 6.7 Adding and saving a subdocument

```
const doAddReview = (req, res, location) => {
  if (!location) {
    res
      .status(404)
      .json({"message": "Location not found"});
  } else {
    const {author, rating, reviewText} = req.body;
    location.reviews.push({
      author,
      rating,
      reviewText
    });
    location.save((err, location) => {
      if (err) {
        res
          .status(400)
          .json(err);
      } else {
        updateAverageRating(location._id);
        const thisReview = location.reviews.slice(-1).pop();
        res
          .status(201)
          .json(thisReview);
      }
    });
  }
};
```

When provided a parent document . . .

. . . pushes new data into a subdocument array . . .

. . . before saving it.

On successful save operation, calls a function to update the average rating

Retrieves the last review added to the array, and returns it as a JSON confirmation response

4. POST methods: Adding data to MongoDB

3. Creating new subdocuments in MongoDB

Listing 6.8 Calculating and updating the average rating

Uses the location supplied data

Uses the JavaScript array reduce method to sum up the ratings of the subdocuments

Calculates the average rating value and updates the rating value of the parent document

Saves the parent document

Finds the location based on the provided locationid data

```
const doSetAverageRating = (location) => {
  if (location.reviews && location.reviews.length > 0) {
    const count = location.reviews.length;
    const total = location.reviews.reduce((acc, {rating}) => {
      return acc + rating;
    }, 0);

    location.rating = parseInt(total / count, 10);
    location.save(err => {
      if (err) {
        console.log(err);
      } else {
        console.log(`Average rating updated to ${location.rating}`);
      }
    });
  }
};

const updateAverageRating = (locationId) => {
  Loc.findById(locationId)
    .select('rating reviews')
    .exec((err, location) => {
      if (!err) {
        doSetAverageRating(location);
      }
    });
};
```

5. PUT methods: Updating data in MongoDB

- **PUT** methods are all about updating existing documents or subdocuments in the database and returning the saved data as confirmation

Table 6.8 Two PUT requests of the Loc8r API for updating locations and reviews

Action	Method	URL path	Example
Update a specific location	PUT	/locations/:locationid	http://loc8r.com/api/locations/123
Update a specific review	PUT	/locations/:locationid/reviews/:reviewid	http://loc8r.com/api/locations/123/reviews/abc

5. PUT methods: Updating data in MongoDB

1. Using Mongoose to update a document in MongoDB

In Loc8r, you may want to update a location to add new facilities, change the open times, or amend any of the other data. The approach to updating data in a document is probably starting to look familiar:

1. Find the relevant document.
2. Make some changes to the instance.
3. Save the document.
4. Send a JSON response.

This approach is made possible by the way an instance of a Mongoose model maps directly to a document in MongoDB. When your query finds the document, you get a model instance. If you make changes to this instance and then save it, Mongoose updates the original document in the database with your changes.

5. PUT methods: Updating data in MongoDB

2. Using the Mongoose save method

A cut-down skeleton of this approach is shown in the following code snippet:

```
Loc
  .findById(req.params.locationid)
  .exec((err, location) => {
    location.name = req.body.name;
    location.save((err, loc) => {
      if (err) {
        res
          .status(404)
          .json(err);
      } else {
        res
          .status(200)
          .json(loc);
      }
    });
  }
);
};
```

Finds the document to update

Makes a change to the model instance, changing a value of one path

Saves the document with Mongoose's save method

Returns a success or failure response

5. PUT methods: Updating data in MongoDB

3. Updating an existing subdocument in MongoDB

Updating a subdocument is exactly the same as updating a document, with one exception: after finding the document, you have to find the correct subdocument to make your changes. Then the save method is applied to the document, not the subdocument.

So the steps for updating an existing subdocument are

1. Find the relevant document.
2. Find the relevant subdocument.
3. Make some changes in the subdocument.
4. Save the document.
5. Send a JSON response.

5. PUT methods: Updating data in MongoDB

3. Updating an existing subdocument in MongoDB

Listing 6.10 Updating a subdocument in MongoDB

```
const reviewsUpdateOne = (req, res) => {
  if (!req.params.locationid || !req.params.reviewid) {
    return res
      .status(404)
      .json({
        "message": "Not found, locationid and reviewid are both required"
      });
  }

Loc
  .findById(req.params.locationid)        <-- Finds the parent document
  .select('reviews')
  .exec((err, location) => {
    if (!location) {
      return res
        .status(404)
        .json({
          "message": "Location not found"
        });
    } else if (err) {
      return res
        .status(400)
        .json(err);
    }
```

5. PUT methods: Updating data in MongoDB

3. Updating an existing subdocument in MongoDB

```
if (location.reviews && location.reviews.length > 0) {
  const thisReview = location.reviews.id(req.params.reviewid);
  if (!thisReview) {
    res
      .status(404)
      .json({
        "message": "Review not found"
      });
  } else {
    thisReview.author = req.body.author;
    thisReview.rating = req.body.rating;
    thisReview.reviewText = req.body.reviewText;
    location.save((err, location) => {
      if (err) {
        res
          .status(404)
          .json(err);
      } else {
        updateAverageRating(location._id);
        res
          .status(200)
          .json(thisReview);
      }
    });
  }
} else {
  res
    .status(404)
    .json({
      "message": "No review to update"
    });
}
  }
  );
};
```

Finds the subdocument

Makes changes to the subdocument from the supplied form data

Saves the parent document

Returns a JSON response, sending the subdocument object on the basis of a successful save

6. DELETE method: Deleting data from MongoDB

Table 6.9 Two DELETE requests of the Loc8r API for deleting locations and reviews

Action	Method	URL path	Example
Delete a specific location	DELETE	/locations/:locationid	http://loc8r.com/api/locations/123
Delete a specific review	DELETE	/locations/:locationid/reviews/:reviewid	http://loc8r.com/api/locations/123/reviews/abc

6. DELETE method: Deleting data from MongoDB

1. Deleting documents in MongoDB

- Mongoose makes deleting a document in MongoDB extremely simple by giving you the method **`findByIdAndRemove()`.**

Listing 6.11 Deleting a document from MongoDB, given an ID

```
const locationsDeleteOne = (req, res) => {
  const {locationid} = req.params;
  if (locationid) {
    Loc
      .findByIdAndRemove(locationid)
      .exec((err, location) => {
          if (err) {
            return res
              .status(404)
              .json(err);
          }
          res
            .status(204)
            .json(null);
        }
    );
  } else {
    res
      .status(404)
      .json({

     "message": "No Location"
    });
  }
};
```

Calls findByIdAndRemove method, passing in locationid

Executes the method

Responds with failure or success

6. DELETE method: Deleting data from MongoDB

2. Deleting a subdocument from MongoDB

The process for deleting a subdocument is no different from the other work you've done with subdocuments; everything is managed through the parent document.

The steps for deleting a subdocument are

1. Find the parent document.
2. Find the relevant subdocument.
3. Remove the subdocument.
4. Save the parent document.
5. Confirm success or failure of operation.

6. DELETE method: Deleting data from MongoDB

2. Deleting a subdocument from MongoDB

Listing 6.12 Finding and deleting a subdocument from MongoDB

```
const reviewsDeleteOne = (req, res) => {
  const {locationid, reviewid} = req.params;
  if (!locationid || !reviewid) {
    return res
      .status(404)
      .json({'message': 'Not found, locationid and reviewid are both
     required'});
  }

  Loc
    .findById(locationid)
    .select('reviews')
    .exec((err, location) => {
      if (!location) {
        return res
          .status(404)
          .json({'message': 'Location not found'});
      } else if (err) {
        return res
          .status(400)
          .json(err);
      }
```

Finds the relevant parent document

6. DELETE method: Deleting data from MongoDB

2. Deleting a subdocument from MongoDB

```
  if (location.reviews && location.reviews.length > 0) {
    if (!location.reviews.id(reviewid)) {
      return res
        .status(404)
        .json({'message': 'Review not found'});
    } else {
      location.reviews.id(reviewid).remove();
      location.save(err => {
        if (err) {
          return res
            .status(404)
            .json(err);
        } else {
          updateAverageRating(location._id);
          res
            .status(204)
            .json(null);
        }
      });
    }
  } else {
    res
      .status(404)
      .json({'message': 'No Review to delete'});
  }
 });
};
```

Finds and deletes the relevant subdocument in one step

Saves the parent document

Returns the appropriate success or failure response

Thank you