



# Getting MEAN with Mongo, Express, Angular, and Node

SECOND EDITION

Simon Holmes





**MEAP Edition**  
**Manning Early Access Program**  
**Getting MEAN with Mongo, Express, Angular, and Node**  
**Second Edition**  
**Version 7**

Copyright 2017 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

**Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>**

# welcome

---

Thank you for purchasing the MEAP for *Getting MEAN Second Edition*. I'm excited to see the book reach this stage and look forward to its continued development and eventual release. This is an intermediate book, designed for anyone with web-development experience—particularly with some exposure to JavaScript—who wants to learn how to be a full-stack developer or see how the whole MEAN stack fits together.

I've strived to make the content both approachable and meaningful, and to explain not just *how* to do things with the MEAN stack but also *why* things are done the way they are. I feel it is important to know about each part of the MEAN stack before diving in to building an application.

To start, chapter 1 is an overview of what full stack development means, and the roles MongoDB, Express, Angular, and Node.js each play generally in forming the MEAN stack.

Chapter 2 takes a closer look at a few of the ways you can use the MEAN stack (there isn't just one "right" way to do it!). We'll explore a couple of scenarios and show how you can combine the components of the MEAN stack in different ways to meet the requirements of your project.

And in chapter 3, we'll get started building our example app, Loc8r. Loc8r will be a responsive web application, with an aim of listing places nearby that based on current location. Chapter 3 will see us setting up a base project using Node and Express, and also pushing it to a live environment for testing.

Looking ahead, the book will continue to cover building a responsive, data-driven web application using Node.js, Express, and MongoDB, with a cast of supporting technologies. Part 3 will complete the MEAN stack by adding an Angular single page application as the front-end to the application.

As you're reading, I hope you'll take advantage of the Author Online forum. I'll be reading your comments and responding, and your feedback is helpful in the development process.

—Simon Holmes

# *brief contents*

---

## **PART 1: SETTING THE BASELINE**

- 1 Introducing full-stack development*
- 2 Designing a MEAN stack architecture*

## **PART 2: BUILDING A NODE WEB APPLICATION**

- 3 Creating and setting up a MEAN project*
- 4 Building a static site with Node and Express*
- 5 Building a data model with MongoDB and Mongoose*
- 6 Writing a REST API: Exposing the MongoDB database to the application*
- 7 Consuming a REST API: Using an API from inside Express*

## **PART 3: ADDING A DYNAMIC FRONT END WITH ANGULAR**

- 8 Creating an Angular application with TypeScript*
- 9 Building a Single Page Application with Angular: Foundations*
- 10 Building a Single Page Application with Angular: The next level*

## **PART 4: MANAGING AUTHENTICATION AND USER SESSIONS**

- 11 Authenticating users, managing sessions and securing APIs*

## **APPENDICES:**

- A Installing the stack*
- B Installing and preparing the supporting cast*
- C Dealing with all of the views*
- D Reintroducing JavaScript*

# 1

## *Introducing full-stack development*

### This chapter covers

- The benefits of full-stack development
- An overview of the MEAN stack components
- What makes the MEAN stack so compelling
- A preview of the application we'll build throughout this book

If you're like me then you're probably impatient to dive into some code and get on with building something. But let's take a moment first to clarify what is meant by *full-stack development*, and look at the component parts of the stack to make sure you understand each.

When I talk about full-stack development, I'm really talking about developing all parts of a website or application. The full stack starts with the database and web server in the back end, contains application logic and control in the middle, and goes all the way through to the user interface at the front end.

The MEAN stack is a pure JavaScript stack comprised of four main technologies, with a cast of supporting technologies:

- **MongoDB** —the database
- **Express** —the web framework
- **Angular** —the front-end framework
- **Node.js** —the web server

MongoDB has been around since 2007, and is actively maintained by MongoDB Inc., previously known as 10gen.

Express was first released in 2009 by T. J. Holowaychuk and has since become the most popular framework for Node.js. It's open source, with more than 100 contributors, and is actively developed and supported.

Angular is open source and backed by Google. The first version of Angular – known as AngularJS or Angular 1 - has been around since 2010. Angular 2 – now known simply as Angular – was officially released in 2016 and is constantly being developed and extended. At the time of writing we are now on Angular 4; Angular 2+ is not backward-compatible with AngularJS. See the *Angular versions and release cycles* sidebar for a bit more information about the number and release cycles.

### **Angular versions and release cycles**

The change from Angular 1.x to Angular 2 was a big deal in the developer community. It was a long time coming, very different and not backwards compatible. But now Angular are releasing versions much more frequently, aiming for every six months. At the time of writing we're on Angular 4, with 5 slated to come soon.

The frequency of change is nothing to worry about though, the changes are nowhere near as big as the complete rewrite that happened between 1.x and 2.0. The changes are generally small incremental changes. There may be some breaking changes between 4 and 5, or 5 and 6 but these are normally small, specific items that are easy to pick up. It is very different from the change from Angular 1.x to 2.0.

Node.js was created in 2009, and its development and maintenance are sponsored by Joyent. Node.js uses Google's open source V8 JavaScript engine at its core.

## **1.1 Why learn the full stack?**

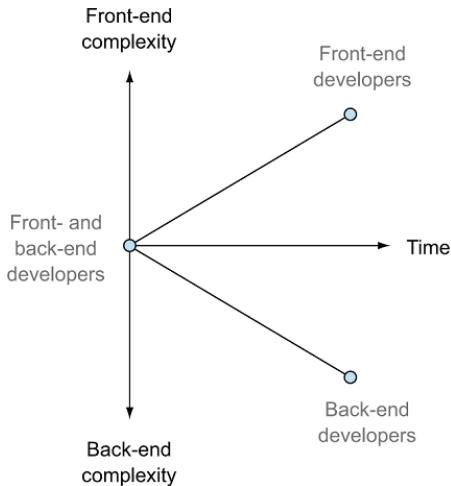
So, indeed, why learn the full stack? It sounds like an awful lot of work! Well yes, it *is* quite a lot of work, but it's also very rewarding as you get to create fully functioning data-driven websites and applications all by yourself. And with the MEAN stack it isn't as hard as you might think.

### **1.1.1 A very brief history of web development**

Back in the early days of the web, people didn't have high expectations of websites. Not much emphasis was given to presentation; it was much more about what was going on behind the scenes. Typically, if you knew something like Perl and could string together a bit of HTML then you were a web developer.

As use of the internet spread, businesses started to take more of an interest in how their online presence portrayed them. In combination with the increased browser support of Cascading Style Sheets (CSS) and JavaScript, this desire started to lead to more complicated front-end implementations. It was no longer a case of being able to string together HTML; you needed to spend time on CSS and JavaScript, making sure it looked right and worked as expected. And all of this needed to work in different browsers, which were much less compliant than they are today.

This is where the distinction between front-end developer and back-end developer came in. Figure 1.1 illustrates this separation over time.

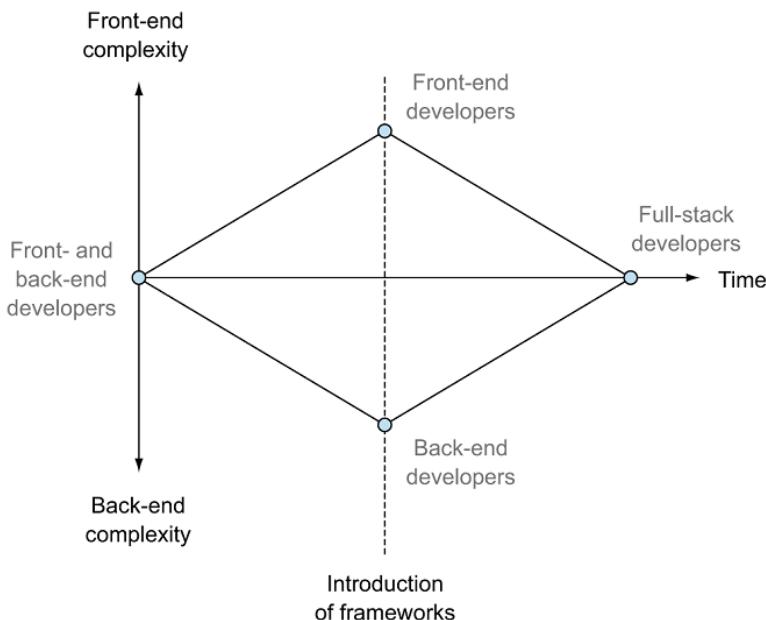


**Figure 1.1 Divergence of front-end and back-end developers over time**

While the back-end developers were focused on the mechanics behind the scenes, the front-end developers focused on building a good user experience. As time went on, higher expectations were made of both camps, encouraging this trend to continue. Developers often had to choose an expertise and focus on it.

#### **HELPING DEVELOPERS WITH LIBRARIES AND FRAMEWORKS**

During the 2000s libraries and frameworks started to become popular and prevalent for the most common languages, on both the front and back ends—think Dojo and jQuery for front-end JavaScript, and Symfony for PHP and Ruby on Rails. These frameworks were designed to make your life as a developer easier, lowering the barriers to entry. A good library or framework abstracts away some of the complexities of development, allowing you to code faster and requiring less in-depth expertise. This trend toward simplification has resulted in a resurgence of full-stack developers who build both the front end and the application logic behind it, as figure 1.2 shows.



**Figure 1.2 Impact of frameworks on the separated web development factions**

Figure 1.2 illustrates a trend rather than proclaiming a definitive “all web developers should be full-stack developers” maxim. There have been, of course, full-stack developers throughout the entire history of the web, and moving forward it’s most likely that some developers will choose to specialize on either front-end or back-end development. The intention is to show that through the use of frameworks and modern tools, you no longer have to choose one side or the other to be a good web developer.

A huge advantage of embracing the framework approach is that you can be incredibly productive, because you’ll have an all-encompassing vision of the application and how it ties together.

#### **MOVING THE APPLICATION CODE FORWARD IN THE STACK**

Continuing with the trend for frameworks, the last few years have seen an increasing tendency for moving the application logic away from the server and into the front end. Think of it as coding the back end in the front end. Some of the more popular JavaScript frameworks doing this are Angular, React, and Vue.

Tightly coupling the application code to the front end like this really starts to blur the lines between traditional front-end and back-end developers. One of the reasons that people like to use this approach is that it reduces the load on the servers, thus reducing cost. What you’re

doing in effect is crowd-sourcing the computational power required for the application by pushing that load into the users' browsers.

I'll discuss the pros and cons of this approach later in the book, and cover when it may or may not be appropriate to use one of these technologies.

### **1.1.2 The trend toward full-stack developers**

As discussed, the paths of front-end and back-end developers are merging, and it's entirely possible to be fully proficient in both disciplines. If you're a freelancer, consultant, or part of a small team, being multiskilled is extremely useful, increasing the value that you can provide for your clients. Being able to develop the full scope of a website or application gives you better overall control and can help the different parts work seamlessly together, because they haven't been built in isolation by separate teams.

If you work as part of a large team, chances are that you'll need to specialize in (or at least focus on) one area. But it's generally advisable to understand how your component fits with other components, giving you a greater appreciation of the requirements and goals of other teams and the overall project.

In the end, building on the full stack by yourself is very rewarding. Each part comes with its own challenges and problems to solve, keeping things interesting. The technology and tools available today enhance this experience and empower you to build great web applications relatively quickly and easily.

### **1.1.3 Benefits of full-stack development**

There are many benefits to learning full-stack development. For starters, there's the enjoyment of learning new things and playing with new technologies, of course. Then there's also the satisfaction of mastering something different and the thrill of being able to build and launch a full database-driven application all by yourself.

The benefits when working in a team include the following:

- You're more likely to have a better view of the bigger picture by understanding the different areas and how they fit together.
- You'll form an appreciation of what other parts of the team are doing and what they need to be successful.
- Team members can move around more freely.

The additional benefits when working by yourself include

- You can build applications end-to-end by yourself with no dependencies on other people.
- You have more skills, services, and capabilities to offer customers.

All in all, there's a lot to be said for full-stack development. Most of the accomplished developers I've met have been full-stack developers. Their overall understanding and ability to see the bigger picture is a tremendous bonus.

### 1.1.4 Why the MEAN stack specifically?

The MEAN stack pulls together some of the “best-of-breed” modern web technologies into a very powerful and flexible stack. One of the great things about the MEAN stack is that it not only uses JavaScript in the browser, it uses JavaScript throughout. Using the MEAN stack, you can code both the front end and back end in the same language.

Figure 1.3 demonstrates the principle technologies of the MEAN, and where each is commonly used.

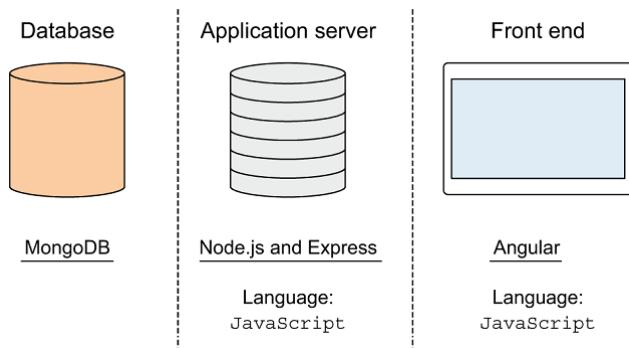


Figure 1.3 The four technologies in the MEAN stack, and what they do

The principle technology allowing full-stack JavaScript to happen is Node.js, bringing JavaScript to the back end.

## 1.2 Introducing Node.js: The web server/platform

Node.js is the N in MEAN. Being last doesn’t mean that it’s the least important: it’s the foundation of the stack!

In a nutshell, Node.js is a software platform that allows you to create your own web server and build web applications on top of it. Node.js isn’t itself a web server, nor is it a language. It contains a built-in HTTP server library, meaning that you don’t need to run a separate web server program such as Nginx, Apache or Internet Information Services (IIS). This ultimately gives you greater control over how your web server works, but also increases the complexity of getting it up and running, particularly in a live environment.

With PHP, for example, you can easily find a shared-server web host running Apache, send some files over FTP, and—all being well—your site is running. This works because the web host has already configured Apache for you and others to use. With Node.js this isn’t the case, because you configure the Node.js server when you create your application. Many of the traditional web hosts are behind the curve on Node.js support, but a number of new platform as a service (PaaS) hosts are springing up to address this need, including Heroku, Nodejitsu, and OpenShift. The approach to deploying live sites on these PaaS hosts is different from the

old FTP model, but is quite easy when you get the hang of it. We'll be deploying a site live to Heroku as we go through the book.

An alternative approach to hosting a Node.js application is to do it all yourself on a dedicated server onto which you can install anything you need. But production server administration is a whole other book! And although you could independently swap out any of the other components with an alternative technology, if you take Node.js out, everything that sits on top of it would change.

### **1.2.1 JavaScript: The single language through the stack**

One of the main reasons that Node.js is gaining broad popularity is that you code it in a language that most web developers are already familiar with: JavaScript. Until Node was released, if you wanted to be a full-stack developer you had to be proficient in at least two languages: JavaScript on the front end and something else like PHP or Ruby on the back end.

---

#### **Microsoft's foray into server-side JavaScript**

In the late 1990s, Microsoft released Active Server Pages (now known as Classic ASP). ASP could be written in either VBScript or JavaScript, but the JavaScript version didn't really take off. This is largely because, at the time, a lot of people were familiar with Visual Basic, which VBScript looks like. Many books and online resources were for VBScript, so it snowballed into becoming the "standard" language for Classic ASP.

---

With the release of Node.js you can leverage what you already know and put it to use on the server. One of the hardest parts of learning a new technology like this is learning the language, but if you already know some JavaScript then you're one step ahead!

There is, of course, a learning curve when taking on Node.js, even if you're an experienced front-end JavaScript developer. The challenges and obstacles in server-side programming are different from those in the front end, but you'll face those no matter what technology you use. In the front end, you might be concerned about making sure everything works in a variety of different browsers on different devices. On the server, you're more likely to be aware of the flow of the code to ensure that nothing gets held up and that you don't waste system resources.

### **1.2.2 Fast, efficient, and scalable**

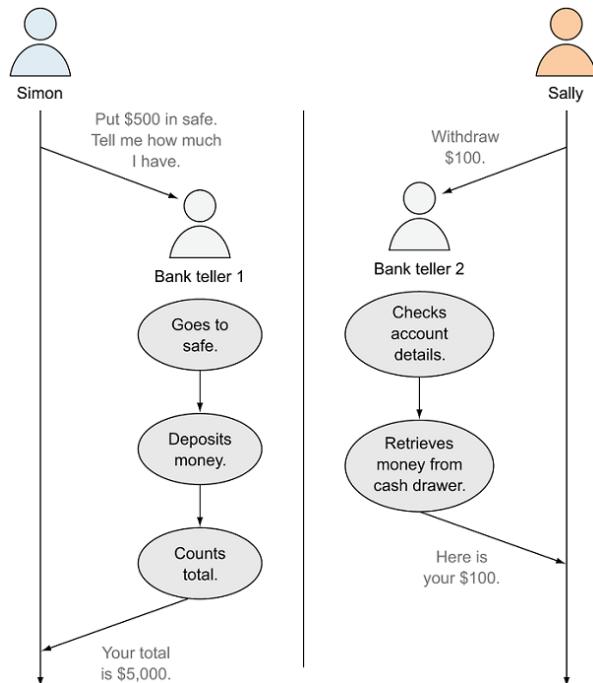
Another reason for the popularity of Node.js is, when coded correctly, it's extremely fast and makes very efficient use of system resources. This enables a Node.js application to serve more users on fewer server resources than most of the other mainstream server technologies. Business owners also like the idea of Node.js because it can reduce their running costs, even at a large scale.

How does it do this? Node.js is light on system resources because it's single-threaded, whereas traditional web servers are multithreaded. Let's look at what that means, starting with the traditional multithreaded approach.

## TRADITIONAL MULTITHREADED WEB SERVER

Most of the current mainstream web servers are multithreaded, including Apache and IIS. What this means is that every new visitor (or session) is given a separate “thread” and associated amount of **RAM**, often around 8 MB.

Thinking of a real-world analogy, imagine two people going into a bank wanting to do separate things. In a multithreaded model, they’d each go to a separate bank teller who would deal with their requests, as shown in figure 1.4.



**Figure 1.4 Example of a multithreaded approach: visitors use separate resources. Visitors and their dedicated resources have no awareness of or contact with other visitors and their resources.**

You can see in figure 1.3 that Simon goes to bank teller 1 and Sally goes to bank teller 2. Neither side is aware of or impacted by the other. Bank teller 1 deals with Simon throughout the entirety of the transaction and nobody else; the same goes for bank teller 2 and Sally.

This approach works perfectly well as long as you have enough tellers to service the customers. When the bank gets busy and the customers outnumber the tellers, that’s when the service starts to slow down and the customers have to wait to be seen. While banks don’t always worry about this too much, and seem happy to make you queue, the same isn’t true of websites. If a website is slow to respond you’re likely to leave and never come back.

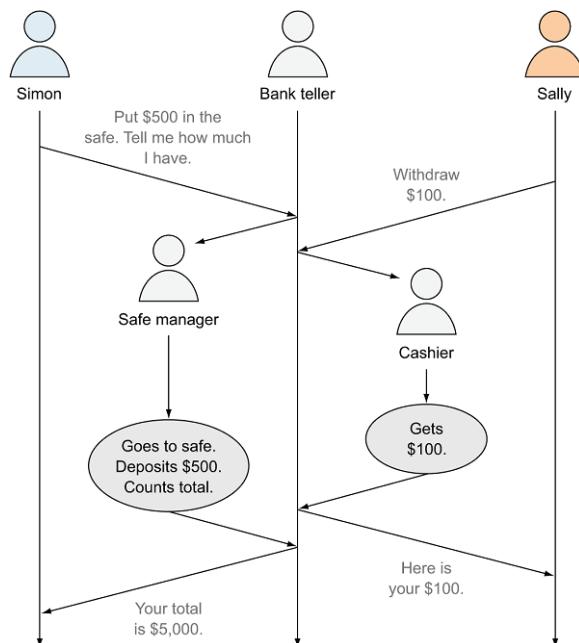
This is one of the reasons why web servers are often overpowered and have so much RAM, even though they don't need it 90% of the time. The hardware is set up in such a way as to be prepared for a huge spike in traffic. It's like the bank hiring an additional 50 full-time tellers and moving to a bigger building because they get busy at lunchtime.

Surely there's a better way, a way that's a bit more scalable? Here's where a single-threaded approach comes in.

### A SINGLE-THREADED WEB SERVER

A Node.js server is single-threaded and works differently than the multithreaded server. Rather than giving each visitor a unique thread and a separate silo of resources, every visitor joins the same thread. A visitor and thread interact only when needed, when the visitor is requesting something or the thread is responding to a request.

Returning to the bank teller analogy, there would be only one teller who deals with all of the customers. But rather than going off and managing all requests end-to-end, the teller delegates any time-consuming tasks to "back office" staff and deals with the next request. Figure 1.5 illustrates how this might work, using the same two requests from the multithreaded example.



**Figure 1.5 Example of a single-threaded approach: visitors use the same central resource. The central resource must be well disciplined to prevent one visitor from affecting others.**

In the single-threaded approach shown in figure 1.5, Sally and Simon both give their requests to the same bank teller. But instead of dealing with one of them entirely before the next, the teller takes the first request and passes it to the best person to deal with it, before taking the next request and doing the same thing. When the teller is told that the requested task is completed, the teller then passes this straight back to the visitor who requested it.

### **Blocking versus nonblocking code**

With the single-threaded model, it's important to remember that all of your users use the same central process. To keep the flow smooth, you need to make sure that nothing in your code causes a delay, blocking another operation. An example would be if the bank teller has to go to the safe to deposit the money for Simon, Sally would have to wait to make her request.

Similarly, if your central process is responsible for reading each static file (such as CSS, JavaScript, or images) it won't be able to process any other request, thus blocking the flow. Another common task that's potentially blocking is interacting with a database. If your process is going to the database each time it's asked, be it searching for data or saving data, it won't be able to do anything else.

So for the single-threaded approach to work, you must make sure your code is nonblocking. The way to achieve this is to make any blocking operations run asynchronously, preventing them from blocking the flow of your main process.

Despite there being just a single teller, neither of the visitors is aware of the other, and neither is affected by the requests of the other. This approach means that the bank doesn't need several tellers constantly on hand. This model isn't infinitely scalable, of course, but it's more efficient. You can do more with fewer resources. This doesn't mean that you'll never need to add more resources.

This particular approach is possible in Node.js due to the asynchronous capabilities of JavaScript. You'll see this in action throughout the book, but if you're not sure on the theory, check out bonus chapter 1 (available online or in the ebook), particularly the section on callbacks.

### **1.2.3 Using prebuilt packages via npm**

A package manager, npm, gets installed when you install Node.js. npm gives you the ability to download Node.js modules or *packages* to extend the functionality of your application. At the time of writing there are more than 350,000 packages available through npm, giving you an indication of just how much depth of knowledge and experience you can bring into the application. This is up from 46,000 when the first edition of Getting MEAN was written two years ago!

Packages in npm vary widely in what they give you. We'll use some throughout this book to bring in an application framework and database driver with schema support. Other examples include helper libraries like *Underscore*, testing frameworks like *Mocha*, and other utilities like *Colors*, which adds color support to Node.js console logs.

We'll look more closely at npm and how it works when we get started building an application in chapter 3.

As you've seen, Node.js is extremely powerful and flexible, but it doesn't give you much help when trying to create a website or application. You can use Express to give you a hand here. Express is installed using npm.

## 1.3 Introducing Express: The framework

Express is the E in MEAN. Because Node.js is a platform, it doesn't prescribe how it should be set up or used. This is one of its great strengths. But when creating websites and web applications, there are quite a few common tasks that need doing every time. Express is a web application framework for Node.js that has been designed to do this in a well-tested and repeatable way.

### 1.3.1 Easing your server setup

As already noted, Node.js is a platform, not a server. This allows you to get creative with your server setup and do things that other web servers can't do. It also makes it harder to get a basic website up and running.

Express abstracts away this difficulty by setting up a web server to listen to incoming requests and return relevant responses. In addition, it also defines a directory structure. One of these folders is set up to serve static files in a nonblocking way; the last thing you want is for your application to have to wait when somebody else requests a CSS file! You could configure this yourself directly in Node.js, but Express does it for you.

### 1.3.2 Routing URLs to responses

One of the great features of Express is that it provides a really simple interface for directing an incoming URL to a certain piece of code. Whether this is going to serve a static HTML page, read from a database, or write to a database doesn't really matter. The interface is simple and consistent.

What Express has done is abstract away some of the complexity of doing this in native Node.js, to make code quicker to write and easier to maintain.

### 1.3.3 Views: HTML responses

It's likely that you'll want to respond to many of the requests to your application by sending some HTML to the browser. By now it will come as no surprise to you that Express makes this easier than it is in native Node.js.

Express provides support for many different templating engines that make it easier to build HTML pages in an intelligent way, using reusable components as well as data from your application. Express compiles these together and serves them to the browser as HTML.

### 1.3.4 Remembering visitors with session support

Being single-threaded, Node.js doesn't remember a visitor from one request to the next. It doesn't have a silo of RAM set aside just for you; it sees only a series of HTTP requests. HTTP is a stateless protocol, so there's no concept of storing a session state. As it stands, this makes it difficult to create a personalized experience in Node.js or have a secure area where a user has to log in: it's not much use if the site forgets who you are on every page. You can do it, of course, but you have to code it yourself.

Or, you'll never guess what: Express has an answer to this too! Express comes with the ability to use *sessions* so that you can identify individual visitors through multiple requests and pages. Thank you Express!

Sitting on top of Node.js, Express gives you a great helping hand and a sound starting point for building web applications. It abstracts away many complexities and repeatable tasks that most of us don't need—or want—to worry about. We just want to build web applications.

## 1.4 Introducing MongoDB: The database

The ability to store and use data is vital for most applications. In the MEAN stack, the database of choice is MongoDB, the M in MEAN. MongoDB fits into the stack incredibly well. Like Node.js, it's renowned for being fast and scalable.

### 1.4.1 Relational versus document databases

If you've used a relational database before, or even a spreadsheet, you'll be used to the concept of columns and rows. Typically, a column defines the name and data type and each row would be a different entry. See table 1.1 for an example of this.

**Table 1.1 How rows and columns can look in a relational database table**

firstName	middleName	lastName	maidenName	nickname
Simon	David	Holmes		Si
Sally	June	Panayiotou		
Rebecca		Norman	Holmes	Bec

MongoDB is *not* like that! MongoDB is a document database. The concept of rows still exists but columns are removed from the picture. Rather than a column defining what should be in the row, each row is a document, and this document both defines and holds the data itself. See table 1.2 for how a collection of documents might be listed (the indented layout is for readability, not a visualization of columns).

**Table 1.2 Each document in a document database defines and holds the data, in no particular order.**

firstName: "Simon"	middleName: "David"	lastName: "Holmes"	nickname: "Si"	firstName: "Simon"
lastName: "Panayiotou"	middleName: "June"	firstName: "Sally"		lastName: "Panayiotou"
maidenName: "Holmes"	firstName: "Rebecca"	lastName: "Norman"	nickname: "Bec"	maidenName: "Holmes"

This less-structured approach means that a collection of documents could have a wide variety of data inside. Let's take a look at a sample document so that you've got a better idea of what I'm talking about.

### 1.4.2 MongoDB documents: JavaScript data store

MongoDB stores documents as BSON, which is binary JSON (JavaScript Serialized Object Notation). Don't worry for now if you're not fully familiar with JSON—check out the relevant section in bonus chapter 1. In short, JSON is a JavaScript way of holding data, hence why MongoDB fits so well into the JavaScript-centric MEAN stack!

The following code snippet shows a very simple sample MongoDB document:

```
{
  "firstName" : "Simon",
  "lastName" : "Holmes",
  "_id" : ObjectId("52279effc62ca8b0c1000007")
}
```

Even if you don't know JSON that well, you can probably see that this document stores the first and last names of Simon Holmes! So rather than a document holding a data set that corresponds to a set of columns, a document holds name and value pairs. This makes a document useful in its own right, because it both describes and defines the data.

A quick word about `_id`. You most likely noticed the `_id` entry alongside the names in the preceding example MongoDB document. The `_id` entity is a unique identifier that MongoDB assigns to any new document when it's created.

We'll look at MongoDB documents in more detail in chapter 5 when we start to add the data into our application.

### 1.4.3 More than just a document database

MongoDB sets itself apart from many other document databases with its support for secondary indexing and rich queries. This means that you can create indexes on more than just the unique identifier field, and querying indexed fields is much faster. You can also create some fairly complex queries against a MongoDB database—not to the level of huge SQL commands with joins all over the place, but powerful enough for most use cases.

As we build an application through the course of this book, we'll get to have some fun with this, and you'll start to appreciate exactly what MongoDB can do.

#### **1.4.4 What is MongoDB not good for?**

MongoDB isn't a transactional database and shouldn't be used as such. A transactional database can take several separate operations as one transaction. If any one of the operations in a transaction should fail, the entire transaction fails and none of the operations complete. MongoDB does *not* work like this. MongoDB takes each of the operations independently; if one fails then it alone fails and the rest of the operations continue.

This is important if you need to update multiple collections or documents at once. If you're building a shopping cart, for example, you need to make sure that the payment is made and recorded, and also that the order is marked as confirmed to be processed. You certainly don't want to entertain the possibility that a customer might have paid for an order that your system thinks is still in the checkout. So these two operations need to be tied together in one *transaction*. Your database structure might allow you to do this in one collection, or you might code fallbacks and safety nets into your application logic in case one fails, or you might choose to use a transactional database.

#### **1.4.5 Mongoose for data modeling and more**

MongoDB's flexibility about what it stores in documents is a great thing for the database. But most applications need some structure to their data. Note that it's the application that needs the structure, not the database. So where does it make most sense to define the structure of your application data? In the application itself!

To this end, the company behind MongoDB created Mongoose. In their own words, Mongoose provides "elegant MongoDB object modeling for Node.js" (<http://mongoosejs.com/>).

#### **WHAT IS DATA MODELING?**

Data modeling, in the context of Mongoose and MongoDB, is defining what data *can* be in a document, and what data *must* be in a document. When storing user information, you might want to be able to save the first name, last name, email address, and phone number. But you *need* only the first name and email address, and the email address must be unique. This information is defined in a schema, which is used as the basis for the data model.

#### **WHAT ELSE DOES MONGOOSE OFFER?**

As well as modeling data, Mongoose adds an entire layer of features on top of MongoDB that are useful when building web applications. Mongoose makes it easier to manage the connections to your MongoDB database, and to save and read data. We'll use all of this later. We'll also discuss how Mongoose enables you to add data validation at the schema level, making sure that you allow only valid data to be saved in the database.

MongoDB is a great choice of database for most web applications because it provides a balance between the speed of pure document databases and the power of relational databases. That the data is effectively stored in JSON makes it the perfect data store for the MEAN stack.

## 1.5 Introducing Angular: The front-end framework

Angular is the A in MEAN. In simple terms, Angular is a JavaScript framework for creating the interface for your website or application. In this book, you'll be working with Angular 2.

You could use Node.js, Express, and MongoDB to build a fully functioning data-driven web application. And we'll do just this as part of the book. But you can put some icing on the cake by adding Angular to the stack.

The traditional way of doing things is to have all data processing and application logic on the server, which then passes HTML to the browser. Angular enables you to move some or all of this processing and logic to the browser, often leaving the server just passing data from the database. We'll take a look at this in a moment when we discuss data binding, but first we need to address the question of whether Angular is like jQuery, the leading front-end JavaScript library.

### 1.5.1 jQuery versus Angular

If you're familiar with jQuery, you might be wondering if Angular works the same way. The short answer is no, not really. jQuery is generally added to a page to provide interactivity, after the HTML has been sent to the browser and the Document Object Model (DOM) has completely loaded. Angular comes in a step earlier puts the HTML together based on the data provided.

Also, jQuery is a library, and as such has a collection of features that you can use as you wish. Angular is what is known as an *opinionated framework*. This means that it forces its opinion on you as to how it needs to be used.

As mentioned, Angular helps put the HTML together based on the data provided, but it does more than this. It also immediately updates the HTML if the data changes, and can also update the data if the HTML changes. This is known as two-way data binding, which we'll now take a quick look at.

### 1.5.2 Two-way data binding: Working with data in a page

To understand two-way data binding let's use a simple example and compare this approach with traditional one-way data binding. Imagine we have a web page and some data, and we want to:

1. Display that data as a list to the user
2. Allow the user to filter that list by inputting text into a form field.

With both approaches - one-way and two-way binding - step one above is very similar: you take the data and use it to generate some HTML markup for the end user to see. It is step two where things get a bit different.

In step two we want to let the user enter some text into a form field to filter the list of data being displayed. With one-way data binding you would have to manually add event listeners to the form input field, in order to capture the data and update the data model (to ultimately change what is displayed to the user).

With two-way data binding any updates to the form can be captured automatically, updating the model and changing what is displayed to the user. This might not sound like a big deal when written down, but to highlight the power of this capability it is good to know that with Angular you can achieve everything from steps one and two without writing a single line of JavaScript code! That's right, it's all done with Angular's two-way data binding ... and a bit of help from some other Angular features.

As we go through part 3 of the book you'll really get to see—and use—this in action. Seeing is believing with this, and you won't be disappointed.

### 1.5.3 Using Angular to load new pages

Something that Angular has been specifically designed for is *single-page application* (SPA) functionality. In real terms, an SPA runs everything inside the browser and never does a full page reload. What this means is that all application logic, data processing, user flow, and template delivery can be managed in the browser.

Think Gmail. That's an SPA. Different views get shown in the page, along with a whole variety of data sets, but the page itself never fully reloads.

This approach can really reduce the amount of resources you need on your server, because you're essentially crowd-sourcing the computational power. Each person's browser is doing the hard work, and your server is basically just serving up static files and data on request.

The user experience can also be better when using this approach. Once the application is loaded there are fewer calls to be made to the server, reducing the potential of latency.

All this sounds great, but surely there's a price to pay? Why isn't everything built in Angular?

### 1.5.4 Are there any downsides?

Despite its many benefits, Angular isn't appropriate for every website. Front-end libraries like jQuery are best used for progressive enhancement. The idea is that your site will function perfectly well without JavaScript, and the JavaScript you use makes the experience better. That isn't the case with Angular, or indeed any other SPA framework. Angular uses JavaScript to build the rendered HTML from templates and data, so if your browser doesn't support JavaScript, or if there's a bug in the code, then the site won't run.

This reliance on JavaScript to build the page also causes problems with search engines. When a search engine crawls your site, it will not run all JavaScript, and with Angular the only

thing you get before JavaScript takes over is the base template from the server. If you want to be 100% certain that your content and data indexed by search engines rather than just your templates, you'll need to think whether Angular is right for that project.

There are ways to combat this issue—in short, you need your server to output compiled content as well as Angular—but if you don't *need* to fight this battle, I'd recommend against doing so.

One thing you can do is use Angular for some things and not others. There's nothing wrong with using Angular selectively in your project. For example, you might have a data-rich interactive application or section of your site that's ideal for building in Angular. You might also have a blog or some marketing pages around your application. These don't need to be built in Angular, and arguably would be better served from the server in the traditional way. So part of your site is served by Node.js, Express, and MongoDB, and another part also has Angular doing its thing.

This flexible approach is one of the most powerful aspects of the MEAN stack. With one stack you can achieve a great many different things, so long as you remember to be flexible in your thinking and don't think of the MEAN stack as a single architecture stack.

### **1.5.5 Developing in TypeScript**

Angular applications can be written in many different flavors of JavaScript, including ES5, ES6, and Dart. But the most popular by far is TypeScript.

TypeScript is a superset of a JavaScript, meaning that it *is* JavaScript, but with added features. In this book, we will be using TypeScript to build the Angular part of our application, but don't worry: we'll start from the ground up in part 3 and cover the parts of TypeScript we need to know.

## **1.6 Supporting cast**

The MEAN stack gives you everything you need for creating data-rich interactive web applications, but you may want to use a few extra technologies to help you on the way. You can use Twitter Bootstrap to help create a good user interface, Git to help manage your code, and Heroku to help by hosting the application on a live URL. In later chapters, we'll look at incorporating these into the MEAN stack. Here, we'll just cover briefly what each can do for you.

### **1.6.1 Twitter Bootstrap for user interface**

In this book we're going to use Twitter Bootstrap to create a responsive design with minimal effort. It's not essential for the stack, and if you're building an application from existing HTML or a specific design then you probably won't want to add it in. But we're going to be building an application in a "rapid prototype" style, going from idea to application with no external influences.

Bootstrap is a front-end framework that provides a wealth of help for creating a great user interface. Among its features, Bootstrap provides a responsive grid system, default styles for many interface components, and the ability to change the visual appearance with themes.

### ***RESPONSIVE GRID LAYOUT***

In a responsive layout, you serve up a single HTML page that arranges itself differently on different devices. This is done through detecting the screen resolution rather than trying to sniff out the actual device. Bootstrap targets four different pixel-width breakpoints for their layouts, loosely aimed at phones, tablets, laptops, and external monitors. So if you give a bit of thought to how you set up your HTML and CSS classes, you can use one HTML file to give the same content in different layouts suited to the screen size.

### ***CSS CLASSES AND HTML COMPONENTS***

Bootstrap comes with a set of predefined CSS classes that can create useful visual components. These include things like page headers, alert-message containers, labels and badges, stylized lists ... the list goes on! They've thought of a lot, and it really helps you quickly build an application without having to spend too much time on the HTML layout and CSS styling.

Teaching Bootstrap isn't an aim of this book, but I'll point out various features as we're using them.

### ***ADDING THEMES FOR A DIFFERENT FEEL***

Bootstrap has a default look and feel that provides a really neat baseline. This is so commonly used that your site could end up looking like anybody else's. Fortunately, it's possible to download themes for Bootstrap to give your application a different twist. Downloading a theme is often as simple as replacing the Bootstrap CSS file with a new one. We'll use a free theme in this book to build our application, but it's also possible to buy premium themes from a number of sites online to give an application a unique feel.

### ***1.6.2 Git for source control***

Saving code on your computer or a network drive is all very well and good, but that only ever holds the current version. It also only lets you, or others on your network, access it.

Git is a distributed revision control and source code management system. This means that several people can work on the same codebase at the same time on different computers and networks. These can be pushed together with all changes stored and recorded. It also makes it possible to roll back to a previous state if necessary.

## **HOW TO USE GIT**

Git is typically used from the command line, although there are GUIs available for Windows and Mac. Throughout this book, we'll use command-line statements to issue the commands that we need. Git is very powerful and we're barely going to scratch the surface of it in this book, but everything we do will be noted.

In a typical Git setup you'll have a local repository on your machine and a remote centralized master repository hosted somewhere like GitHub or BitBucket. You can pull from the remote repository into your local one, or push from local to remote. All of this is really easy in the command line, and both GitHub and BitBucket have web interfaces so that you can keep a visual track on everything committed.

## **WHAT ARE WE USING GIT FOR HERE?**

In this book we're going to be using Git for two reasons:

- First, the source code of the sample application in this book will be stored on GitHub, with different branches for various milestones. We'll be able to clone the master or the separate branches to use the code.
- Second, we'll use Git as the method for deploying our application to a live web server for the world to see. For hosting we'll be using Heroku.

### **1.6.3 Hosting with Heroku**

Hosting Node.js applications can be complicated, but it doesn't have to be. Many traditional shared hosting providers haven't kept up with the interest in Node.js. Some will install it for you so that you can run applications, but the servers are generally not set up to meet the unique needs of Node.js. To run a Node.js application successfully you need either a server that has been configured with that in mind, or you can use a PaaS provider that's aimed specifically at hosting Node.js.

In this book, we're going to go for the latter. We're going to use Heroku ([www.heroku.com](http://www.heroku.com)) as our hosting provider. Heroku is one of the leading hosts for Node.js applications, and it has an excellent free tier that we'll be making use of.

Applications on Heroku are essentially Git repositories, making the publishing process incredibly simple. After everything is set up you can publish your application to a live environment using a single command:

```
$ git push heroku master
```

I told you it didn't have to be complicated.

## 1.7 Putting it together with a practical example

As already mentioned a few times, throughout the course of this book we'll build a working application on the MEAN stack. This will give you a good grounding in each of the technologies, as well as showing how they all fit together.

### 1.7.1 Introducing the example application

So what are we actually going to be building as we go through the book? We'll be building an application called Loc8r. Loc8r will list nearby places with WiFi where people can go and get some work done. It will also display facilities, opening times, a rating, and a location map for each place. Users will be able to log in and submit ratings and reviews.

This application has some grounding in the real world. Location-based applications themselves are nothing particularly new and come in a few different guises. Swarm and Facebook Check In list everything nearby that they can, and crowd-source data for new places and information updates. UrbanSpoon helps people find nearby places to eat, allowing a user to search on price bracket and type of cuisine. Even companies like Starbucks and McDonald's have sections of their applications to help users find the nearest one.

#### **REAL OR FAKE DATA?**

Okay, so we're going to fake the data for Loc8r in this book, but you could collate the data, crowd-source it, or use an external source if you wanted. For a rapid prototype approach, you'll often find that faking data for the first private version of your application speeds up the process.

#### **END PRODUCT**

We'll use all layers of the MEAN stack to create Loc8r, including Twitter Bootstrap to help us create a responsive layout. Figure 1.6 shows some screenshots of what we're going to be building throughout the book.

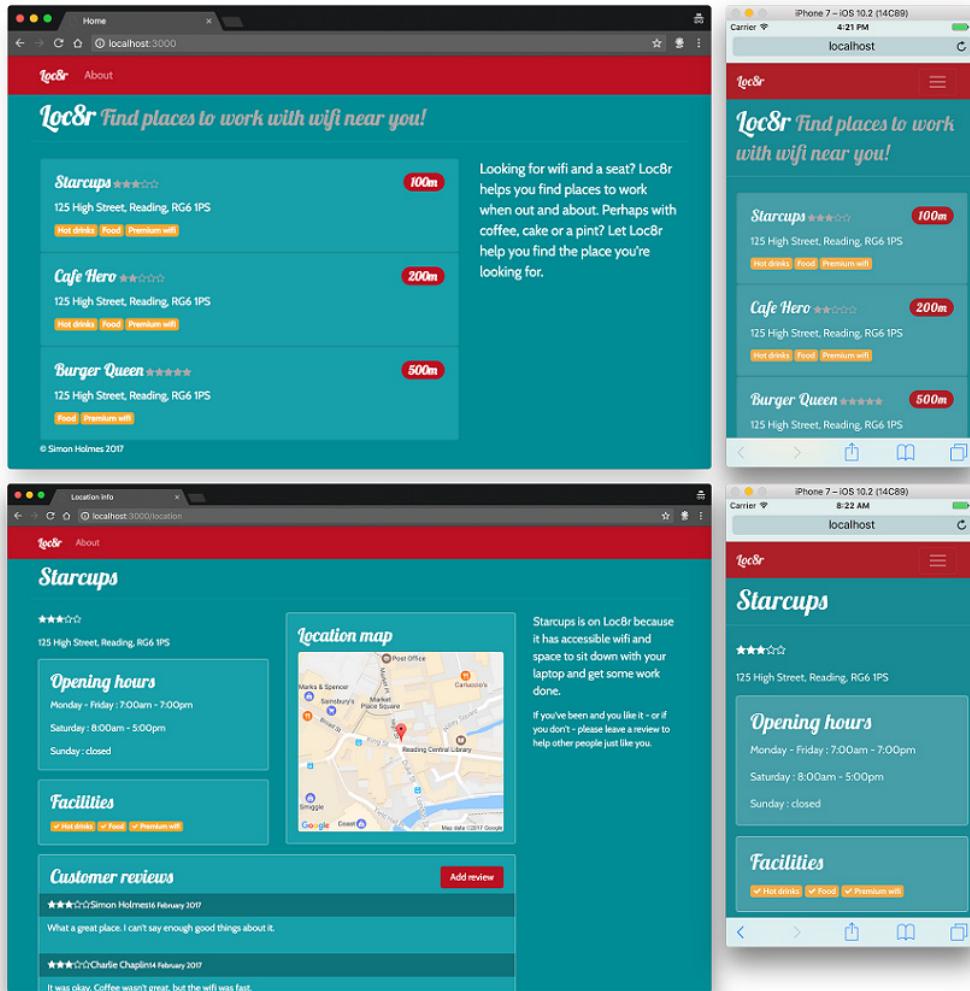


Figure 1.6 Loc8r is the application we're going to build throughout this book. It will display differently on different devices, showing a list of places and details about each place, and will allow visitors to log in and leave reviews.

### 1.7.2 How the MEAN stack components work together

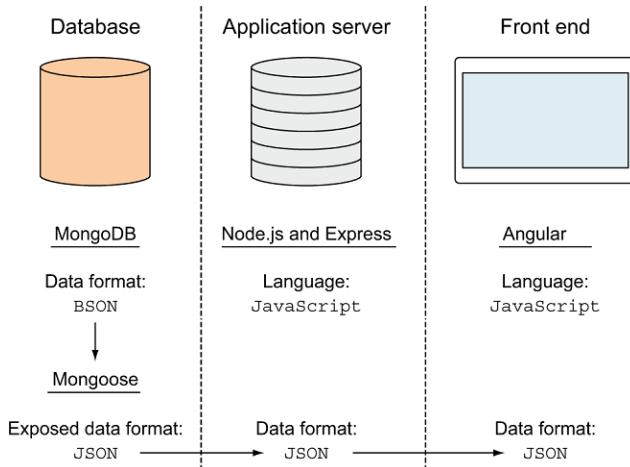
By the time you've been through this book you'll have an application running on the MEAN stack, using JavaScript all the way through. MongoDB stores data in binary JSON, which through Mongoose is exposed as JSON. The Express framework sits on top of Node.js, where

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>

the code is all written in JavaScript. In the front end is Angular, which again is JavaScript. Figure 1.7 illustrates this flow and connection.



**Figure 1.7** JavaScript is the common language throughout the MEAN stack, and JSON is the common data format.

We're going to explore various ways you can architect the MEAN stack and how we're going to build Loc8r in chapter 2.

## 1.8 Summary

In this chapter, you've learned about the following:

- The different technologies making up the MEAN stack
- MongoDB as the database layer
- Node.js and Express working together to provide an application server layer
- Angular providing an amazing front-end, data-binding layer
- How the MEAN components work together
- A few ways to extend the MEAN stack with additional technologies

Because JavaScript plays such a pivotal role in the stack, please take a look at bonus chapter 1 (available online and in the ebook), which has a refresher on JavaScript pitfalls and best practices.

Coming up next in chapter 2 we're going to discuss how flexible the MEAN stack is and how you can architect it differently for different scenarios.

# 2

## *Designing a MEAN stack architecture*

### This chapter covers

- Introducing a common MEAN stack architecture
- Considerations for single-page applications
- Discovering alternative MEAN stack architectures
- Designing an architecture for a real application
- Planning a build based on the architecture design

In chapter 1 we took a look at the component parts of the MEAN stack and how they fit together. In this chapter, we're going to look at how they fit together in more detail.

We'll start off by looking at what some people think of as *the* MEAN stack architecture, especially when they first encounter the stack. Using some examples, we'll explore why you might use a different architecture, and then switch things up a bit and move things around. MEAN is a very powerful stack and can be used to solve a diverse range of problems ... if you get creative with how you design your solutions.

### 2.1 A common MEAN stack architecture

A common way to architect a MEAN stack application is to have a representational state transfer (REST) API feeding a single-page application (SPA). The API is typically built with MongoDB, Express, and Node.js, with the SPA being built in Angular. This approach is particularly popular with those who come to the MEAN stack from an Angular background and are looking for a stack that gives a fast, responsive API. Figure 2.1 illustrates the basic setup and data flow.

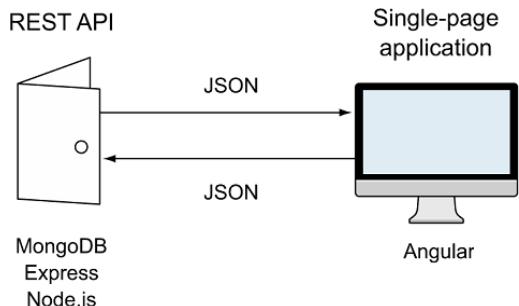


Figure 2.1 A common approach to MEAN stack architecture, using MongoDB, Express, and Node.js to build a REST API that feeds JSON data to an Angular SPA run in the browser.

### What is 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 each other. In the case of the web, an API is normally a set of URLs that respond with data when called in the correct manner with the correct information.

So a REST API is a stateless interface to your application. 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 - for example an Angular SPA - to work with the data. In other words, you create a collection of structured URLs that will return specific data when called.

Figure 2.1 is a great setup, ideal if you have or intend to build an SPA as your user-facing side. Angular is designed with a focus on building SPAs, pulling in data from a REST API, as well as pushing it back. MongoDB, Express, and Node.js are also extremely capable when it comes to building an API, using JSON all the way through the stack, including the database itself.

This is where many people start with the MEAN stack, looking for an answer to the question, “I’ve built an application in Angular; now where do I get the data?”

Having an architecture like this is great if you have an SPA, but what if you don’t have or want to use an SPA? If this is the only way you think of the MEAN stack, you’re going to get a bit stuck and start looking elsewhere. But the MEAN stack is very flexible. All four components are very powerful and have a lot to offer.

## 2.2 Looking beyond SPAs

Coding an SPA in Angular is like driving a Porsche along a coastal road with the roof down. Both are amazing. They’re fun, fast, sexy, agile, and very, very capable. And it’s most likely that both are a vast improvement on what you’ve been doing before.

But sometimes they're not appropriate. If you want to pack up the surfboards and take your family away for the week you're going to struggle with the sports car. As amazing as your car may be, in this case you're going to want to use something different. It's the same story with SPAs. Yes, building them in Angular is amazing, but sometimes an SPA is not the best solution to your problem.

Let's take a brief look at some things to bear in mind about SPAs when designing a solution and deciding whether a full SPA is right for your project or not. This section isn't intended to be in any way "anti-SPA." SPAs generally offer a fantastic user experience while reducing the load on your servers, and therefore also your hosting costs. In sections 2.3.1 and 2.3.2 we'll look at a good use case for an SPA and a bad one, and we'll actually build a full SPA by the end of this book!

### **2.2.1 Hard to crawl**

JavaScript applications are very hard for search engines to crawl and index. Most search engines look at the HTML content on a page but don't download or execute much JavaScript. For those that do, the actual crawling of JavaScript-created content is nowhere near as good as content delivered by the server. If all of your content is served via a JavaScript application, then you cannot be sure how much of it will be indexed.

A related downside is that automatic previews from social-sharing sites like Facebook, LinkedIn, and Pinterest don't work very well. This is also because they look at the HTML of the page you're linking to and try to extract some relevant text and images. Like search engines, they don't run JavaScript on the page, so content served by JavaScript won't be seen.

All of this is slowly improving and I hope that future editions of this book won't need to have this section!

---

#### **But Google says they crawl JavaScript sites!**

In 2015 Google released an update and stated that they were "generally able" to crawl JavaScript-based websites, like Angular SPAs. However, in reality "generally able" hasn't lived up to expectations and results are extremely varied.

---

#### **MAKING AN SPA CRAWLABLE**

There are a couple of workarounds to make it look as though your site is crawlable. Both involve creating separate HTML pages that mirror the content of your SPA. You can have your server create an HTML-based version of your site and deliver that to crawlers, or you can use a headless browser such as PhantomJS to run your JavaScript application and output the resulting HTML.

Both require quite a bit of effort, and can end up being a maintenance headache if you have a large, complex site. There also are potential search engine optimization (SEO) pitfalls. If your server-generated HTML is deemed to be too different from the SPA content, then your site will be penalized. Running PhantomJS to output the HTML can slow down the response

speed of your pages, which is something for which search engines—Google in particular—downgrade you.

### **DOES IT MATTER?**

Whether this matters or not depends on what you want to build. If the main growth plan for whatever you’re building is through search engine traffic or social sharing, then this is something you want to give a great deal of thought. If you’re creating something small that will stay small then managing the workarounds is achievable, whereas at a larger scale you’ll struggle.

On the other hand, if you’re building an application that doesn’t need much SEO—or indeed if you *want* your site to be harder to scrape—then this isn’t an issue you need to be concerned about. It could even be an advantage.

### **2.2.2 Analytics and browser history**

Analytics tools like Google Analytics rely heavily on entire new pages loading in the browser, initiated by a URL change. SPAs don’t work this way. There’s a reason they’re called *single-page* applications!

After the first page load, all subsequent page and content changes are handled internally by the application. So the browser never triggers a new page load, nothing gets added to the browser history, and your analytics package has no idea who’s doing what on your site.

### **ADDING PAGE LOADS TO AN SPA**

You can add page load events to an **SPA** using the HTML5 history API; this will help you integrate analytics. The difficulty comes in managing this and ensuring that everything is being tracked accurately, which involves checking for missing reports and double entries.

The good news is that you don’t have to build everything from the ground up. There are several open source analytics integrations for Angular available online, addressing most of the major analytics providers. You still have to integrate them into your application and make sure that everything is working correctly, but you don’t have to do everything from scratch.

### **IS IT A MAJOR PROBLEM?**

The extent to which this is a problem depends on your need for undeniably accurate analytics. If you want to monitor trends in visitor flows and actions, then you’re probably going to find it easy to integrate. The more detail and definite accuracy you need, the more work it is to develop and test. While it’s arguably much easier to just include your analytics code on every page of a server-generated site, analytics integration isn’t likely to be the sole reason that you choose a non-**SPA** route.

### 2.2.3 Speed of initial load

SPAs have a slower first page load than server-based applications. This is because the first load has to bring down the framework and the application code before rendering the required view as HTML in the browser. A server-based application just has to push out the required HTML to the browser, reducing the latency and download time.

#### SPEEDING UP THE PAGE LOAD

There are some ways of speeding up the initial load of an **SPA**, such as a heavy approach to caching and lazy-loading modules when you need them. But you'll never get away from the fact that it needs to download the framework, at least some of the application code, and will most likely hit an API for data before displaying something in the browser.

#### SHOULD YOU CARE ABOUT SPEED?

The answer to whether you should care about the speed of the initial page load is, once again, "it depends." It depends on what you're building and how people are going to interact with it.

Think about Gmail. Gmail is an **SPA** and takes quite a while to load. Granted this is only normally a couple of seconds, but everyone online is impatient these days and expects immediacy. But people don't mind waiting for Gmail to load, because it's snappy and responsive once you're in. And once you're in, you often stay in for a while.

But if you have a blog pulling in traffic from search engines and other external links, you don't want the first page load to take a few seconds. People will assume your site is down or running slowly and click the back button before you've had the chance to show them content. I'm willing to bet that you know this happens because you've done it yourself!

### 2.2.4 To SPA or not to SPA?

Just a reminder that this wasn't an exercise in SPA-bashing; we're just taking a moment to think about some things that often get pushed to the side until it's too late. The three points about crawlability, analytics integration, and page load speed aren't designed to give clear-cut definitions about when to create an SPA and when to do something else. They're there to give a framework for consideration.

It might be the case that none of those things is an issue for your project, and that an **SPA** is definitely the right way to go. If you find that each point makes you pause and think, and it looks like you need to add in workarounds for all three, then an **SPA** probably isn't the way to go.

If you're somewhere in between then it's a judgment call about what is most important, and, crucially, what is the best solution for the project. As a rule of thumb, if your solution includes a load of workarounds at the outset then you probably need to rethink it.

Even if you decide that an **SPA** isn't right for you, that doesn't mean that you can't use the MEAN stack. Let's move on and take a look at how you can design a different architecture.

## 2.3 Designing a flexible MEAN architecture

If Angular is like having a Porsche, then the rest of the stack is like also having an Audi RS6 in the garage. A lot of people may be focusing on your sports car out front and not give a second glance to the estate car in your garage. But if you do go into the garage and have a poke around, you'll find that there's a Lamborghini V10 engine under the hood. There's a lot more to that estate car than you might first think!

Only ever using MongoDB, Express, and Node.js together to build a REST API is like only ever using the Audi RS6 to do the school drop-off runs. They're all extremely capable and will do the job very well, but they have a lot more to offer.

We talked a little about what the technologies can do in chapter 1, but here are a few starting points:

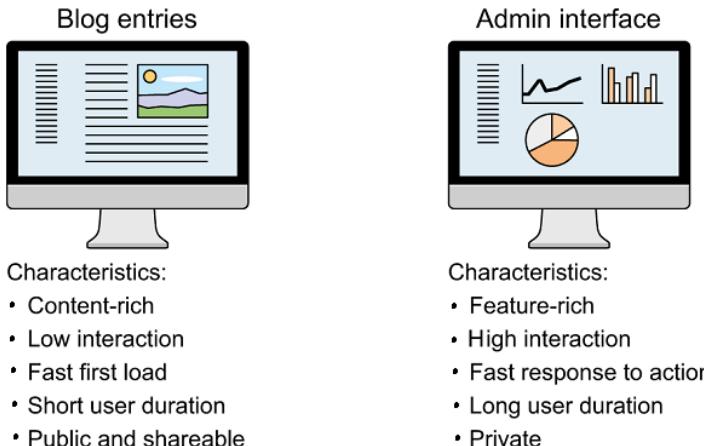
- MongoDB can store and stream binary information.
- Node.js is particularly good for real-time connections using web sockets.
- Express is a web application framework with templating, routing, and session management built in.

There's also a lot more, and I'm certainly not going to be able to address the full capabilities of all of the technologies in this book. I'd need several books to do that! What I can do here is give you a simple example and show you how you can fit together the pieces of the MEAN stack to design the best solution.

### 2.3.1 Requirements for a blog engine

Let's take a look at the familiar idea of a blog engine, and see how you could best architect the MEAN stack to build one.

A blog engine typically has two sides to it. There's a public-facing side serving up articles to readers, and hopefully being syndicated and shared across the internet. A blog engine will also have an administrator interface where blog owners log in to write new articles and manage their blogs. Figure 2.2 shows some of the key characteristics for these two sides.



**Figure 2.2 Conflicting characteristics of the two sides of a blog engine, the public-facing blog entries, and the private admin interface.**

Looking at the lists in figure 2.2, it's quite easy to see a high level of conflict between the characteristics of the two sides. You've got content-rich, low interaction for the blog articles, but a feature-rich, highly interactive environment for the admin interface. The blog articles should be quick to load to reduce bounce rates, whereas the admin area should be quick to respond to user input and actions. Finally, users typically stay on a blog entry for a short time, but may share it with others, whereas the admin interface is very private and an individual user could be logged in for a long time.

Taking what we've discussed about potential issues with SPAs, and looking at the characteristics of blog entries, you'll see quite a lot of overlap. It's quite likely that bearing this in mind you'd choose not to use an SPA to deliver your blog articles to readers. On the other hand, the admin interface is a perfect fit for an SPA.

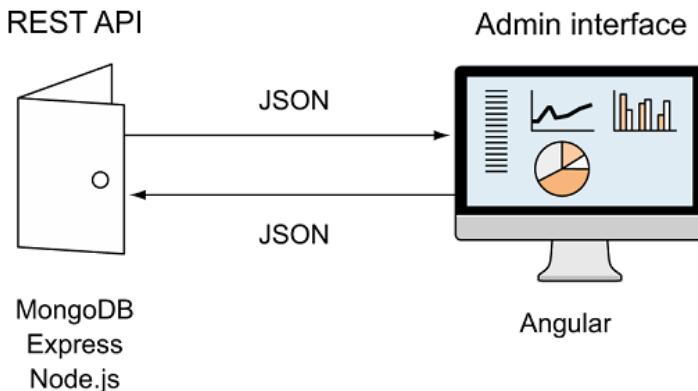
So what do you do? Arguably the most important thing is to keep the blog readers coming—if they get a bad experience they won't come back and they won't share. If a blog doesn't get readers, then the writer will stop writing or move to another platform. Then again, a slow and unresponsive admin interface will also see your blog owners jumping ship. So what do you do? How do you keep everybody happy and keep the blog engine in business?

### 2.3.2 A blog engine architecture

The answer lies in not looking for a one-size-fits-all solution. You effectively have two applications. You have public-facing content that should be delivered direct from the server and an interactive private admin interface that you want to build as an SPA. Let's start by looking at each of the two applications separately, starting with the admin interface.

### **ADMIN INTERFACE: AN ANGULAR SPA**

We've already discussed that this would be an ideal fit for an SPA built in Angular. So the architecture for this part of the engine will look very familiar: a REST API built with MongoDB, Express, and Node.js with an Angular SPA up front. Figure 2.3 shows how this looks.



**Figure 2.3** A familiar sight: the admin interface would be an Angular SPA making use of a REST API built with MongoDB, Express, and Node.js.

There's nothing particularly new shown in figure 2.3. The entire application is built in Angular and runs in the browser, with JSON data being passed back and forth between the Angular application and the REST API.

### **BLOG ENTRIES: WHAT TO DO?**

Looking at the blog entries, things get a little more difficult.

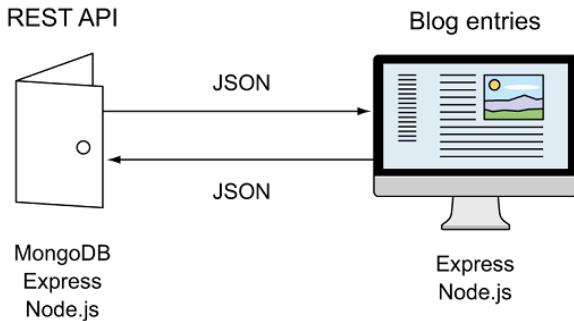
If you only think of the MEAN stack as an Angular SPA calling a REST API then you're going to get a bit stuck. You could build the public-facing site as an SPA anyway, because you want to use JavaScript and the MEAN stack. But it's not the best solution. You could decide that the MEAN stack isn't appropriate in this case and choose a different technology stack. But you don't want to do that! You want end-to-end JavaScript.

So let's take another look at the MEAN stack, and think about all of the components. You know that Express is a web application framework. You know that Express can use template engines to build HTML on the server. You know that Express can use URL routing and MVC patterns. You should start to think that perhaps Express has the answer!

### **BLOG ENTRIES: MAKING GOOD USE OF EXPRESS**

In this blog scenario, delivering the HTML and content directly from the server is exactly what you want to do. Express does this particularly well, even offering a choice of template engines

right from the get-go. The HTML content will require data from the database, so you'll use a REST API again for that (more on why it's best to take this approach in section 2.3.3). Figure 2.4 lays out the basis for this architecture.



**Figure 2.4** An architecture for delivering HTML directly from the server: an Express and Node.js application at the front, interacting with a REST API built in MongoDB, Express, and Node.js.

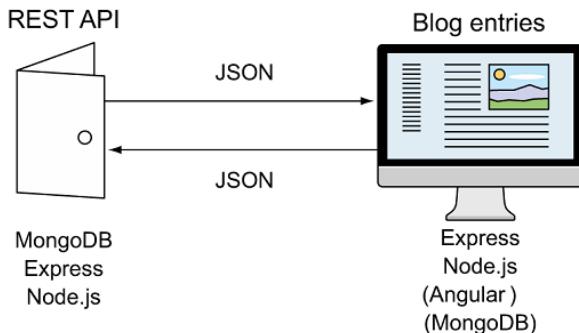
This gives you an approach where you can use the MEAN stack, or part of it at least, to deliver database-driven content directly from the server to the browser. But it doesn't have to stop there. The MEAN stack is yet again more flexible.

#### BLOG ENTRIES: USING MORE OF THE STACK

You're looking at an Express application delivering the blog content to the visitors. If you want visitors to be able to log in, perhaps to add comments to articles, you need to track user sessions. You could use MongoDB with your Express application to do just this.

You might also have some dynamic data in the sidebar of your posts, such as related posts or a search box with type-ahead auto-completion. You could implement these in Angular. Remember, Angular isn't only for SPAs; it can also be used to add some rich data interactivity to an otherwise static page.

Figure 2.5 shows these optional parts of MEAN added to the blog entry architecture.



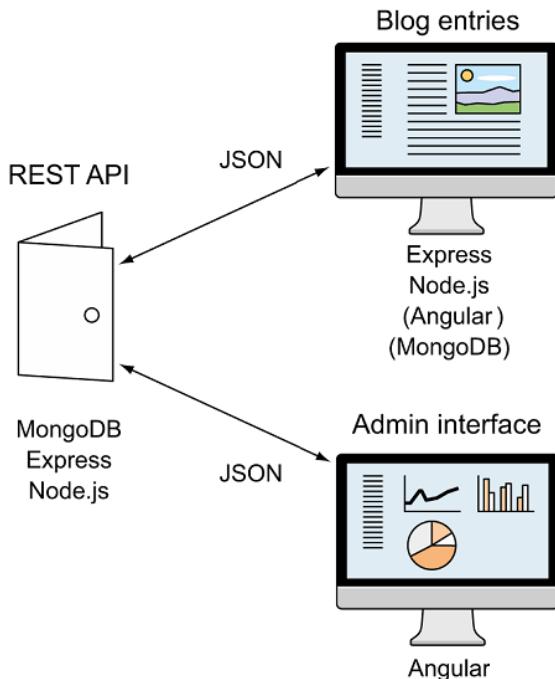
**Figure 2.5 Adding the options of using Angular and MongoDB as part of the public-facing aspect of the blog engine, serving the blog entries to visitors.**

Now you have the possibility of a full MEAN application delivering content to visitors interacting with your REST API.

#### **BLOG ENGINE: A HYBRID ARCHITECTURE**

At this point, there are two separate applications, each using a REST API. With a little bit of planning this can be a common REST API, used by both sides of the application.

Figure 2.6 shows what this looks like as a single architecture, with the one REST API interacting with the two front-end applications.



**Figure 2.6** A hybrid MEAN stack architecture: a single REST API feeding two separate user-facing applications, built using different parts of the MEAN stack to give the most appropriate solution.

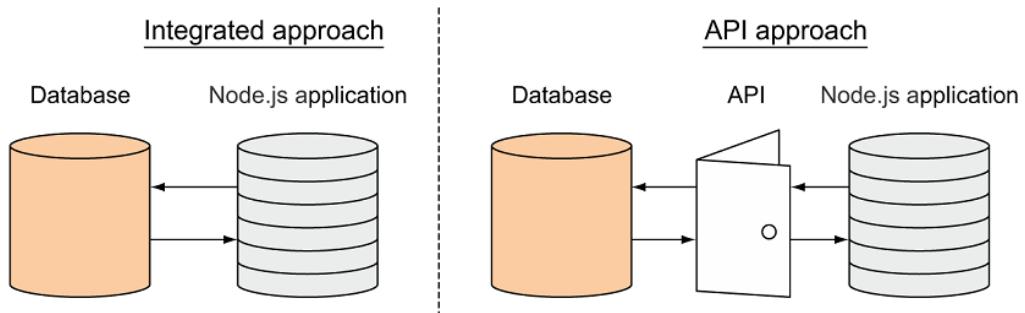
This is just a simple example to show how you can piece together the various parts of the MEAN stack into different architectures to answer the questions that your projects ask of you. Your options are only limited by your understanding of the components and your creativity in putting them together. There's no one correct architecture for the MEAN stack.

### 2.3.3 Best practice: Build an internal API for a data layer

You've probably noticed that every version of the architecture includes an API to surface the data, and allows interaction between the main application and the database. There's a good reason for this.

If you were to start off by building your application in Node.js and Express, serving HTML directly from the server, it would be really easy to talk to the database directly from the Node.js application code. With a short-term view this is the easy way. But with a long-term view this becomes the difficult way, because it will tightly couple your data to your application code in a way that nothing else could use it.

The other option is to build your own **API** that can talk to the database directly and output the data you need. Your Node.js application can then talk with this **API** instead of directly with the database. Figure 2.7 shows a comparison of the two setups.



**Figure 2.7** The short-term view of data integration into your Node.js application. You can set up your Node.js application to talk directly to your database, or you can create an API that interacts with the database and have your Node.js application talk only with the API.

Looking at figure 2.7 you could well be wondering why you'd want to go to the effort of creating an **API** just to sit in between your application and your database. Isn't it creating more work? At this stage, yes, it's creating more work—but you want to look further down the road here. What if you want to use your data in a native mobile application a little later? Or, for example, in an Angular front end?

You certainly don't want to find yourself in the position where you have to write separate but similar interfaces for each. If you've built your own **API** up front that outputs the data you need, you can avoid all of this. If you have an **API** in place, when you want to integrate the data layer into your application you can simply make it reference your **API**. It doesn't matter if your application is Node.js, Angular, or iOS. It doesn't have to be a public **API** that anyone can use, so long as you can access it. Figure 2.8 shows a comparison of the two approaches when you have Node.js, Angular, and iOS applications all using the same data source.

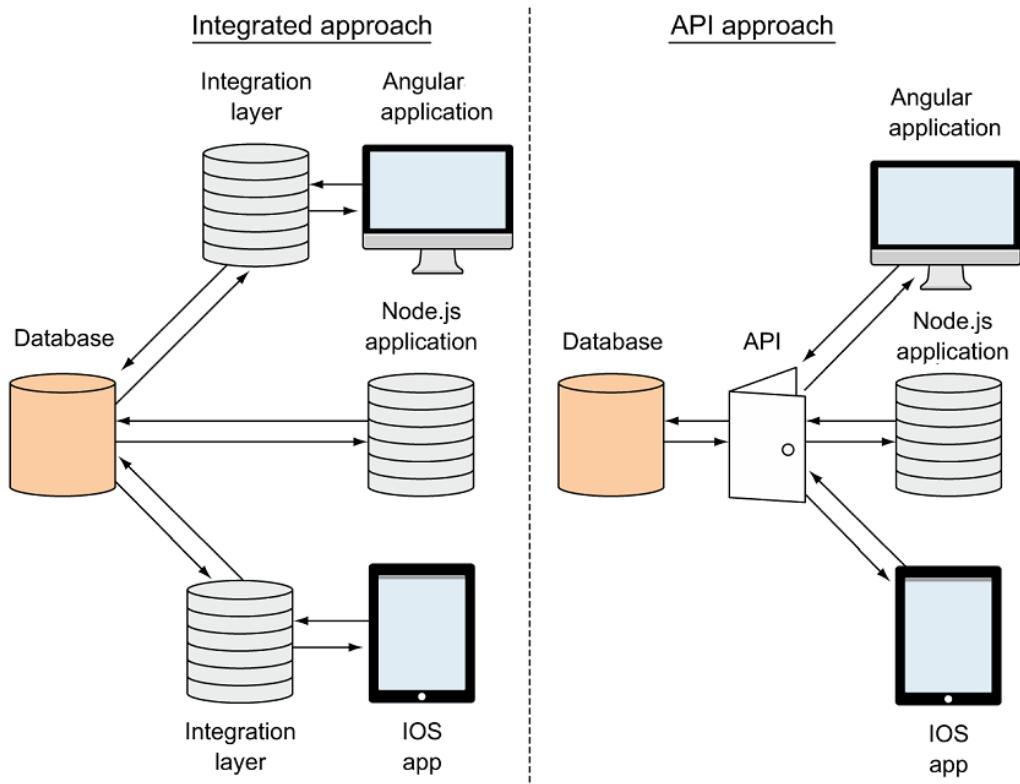


Figure 2.8 The long-term view of data integration into your Node.js application, and additional Angular and iOS applications. The integrated approach has now become fragmented, whereas the API approach is simple and maintainable.

As figure 2.8 shows, the previously simple integrated approach is now becoming fragmented and complex. You'll have three data integrations to manage and maintain, so any changes will have to be made in multiple places to retain consistency. If you have a single API, you don't have any of these worries. So with a little bit of extra work at the beginning, you can make life much easier for yourself further down the road. We'll look at creating internal APIs in chapter 6.

## 2.4 Planning a real application

As we talked about in chapter 1, throughout the course of this book we'll build a working application on the MEAN stack called Loc8r. Loc8r will list nearby places with WiFi where people can go and get some work done. It will also display facilities, opening times, a rating, and a location map for each place. Visitors will be able to submit ratings and reviews.

For the sake of the demo application, we'll be creating fake data so that we can test it quickly and easily. So let's get planning.

### **2.4.1 Planning the application at a high level**

The first step is to think about what screens we'll need in our application. We'll focus on the separate page views and the user journeys. We can do this at a very high level, not really concerning ourselves with the details of what is on each page. It is a good idea to sketch out this stage on a piece of paper or a whiteboard, as it helps to visualize the application as a whole. It also helps with organizing the screens into collections and flows, while serving as a good reference point for when we come to build it. As there's no data attached to the pages or application logic behind them, it's really easy to add and remove parts, change what is displayed where, and even change how many pages we want. The chances are that we won't get it right the first time; the key is to start and iterate and improve until we're happy with the separate pages and overall user flow.

#### **PLANNING THE SCREENS**

Let's think about Loc8r. As stated our aim is as follows:

*Loc8r will list nearby places with WiFi where people can go and get some work done. It will also display facilities, opening times, a rating, and a location map for each place. Visitors will be able to submit ratings and reviews.*

From this we can get an idea about some of the screens we're going to need:

1. A screen that lists nearby places
2. A screen that shows details about an individual place
3. A screen for adding a review about a place

We'll probably also want to tell visitors what Loc8r is for and why it exists, so we should add another screen to the list:

4. A screen for "about us" information

#### **DIVIDING THE SCREENS INTO COLLECTIONS**

Next, we want to take the list of screens and collate them where they logically belong together. For example, the first three in the list are all dealing with locations. The About page doesn't really belong anywhere so it can go in a miscellaneous Others collection. Sketching this out brings us something like figure 2.9.

## Locations

List page



Details page



Add Review page



## Others

About page

**Figure 2.9 Collate the separate screens for our application into logical collections.**

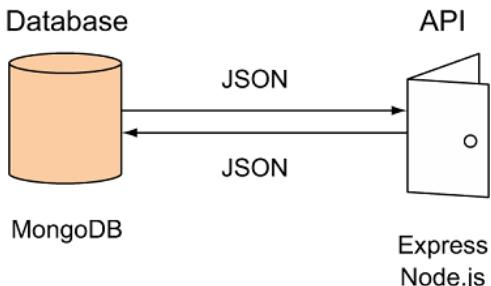
Having a quick sketch like this is the first stage in planning, and we really need to go through this stage before we can start thinking about architecture. This stage gives us a chance to look at the basic pages, and to also think about the flow. Figure 2.9, for example, also shows a basic user journey in the Locations collection, going from the List page, to a Details page, and then onto the form to add a review.

### **2.4.2 Architecting the application**

On the face of it Loc8r is a fairly simple application, with just a few screens. But we still need to think about how to architect it, because we're going to be transferring data from a database to a browser, letting users interact with the data and allowing data to be sent back to the database.

#### **STARTING WITH THE API**

Because the application is going to be using a database and passing data around, we'll start building the architecture with the piece we're definitely going to need. Figure 2.10 shows the starting point, a REST API built with Express and Node.js to enable interactions with the MongoDB database.



**Figure 2.10 Start with the standard MEAN REST API, using MongoDB, Express, and Node.js.**

As already discussed in this chapter, building an API to interface with our data is a bit of a given and is the base point of the architecture. So the more interesting and difficult question is: How do we architect the application itself?

#### **APPLICATION ARCHITECTURE OPTIONS**

At this point, we need to take a look at the specific requirements of our application and how we can put together the pieces of the MEAN stack to build the best solution. Do we need something special from MongoDB, Express, Angular, or Node.js that will swing the decision a certain way? Do we want HTML served directly from the server, or is an SPA the better option?

For Loc8r there are no unusual or specific requirements, and whether or not it should be easily crawlable by search engines depends on the business growth plan. If the aim is to bring in organic traffic from search engines, then yes, it needs to be crawlable. If the aim is to promote the application as an application and drive use that way, then search engine visibility is a lesser concern.

Thinking back to the blog example, we can immediately envisage three possible application architectures, as shown in figure 2.11:

1. A Node.js and Express application
2. A Node.js and Express application with Angular additions for interactivity
3. An Angular SPA

With these three options in mind, which is the best for Loc8r?

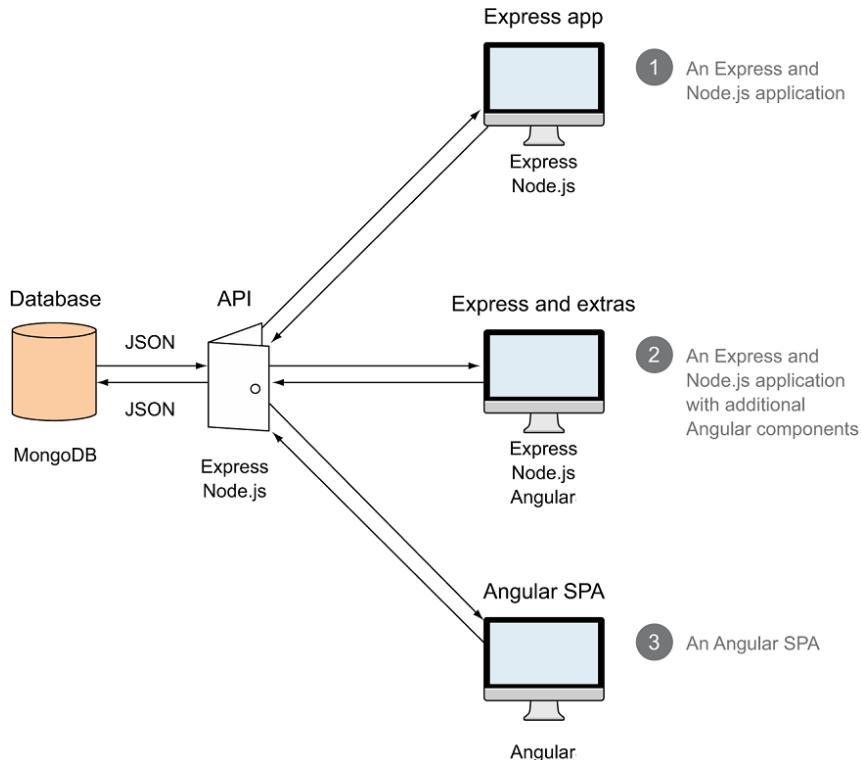


Figure 2.11 Three options for building the Loc8r application, ranging from a server-side Express and Node.js application to a full client-side Angular SPA.

### CHOOSING AN APPLICATION ARCHITECTURE

There are no specific business requirements pushing us to favor one particular architecture over another. It doesn't matter, because we're going to do all three in this book. Building all three of the architectures will allow us to explore how each approach works and enable us to take a look at each of the technologies in turn, building up the application layer by layer.

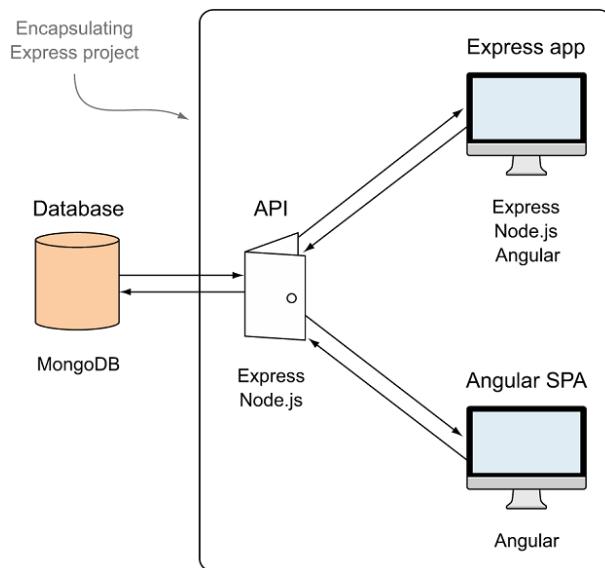
We'll be building the architectures in the order they're shown in figure 2.11, starting with a Node.js and Express application, then moving on to add some Angular before refactoring to an Angular SPA. Although this isn't necessarily how you might build a site normally, it gives you a great opportunity for learning all aspects of the MEAN stack. We'll talk shortly in section 2.5 about the approach and walk through the plan in a bit more detail.

#### 2.4.3 Wrapping everything in an Express project

The architecture diagrams we've been looking at so far imply that we'll have separate Express applications for the API and the application logic. This is perfectly possible and a good way to

go for a large project. If we're expecting large amounts of traffic, we might even want our main application and our API on different servers. An additional benefit of this is that we can have more specific settings for each of the servers and applications that are best suited to the individual needs.

Another way is to keep things simple and contained and have everything inside a single Express project. With this approach, we have only one application to worry about hosting and deploying and one set of source code to manage. This is what we'll be doing with Loc8r, giving us one Express project containing a few sub-applications. Figure 2.12 illustrates this approach.



**Figure 2.12** The architecture of the application with the API and application logic wrapped inside the same Express project.

When putting together an application in this way, it's important to organize our code well so that the distinct parts of the application are kept separate. As well as making our code easier to maintain, it also makes it easier to split it out into separate projects further down the line if we decide that's the right route. This is a key theme that we'll keep coming back to throughout the book.

#### 2.4.4 The end product

As you can see, we'll use all layers of the MEAN stack to create Loc8r. We'll also include Twitter Bootstrap to help us create a responsive layout. Figure 2.13 shows some screenshots of what we're going to be building throughout the book.

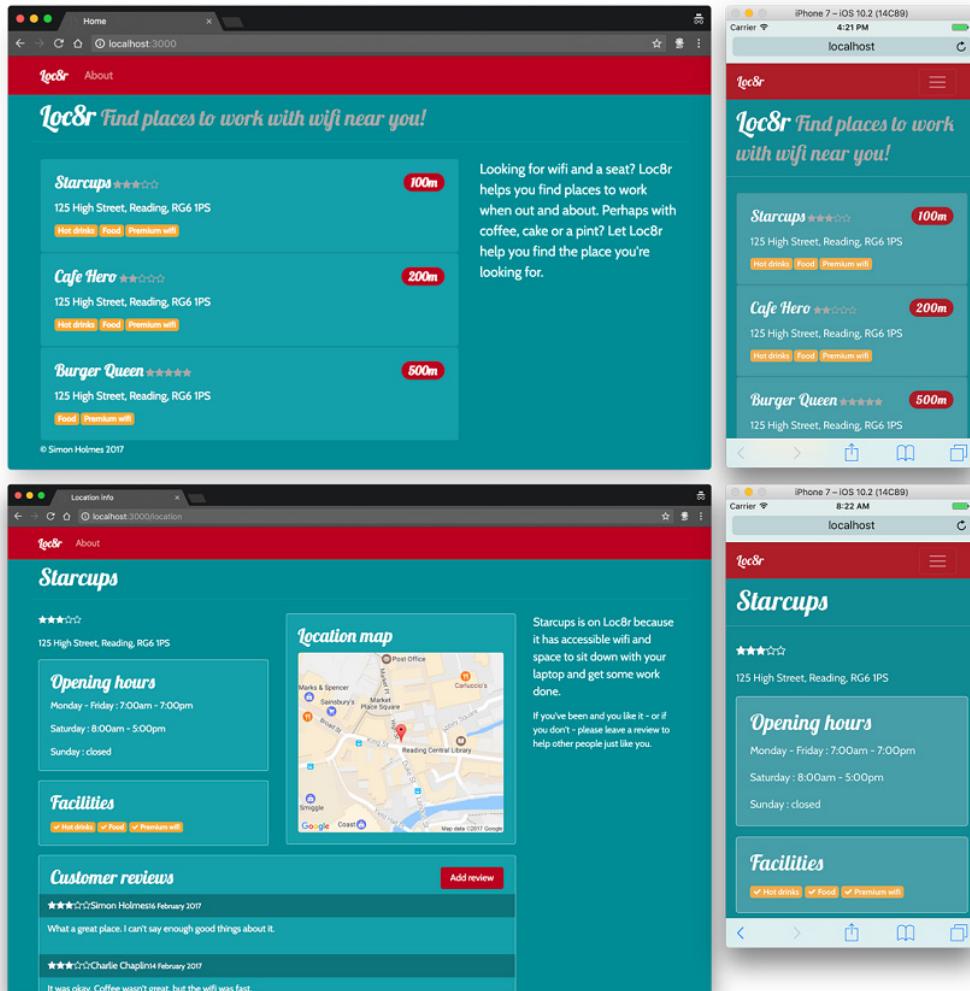


Figure 2.13 Loc8r is the application we're going to build throughout this book. It will display differently on different devices, showing a list of places and details about each place, and will enable visitors to log in and leave reviews.

## 2.5 Breaking the development into stages

In this book, we have two aims:

1. Build an application on the MEAN stack
2. Learn about the different layers of the stack as we go

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>

We'll approach the project in the way I'd personally go about building a rapid prototype, but with a few tweaks to give you the best coverage of the whole stack. We'll start by looking at the five stages of rapid prototype development, and then see how we can use this approach to build up Loc8r layer by layer, focusing on the different technologies as we go.

### **2.5.1 Rapid prototype development stages**

Let's break down the process into stages, which lets us concentrate on one thing at a time, increasing our chances of success. I find this approach works well for making an idea a reality.

#### **STAGE 1: BUILD A STATIC SITE**

The first stage is to build a static version of the application, which is essentially a number of HTML screens. The aims of this stage are

- To quickly figure out the layout
- To ensure that the user flow makes sense

At this point we're not concerned with a database or flashy effects on the user interface; all we want to do is create a working mockup of the main screens and journeys that a user will take through the application.

#### **STAGE 2: DESIGN THE DATA MODEL AND CREATE THE DATABASE**

When we have a working static prototype that we're happy with, the next thing to do is look at any hard-coded data in the static application and put it into a database. The aims of this stage are

- To define a data model that reflects the requirements of the application
- To create a database to work with the model

The first part of this is to define the data model. Stepping back to a bird's-eye view, what are the objects we need data about, how are the objects connected, and what data is held in them?

If we try to do this stage before building the static prototype, we're dealing with abstract concepts and ideas. When we have a prototype, we can see what is happening on different pages and what data is needed where. Suddenly this stage becomes much easier. Almost unknown to us, we've done the hard thinking while building the static prototype.

#### **STAGE 3: BUILD OUR DATA API**

After stages 1 and 2, we have a static site on one hand and a database on the other. This stage and the next take the natural steps of linking them together. The aim of stage 3 is

- To create a REST API that will allow our application to interact with the database

#### **STAGE 4: HOOK THE DATABASE INTO THE APPLICATION**

When we get to this stage, we have a static application and an API exposing an interface to our database. The aim of this stage is

- To get our application to talk to our API

When this stage is complete, the application will look pretty much the same as it did before, but the data will be coming from the database. When it's done, we'll have a data-driven application!

#### **STAGE 5: AUGMENT THE APPLICATION**

This stage is all about embellishing the application with additional functionality. We might add authentication systems, data validation, or methods for displaying error messages to users. It could include adding more interactivity to the front end or tightening up the business logic in the application.

So, really, the aims of this stage are

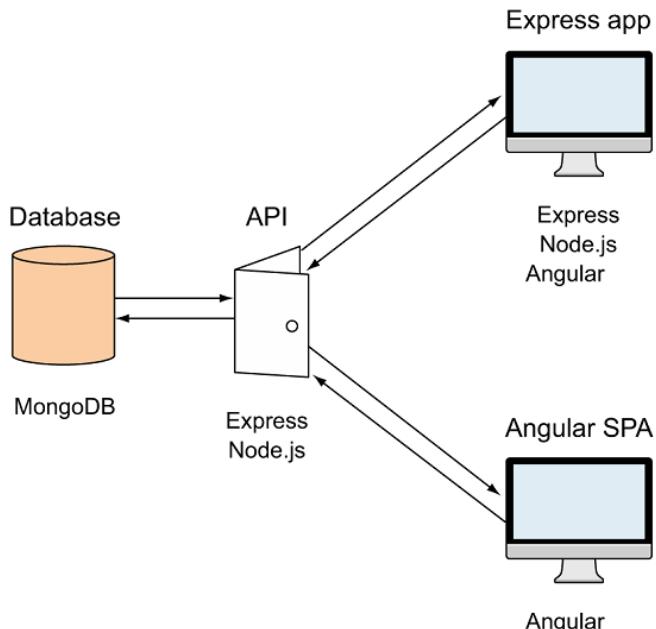
- To add finishing touches to our application
- To get the application ready for people to use

These five stages of development provide a great methodology for approaching a new build project. Let's take a look at how we'll follow these steps to build Loc8r.

#### **2.5.2 The steps to build Loc8r**

In building Loc8r throughout this book, we have two aims. First, of course, we want to build a working application on the MEAN stack. Second, we want to learn about the different technologies, how to use them, and how to put them together in different ways.

So throughout the book we'll be following the five stages of development, but with a couple of twists so that we get to see the whole stack in action. Before we look at the steps in detail, let's quickly remind ourselves of the proposed architecture as shown in figure 2.14.



**Figure 2.14 Proposed architecture for Loc8r as we'll build it throughout this book. Step 1: Build a static site**

We'll start by following stage 1 and building a static site. I recommend doing this for any application or site, because you can learn a lot with relatively little effort. When building the static site, it's good to keep one eye on the future, keeping in mind what the final architecture will be. We've already defined the architecture for Loc8r, as shown in figure 2.14.

Based on this architecture we'll build the static application in Node and Express, using that as our starting point into the MEAN stack. Figure 2.15 highlights this step in the process as the first part of developing the proposed architecture.

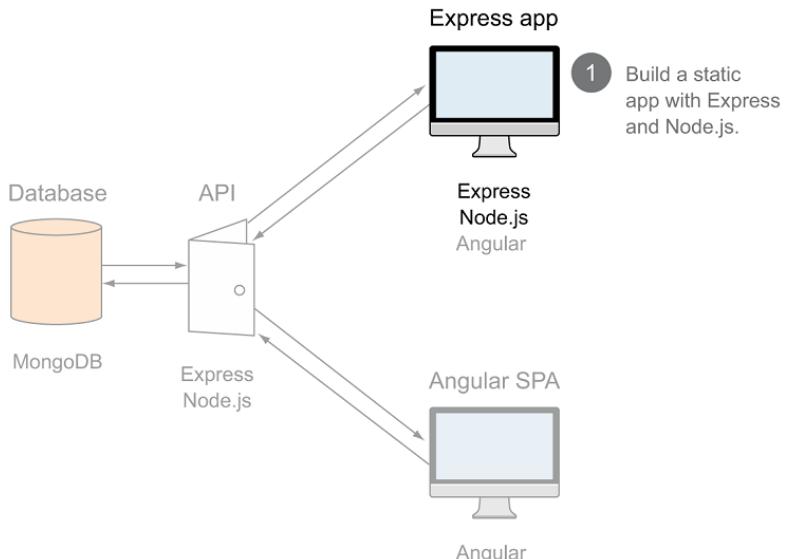


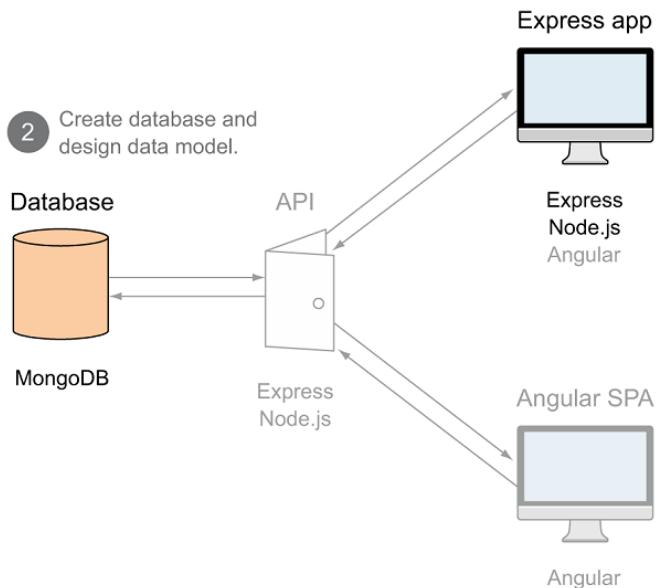
Figure 2.15 The starting point for our application is building the user interface in Express and Node.js.

This step is covered in chapters 3 and 4.

#### Step 2: Design the data model and create the database

Still following the stages of development, we'll continue to stage 2 by creating the database and designing the data model. Again, any application is likely to need this step, and you'll get much more out of it if you've been through step 1 first.

Figure 2.16 illustrates how this step adds to the overall picture of building up the application architecture.



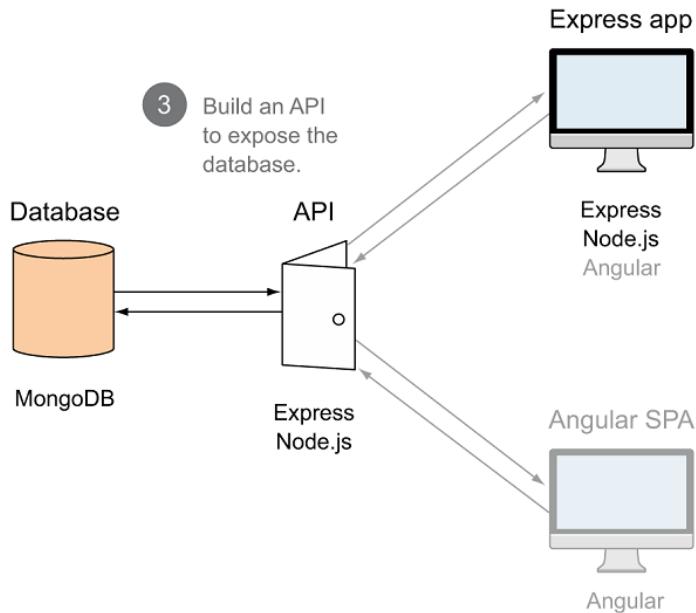
**Figure 2.16** After the static site is built we'll use the information gleaned to design the data model and create the MongoDB database.

In the MEAN stack, we'll use MongoDB for this step, relying heavily on Mongoose for the data modeling. The data models will actually be defined inside the Express application. This step will be covered in chapter 5.

### STEP 3: BUILD OUR REST API

When we've built the database and defined the data models we'll want to create a REST API so that we can interact with the data through making web calls. Pretty much any data-driven application will benefit from having an API interface, so this is another step you'll want to have in most build projects.

You can see where this step fits into building the overall project in figure 2.17.



**Figure 2.17 Use Express and Node.js to build an API, exposing methods of interacting with the database.**

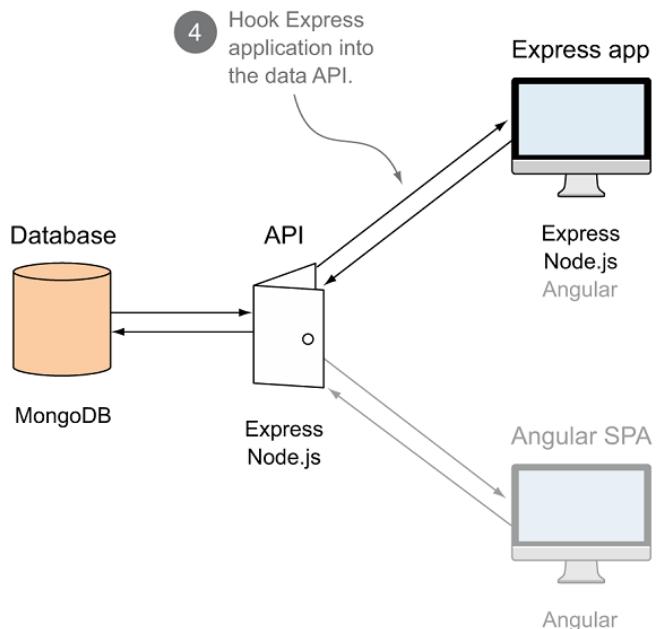
In the MEAN stack this step is mainly done in Node.js and Express, with quite a bit of help from Mongoose. We'll use Mongoose to interface with MongoDB rather than dealing with MongoDB directly. This step will be covered in chapter 6.

#### **STEP 4: USE THE API FROM OUR APPLICATION**

This step matches stage 4 of the development process and is where Loc8r will start to come to life. The static application from step 1 will be updated to use the REST API from step 3 to interact with the database created in step 2.

To learn about all parts of the stack, and the different ways in which we can use them, we'll be using Express and Node.js to make calls to the API. If, in a real-world scenario, you were planning to build the bulk of an application in Angular then you'd hook your API into Angular instead. We'll cover that in chapters 8, 9, and 10.

At the end of this step we'll have an application running on the first of the three architectures: an Express and Node.js application. Figure 2.18 shows how this step glues together the two sides of the architecture.



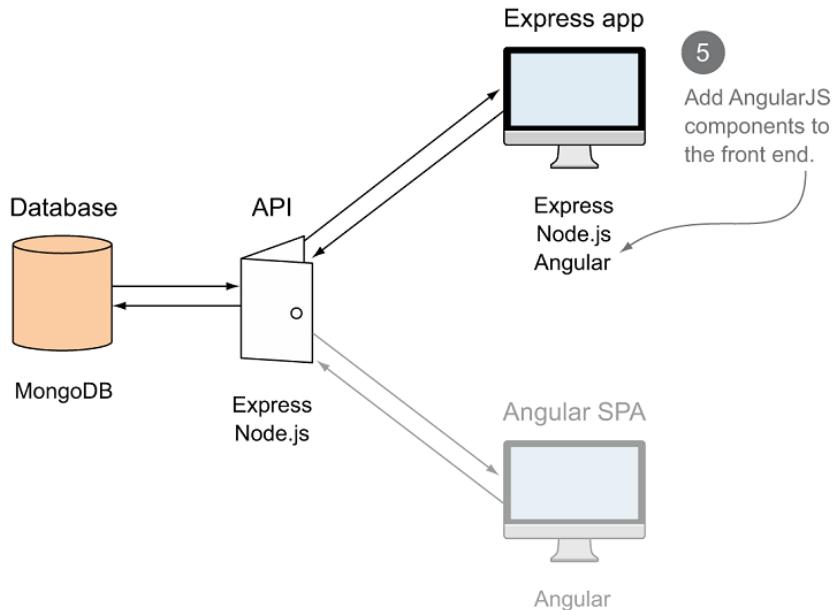
**Figure 2.18** Update the static Express application by hooking it into the data API, allowing the application to be database-driven.

In this build we'll be doing the majority of this step in Node.js and Express, and it will be covered in chapter 7.

#### **STEP 5: EMBELLISH THE APPLICATION**

Step 5 relates to stage 5 in the development process, where we get to add extra touches to the application. We're going to use this step to take a look at Angular, and we'll see how we can integrate Angular components into an Express application.

You can see this addition to the project architecture highlighted in figure 2.19.



**Figure 2.19** One way to use Angular in a MEAN application is to add components to the front end in an Express application.

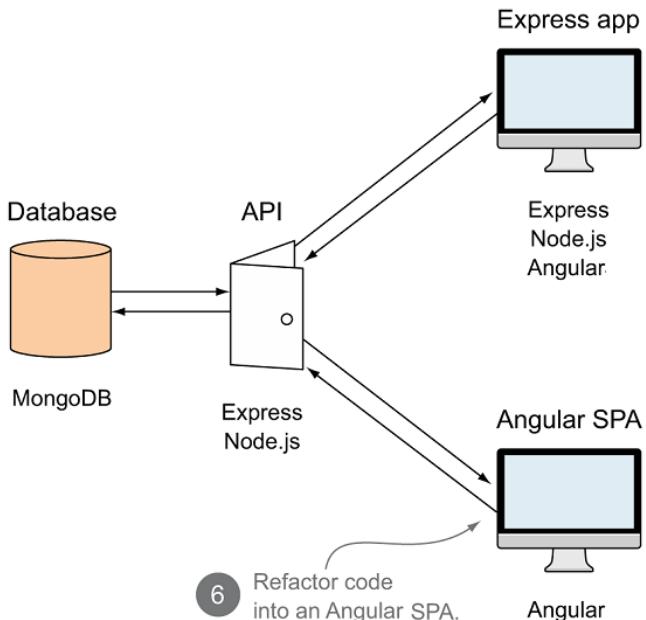
This step is all about introducing and using Angular. To support this, we'll most likely also change some of our Node.js and Express setup. This step will be covered in chapter 8.

#### STEP 6: REFACTOR THE CODE INTO AN ANGULAR SPA

In step 6 we'll radically change the architecture by replacing the Express application and moving all of the logic into an SPA using Angular. Unlike all of the previous steps, this replaces some of what has come before it, rather than building upon it.

This would be an unusual step in a normal build process, to develop an application in Express and then redo it in Angular, but it suits the learning approach in this book particularly well. We'll be able to focus on Angular as we already know what the application should do, and there's a data API ready for us.

Figure 2.20 shows how this change affects the overall architecture.



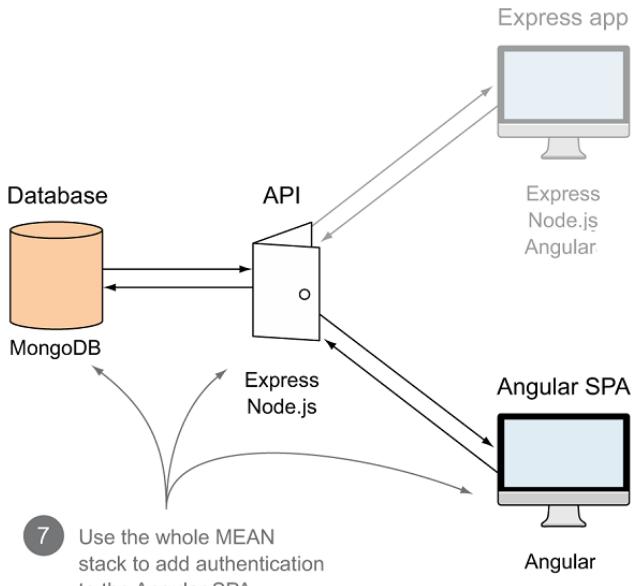
**Figure 2.20** Effectively rewriting the application as an Angular SPA.

This step is once again focused on Angular, and will be covered in chapters 9 and 10.

#### **STEP 7: ADD AUTHENTICATION**

In step 7 we'll add functionality to the application and enable users to register and log in, and we'll also see how to make use of the user's data whilst they are using the application. We'll build on everything we've done so far and add authentication to the Angular SPA. As a part of this we'll save user information in the database and secure certain API end points so that they can only be used by authenticated users.

Figure 2.21 shows what we'll be working with in the architecture.



7

Use the whole MEAN stack to add authentication to the Angular SPA.

**Figure 2.21** Using all of the MEAN stack to add authentication to the Angular SPA.

In this step, we'll work with all of the MEAN technologies; this is covered in chapter 11. That's the software architecture planned, let's have a quick chat about hardware.

## 2.6 Hardware architecture

No discussion about architecture would be complete without a section on hardware. You've seen how all of the software and code components can be put together, but what type of hardware do you need to run it all?

### 2.6.1 Development hardware

The good news is that you don't need anything particularly special to run a development stack. Just a single laptop, or even a virtual machine (VM), is enough to develop a MEAN application. All components of the stack can be installed on Windows, Mac OS X, and most Linux distributions.

I've successfully developed applications on Windows and Mac OS X laptops, and also on Ubuntu VMs. My personal preference is native development in OS X, but I know of others who swear by using Linux VMs.

If you have a local network and a number of different servers, you can run different parts of your application across them. For example, it's possible to have one machine as a database

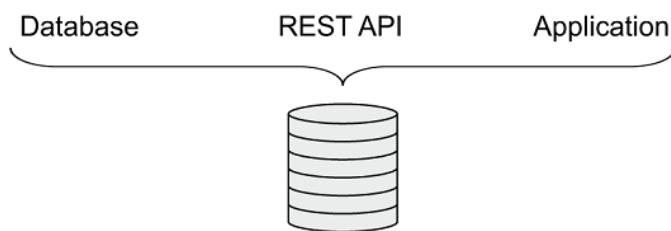
server, another for the REST API, and a third for the main application code itself. So long as the servers can talk to each other this isn't a problem.

### 2.6.2 Production hardware

The approach to production hardware architecture isn't all that different from development hardware. The main difference is that production hardware is normally higher spec, and is open to the internet to receive public requests.

#### STARTER SIZE

It's quite possible to have all parts of your application hosted and running on the same server. You can see a basic diagram of this in figure 2.22.

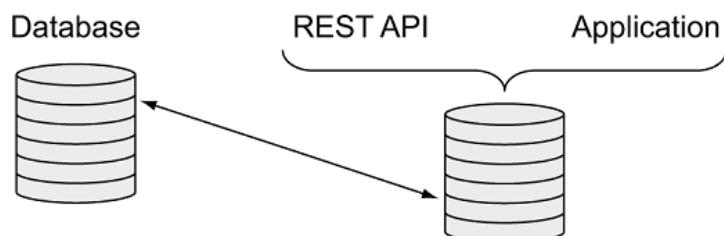


**Figure 2.22 The simplest of hardware architectures, having everything on a single server.**

This architecture is okay for applications with low traffic, but isn't generally advised as your application grows, because you don't want your application and database fighting over the same resources.

#### GROWING UP: A SEPARATE DATABASE SERVER

One of the first things to be moved onto a separate server is often the database. So now you have two servers: one for the application code and one for the database. Figure 2.23 illustrates this approach.



**Figure 2.23 A common hardware architecture approach: one server to run the application code and API, and a second, separate database server.**

This is quite a common model, particularly if you choose to use a platform as a service (PaaS) provider for your hosting. We'll be using this approach in this book.

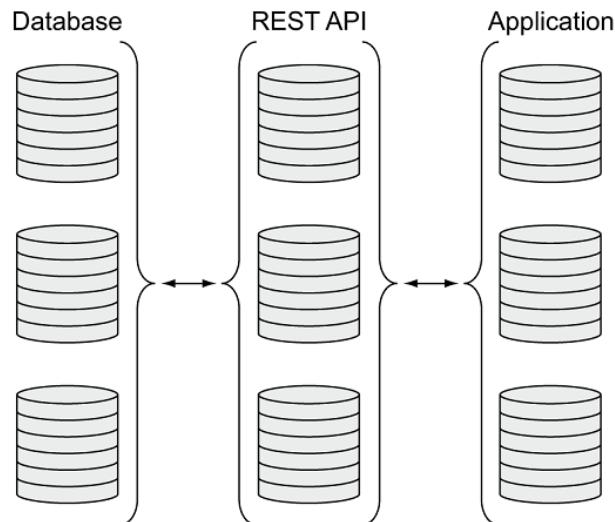
### GOING FOR SCALE

Much like we talked about in the development hardware, you can have a different server for the different parts of your application—a database server, an API server, and an application server. This will allow you to deal with more traffic as the load is spread across three servers, as illustrated in figure 2.24.



**Figure 2.24** A decoupled architecture using three servers: one for the database, one for the API, and one for the application code.

But it doesn't stop there. If your traffic starts to overload your three servers, you can have multiple instances—or clusters—of these servers, as shown in figure 2.25.



**Figure 2.25** You can scale MEAN applications by having clusters of servers for each part of your entire application.

Setting up this approach is a little more involved than the previous methods because you need to ensure that your database remains accurate, and that the load is balanced across the servers. Once again, PaaS providers offer a convenient route into this type of architecture.

## 2.7 Summary

In this chapter, you've learned the following:

- A common MEAN stack architecture with an Angular SPA using a REST API built in Node.js, Express, and Mongo
- Points to consider when deciding whether to build an SPA or not
- How to design a flexible architecture in the MEAN stack
- The best practice of building an API to expose a data layer
- The steps we're going to take to build the sample application Loc8r
- Development and production hardware architectures

We'll get started on the journey in chapter 3 by creating the Express project that will hold everything together.

# 3

## *Creating and setting up a MEAN project*

### This chapter covers

- Managing dependencies by using npm and a package.json file
- Creating and configuring Express projects
- Setting up an MVC environment
- Adding Twitter Bootstrap for layout
- Publishing to a live URL and using Git and Heroku

Now we're really ready to get underway, and in this chapter we'll get going on building our application. Remember from chapters 1 and 2 that throughout this book we're going to build an application called Loc8r. This is going to be a location-aware web application that will display listings near users and invite people to log in and leave reviews.

### Getting the source code

The source code for this application is on GitHub at [github.com/simonholmes/getting-MEAN](https://github.com/simonholmes/getting-MEAN). Each chapter with a significant update will have its own branch. I encourage you to build it up from scratch through the course of the book, but if you wish you can get the code we'll be building throughout this chapter from GitHub on the chapter-03 branch. In a fresh folder in terminal, the following two commands will clone it if you already have Git installed:

```
$ git clone -b chapter-03 https://github.com/simonholmes/getting-MEAN-2.git
```

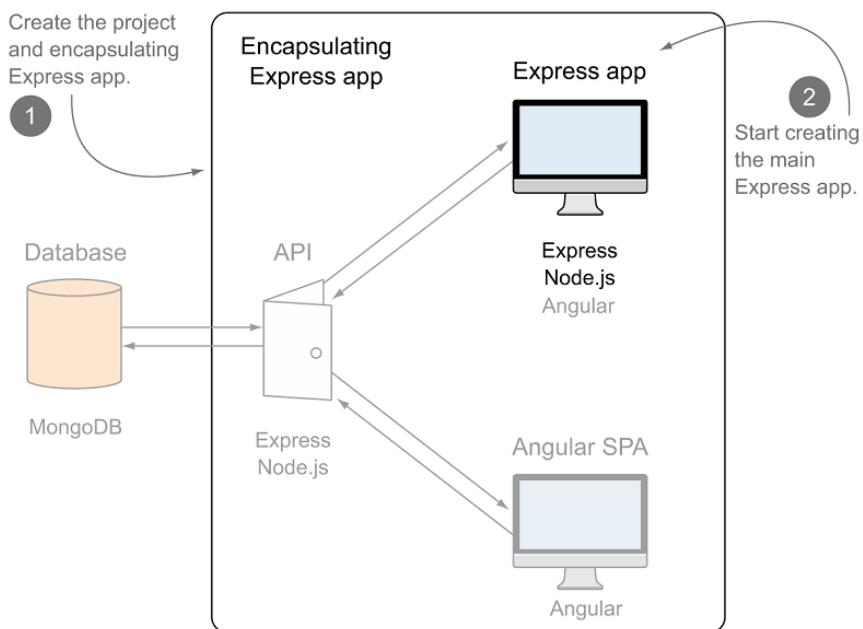
This will give you a copy of the code that's stored on GitHub. To run the application, you'll need to install some dependencies with the following commands:

```
$ cd getting-MEAN-2
$ npm install
```

Don't worry if some of this doesn't make sense just yet, or if some of the commands aren't working. During this chapter, we'll install these technologies as we go.

In the MEAN stack, Express is the Node web application framework. Together Node and Express underpin the entire stack, so let's start here. In terms of building up the application architecture, figure 3.1 shows where we'll be focusing in this chapter. We'll be doing two things:

1. Creating the project and encapsulating the Express application that will house everything else except the database
2. Setting up the main Express application



**Figure 3.1** Creating the encapsulating Express application, and starting to set up the main Express application

We'll start with a bit of groundwork by looking at Express and seeing how we can manage dependencies and modules using npm and a package.json file. We'll need this background knowledge to get going and set up an Express project.

Before we can really do anything, we'll make sure that you have everything you need installed on your machine. When that's all done, we'll look at creating new Express projects from the command line and the various options we can specify at this point.

Express is great, but you can make it better—and get to know it better—by tinkering a little and changing some things around. This ties into a quick look at model-view-controller (MVC) architecture. Here's where we'll get under the hood of Express a little, and see what it's doing by modifying it to have a very clear MVC setup.

When the framework of Express is set up as we want it, we'll next include Twitter's Bootstrap framework and then make the site responsive by updating the Jade templates. In the final step of this chapter we'll push the modified, responsive MVC Express application to a live URL using Heroku and Git.

## 3.1 A brief look at Express, Node, and npm

As mentioned before, Express is a web application framework for Node. In basic terms, an Express application is simply a Node application that happens to use Express as the framework. Remember from chapter 1 that npm is a package manager that gets installed when you install Node, which gives you the ability to download Node modules or packages to extend the functionality of your application.

But how do these things work together, and how do you use them? A key piece to understanding this puzzle is the `package.json` file.

### 3.1.1 Defining packages with `package.json`

In every Node application, there should be a file in the root folder of the application called `package.json`. This file can contain various metadata about a project, including the packages that it depends on to run. The following listing shows an example `package.json` file that you might find in the root of an Express project.

**Listing 3.1 Example `package.json` file in a new Express project**

```
{
  "name": "application-name",          1
  "version": "0.0.0",                  1
  "private": true,                    1
  "scripts": {                        1
    "start": "node ./bin/www"        1
  },                                  1
  "dependencies": {                  2
    "body-parser": "~1.15.2",         2
    "cookie-parser": "~1.4.3",       2
    "debug": "~2.2.0",               2
    "express": "^4.14.0",            2
    "morgan": "^1.7.0",              2
    "pug": "^2.0.0-beta6" ,          2
    "serve-favicon": "~2.3.0"        2
  }                                   2
}
```

- ① Various metadata defining application
- ② Package dependencies needed for application to run

This is the file in its entirety, so it's not particularly complex. There's various metadata at the top of the file followed by the dependencies section. In this default installation of an Express project, there are quite a few dependencies that are required for Express to run, but we don't need to worry about what each one does. Express itself is modular so that you can add in components or upgrade them individually.

### **WORKING WITH DEPENDENCY VERSIONS IN PACKAGE.JSON**

Alongside the name of each dependency is the version number that the application is going to use. Notice that they're all prefixed with a ~.

Let's take a look at the dependency definition for Express 4.14.0. It specifies a particular version at three levels:

- Major version (4)
- Minor version (14)
- Patch version (0)

Prefixing the whole version number with a ~ is like replacing the patch version with a wildcard, which means that the application will use the latest patch version available. Similarly, prefixing the version with a ^ is like replacing the minor version with a wildcard. This has become best practice, because patches and minor version should only contain fixes that won't have any impact on the application. But different major versions are released when a breaking change is made, so you want to avoid automatically using later versions of these in case the breaking change affects your application. However, if you find a module that breaks these rules it is easy to specify an exact version to use by removing any prefixes. Note that it's a good practice to always specify the full version and not use wildcards for this reason - you always have a reference for a specific version that you know works.

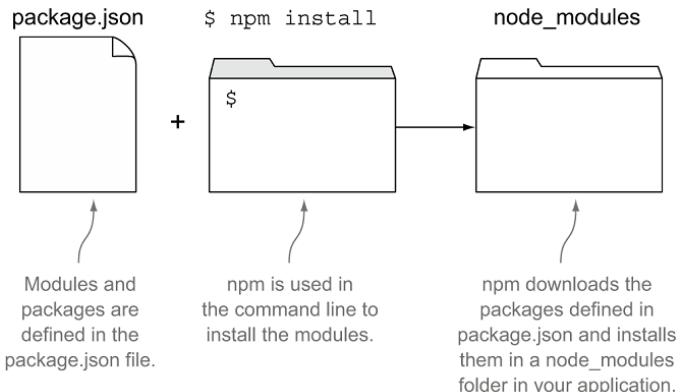
#### **3.1.2 Installing Node dependencies with npm**

Any Node application or module can have dependencies defined in a package.json file. Installing them is really easy and is done in the same way regardless of the application or module.

Using a terminal prompt in the same folder as the package.json file, you simply need to run the following command:

```
$ npm install
```

This tells npm to install all of the dependencies listed in the package.json file. When you run it, npm will download all of the packages listed as dependencies and install them into a specific folder in the application called *node\_modules*. Figure 3.2 illustrates the three key parts.



**Figure 3.2** The npm modules defined in a `package.json` file are downloaded and installed into the application's `node_modules` folder when you run the `npm install` terminal command.

npm will install each package into its own subfolder because each one is effectively a Node package in its own right. As such, each package also has its own `package.json` file defining the metadata, including the specific dependencies. It's quite common for a package to have its own `node_modules` folder. You don't need to worry about manually installing all of the nested dependencies though, because this is all handled by the original `npm install` command.

### ADDING MORE PACKAGES TO AN EXISTING PROJECT

You're unlikely to have the full list of dependencies for a project right from the outset. It's far more likely that you'll start off with a few key ones that you know you'll need, and perhaps some that you always use in your workflow.

Using npm, it's really easy to add more packages to the application whenever you want. You simply find the name of the package you want to install and open a command prompt in the same folder as the `package.json` file. You then run a simple command like this:

```
$ npm install --save package-name
```

With this command, npm will download and install the new package into the `node_modules` folder. The `--save`, flag tells npm to add this package to the list of dependencies in the `package.json` file.

### UPDATING PACKAGES TO LATER VERSIONS

The only time npm downloads and reinstalls existing packages is when you're upgrading to a new version. When you run `npm install`, npm will go through all of the dependencies and check the following:

- The version defined in the `package.json` file

- The latest matching version on npm (which might be different if you've used ~ or ^)
- The version installed in the node\_modules folder (if at all)

If your installed version is different from the definition in the package.json file, npm will download and install the version defined in package.json. Similarly, if you're using a wildcard and there's a later matching version available, npm will download and install it in place of the previous version.

With that knowledge under your belt, we can start creating our first Express project.

## 3.2 Creating an Express project

All journeys must have a starting point, which for building a MEAN application is to create a new Express project. To create an Express project, you'll need to have five key things installed on your development machine:

- Node and npm
- The Express generator installed globally
- Git
- Heroku
- Suitable command-line interface (CLI) or terminal

### 3.2.1 Installing the pieces

If you don't have Node, npm, or the Express generator installed yet, see appendix A for instructions and pointers to online resources. They can all be installed on Windows, Mac OS X, and most mainstream Linux distributions.

By the end of this chapter, we'll also have used Git to manage the source control of our Loc8r application, and pushed it to a live URL using Heroku. Please take a look through appendix B, which guides you through setting up Git and Heroku.

Depending on your operating system, you may need to install a new CLI or terminal. See appendix B to find out if this applies to you or not.

**NOTE** Throughout this book I'll often refer to the CLI as *terminal*. So when I say "run this command in terminal," simply run it in whichever CLI you're using. When terminal commands are included as code snippets throughout this book, they'll start with a \$. You shouldn't type this into terminal; it's simply there to denote that this is a command-line statement. For example, using the echo command \$ echo 'Welcome to Getting MEAN', you'd just type in echo 'Welcome to Getting MEAN'.

### VERIFYING THE INSTALLATIONS

To create a new Express project, you must have Node and npm installed and also have the Express generator installed globally. You can verify this by checking for the version numbers in the terminal using the following commands:

```
$ node --version
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>

```
$ npm --version
$ express --version
```

Each of these commands should output a version number to terminal. If one of them fails, head back to appendix A on how to install it again.

### 3.2.2 Creating a project folder

Assuming all is good, let's start by creating a new folder on your machine called *loc8r*. This can be on your desktop, in your documents, in a Dropbox folder—it doesn't really matter, as long as you have full read and write access rights to the folder.

I personally do a lot of my MEAN development in Dropbox folders so that it's immediately backed up and accessible on any of my machines. If you're in a corporate environment, this may not be suitable for you, so create the folder wherever you think best.

### 3.2.3 Configuring an Express installation

An Express project is installed from the command line, and the configuration is passed in using parameters on the command that you use. If you're not familiar with using the command line, don't worry, none of what we'll go through in the book is particularly complex, and it's all pretty easy to remember. When you've started using it you'll probably start to love how it makes some operations so fast!

For example—don't do this just yet—you can install Express into a folder with a simple command:

```
$ express
```

This would install the framework with default settings into your current folder. This is probably a good start, but let's take a look at some configuration options first.

#### **CONFIGURATION OPTIONS WHEN CREATING AN EXPRESS PROJECT**

What can you configure when creating an Express project? When creating an Express project in this way, you can specify the following:

- Which HTML template engine to use
- Which CSS preprocessor to use
- Whether to create a *.gitignore* file

A default installation will use the Jade template engine, but it will have no CSS preprocessing or session support. You can specify a few different options as laid out in table 3.1.

**Table 3.1 Command-line configuration options when creating a new Express project**

Configuration command	Effect
--css=less stylus	Adds a CSS preprocessor to your project, either Less or Stylus, depending on which you type in the command.
--view=ejs hbs pug	Changes the HTML template engine from Jade to EJS, handlebars or pug, depending on which you type.
--git	Adds a .gitignore file to the directory

For example—and this isn’t what we’re going to do here—if you want to create a project that uses the Less CSS preprocessor, the Handlebars template engine and includes a .gitignore file you’d run the following command in terminal:

```
$ express --css=less --view=hbs --git
```

To keep things simple in our project we won’t use CSS preprocessing, so we can stick with the default of plain CSS. But we do need to use a template engine, so let’s take a quick look at the options.

### **DIFFERENT TEMPLATE ENGINES**

When using Express in this way there are a few template options available, including Jade, EJS, Handlebars, and Pug. The basic workflow of a template engine is that you create the HTML template, including placeholders for data, and then pass it some data. The engine will then compile the two together to create the final HTML markup that the browser will receive.

All of the engines have their own merits and quirks, and if you already have a preferred one then that’s fine. In this book, we’re going to be using Pug. Pug is very powerful and provides all of the functionality we’re going to need. Pug is the next evolution of Jade: due to trademark issues, the creators of Jade had to rename it and they chose *Pug*. Jade still exists so that existing projects don’t break, but all new releases will be under the name Pug. Jade was – and still is at the time of writing – the default template engine in Express, so you’ll find that most examples and projects online use it. This means it’s very helpful to be familiar with the syntax. Finally, the minimal style of Jade and Pug make them ideal for code samples in a book!

### **A QUICK LOOK AT PUG**

Pug is unusual when compared to the other template engines in that it doesn’t actually contain HTML tags in the templates. Instead, Pug takes a rather minimalist approach, using tag names, indentation, and a CSS-inspired reference method to define the structure of the HTML. The exception to this is the `<div>` tag. Because it’s so common, if the tag name is omitted from the template, Pug will assume that you want a `<div>`.

**TIP** Pug templates must be indented using spaces, not tabs.

The following code snippet shows a simple example of a Pug template:

```
#banner.page-header
  h1 My page
  p.lead Welcome to my page
```

1  
1  
1

And this snippet show the compiled output:

```
<div id="banner" class="page-header">
  <h1>My page</h1>
  <p class="lead">Welcome to my page</p>
</div>
```

2  
2  
2  
2

- ① Pug template contains no HTML tags
- ② Compiled output is recognizable HTML

From the first lines of the input and output you should be able to see that

- With no tag name specified, a `<div>` is created.
- `#banner` in Pug becomes `id="banner"` in HTML.
- `.page-header` in Pug becomes `class="page-header"` in HTML.

Note also that the indentation in Pug is very important as that is what defines the nesting of the HTML output - remember that the indentation must be done using spaces not tabs!

So, to recap, we don't need a CSS pre-processor, but we do want the Pug template engine. How about the `.gitignore` file?

### A QUICK INTRO TO `.GITIGNORE` FILES

A `.gitignore` file is a simple configuration file that sits in the root of your project folder. This file specifies which files and folders Git commands should ignore; in essence it says "pretend these files don't exists and don't track them" meaning that they won't end up in source control.

Common examples include log files and the `node_modules` folder. Log files don't need to be up on GitHub for everyone to see, and your Node dependencies should be installed from npm whenever your application downloaded. We will be using Git later in section 3.5, so we'll ask the Express generator to create a file for us.

So with that starting knowledge behind us, it's time for us to create a project.

#### 3.2.4 Creating an Express project and trying it out

So we know the basic command for creating an Express project, and have decided to use the Pug template engine. We'll also let it generate a `.gitignore` file for us, so let's go ahead and create a new project. In section 3.2.2 you should have created a new folder called `loc8r`. Navigate to this folder in your terminal, and run the following command:

```
$ express --view=pug --git
```

This will create a bunch of folders and files inside the loc8r folder that will form the basis of our Loc8r application. But we're not quite ready yet. Next you'll need to install the dependencies. As you may remember, this is simply done by running the following command from a terminal prompt in the same folder as the package.json file:

```
$ npm install
```

As soon as you run it you'll see your terminal window light up with all of the things it's downloading. When it has finished, the application is ready for a test drive.

### **TRYING IT OUT**

Running the application is a piece of cake. We'll take a look at a better way of doing this in just a moment, but if you're impatient like me, you'll want to see that what you've done so far works.

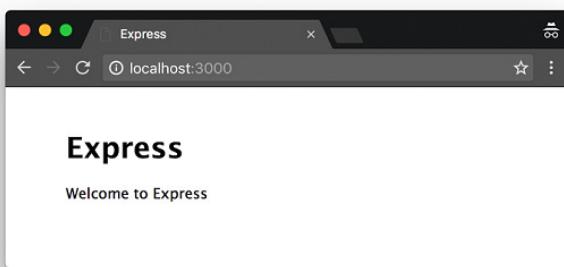
In terminal, in the loc8r folder, run the following command (if your application is in a different folder name, swap out loc8r accordingly):

```
$ DEBUG=loc8r:* npm start
```

You should see a confirmation similar to this:

```
loc8r:server Listening on port 3000 +0ms
```

This means that the Express application is running! You can see it in action by opening a browser and heading over to localhost:3000. Hopefully you'll see something like the screenshot in figure 3.3.



**Figure 3.3** Landing page for a barebones Express project

Admittedly, this is not exactly ground-breaking stuff right now, but getting the Express application up and running to the point of working in a browser was pretty easy, right?

If you head back to terminal now you should see a couple of log statements confirming that the page has been requested, and that a stylesheet has been requested. To get to know Express a little better, let's take a look at what's going on here.

### **HOW EXPRESS HANDLES THE REQUESTS**

The default Express landing page is pretty simple. There's a small amount of HTML, of which some of the text content is pushed as data by the Express route. There's also a CSS file. The logs in terminal should confirm that this is what Express has had requested and has returned to the browser. But how does it do it?

---

#### **About Express middleware**

In the middle of the `app.js` file there are a bunch of lines that start with `app.use`. These are known as *middleware*. When a request comes in to the application it passes through each piece of middleware in turn. Each piece of middleware may or may not do something with the request, but it's always passed on to the next one until it reaches the application logic itself, which returns a response.

Take `app.use(express.cookieParser())`; for example. This will take an incoming request, parse out any of the cookie information, and then attach the data to the request in a way that makes it easy to reference it in the controller code.

You don't really need to know what each piece does right now, but you may well find yourself adding to this list as you build out applications.

---

All requests to the Express server run through the middleware defined in the `app.js` file (see the sidebar "About Express middleware"). As well as doing other things, there's a default piece of middleware that looks for paths to static files. When the middleware matches the path against a file, Express will return this asynchronously, ensuring that the Node.js process isn't tied up with this operation and therefore blocking other operations. When a request runs through all of the middleware, Express will then attempt to match the path of the request against a defined route. We'll get into this in a bit more detail later in this chapter.

Figure 3.4 illustrates this flow, using the example of the default Express homepage from figure 3.3.

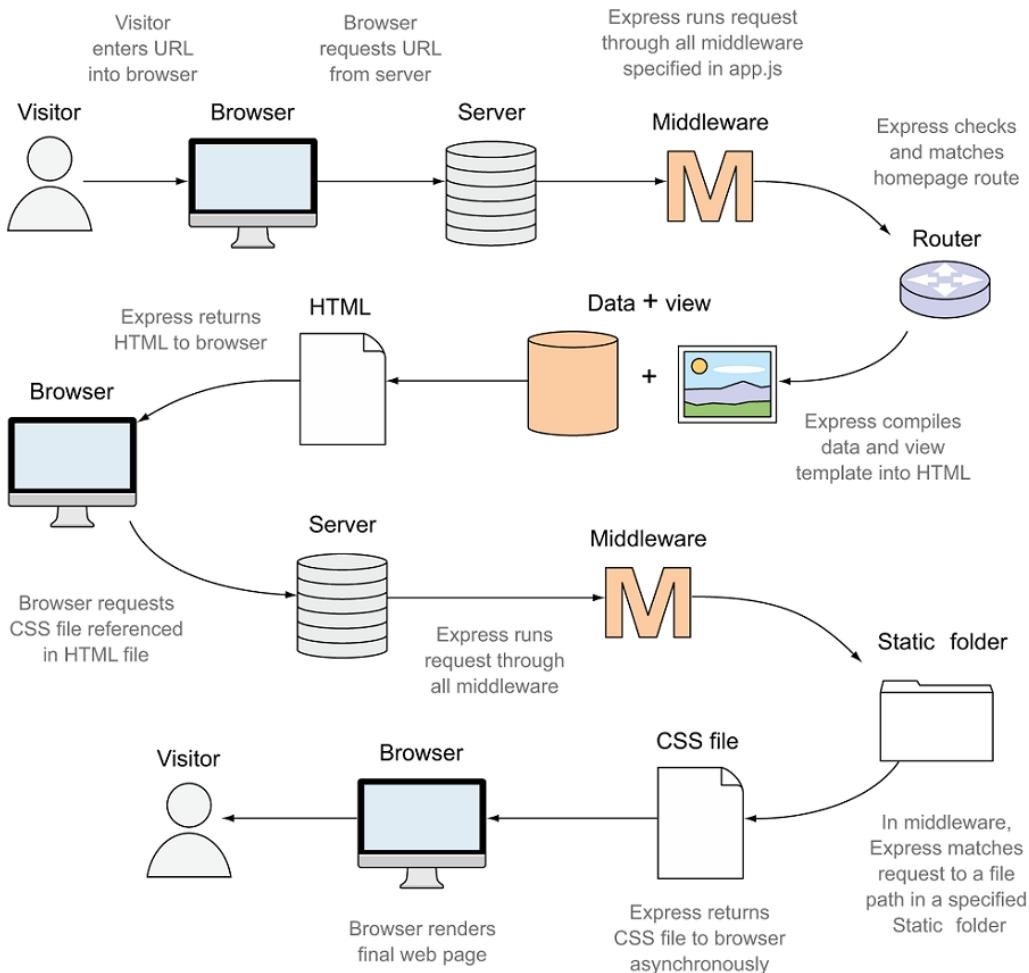


Figure 3.4 The key interactions and processes that Express goes through when responding to the request for the default landing page. The HTML page is processed by Node to compile data and a view template, and the CSS file is served asynchronously from a static folder.

The flow in figure 3.4 shows the separate requests made and how Express handles them differently. Both requests run through the middleware as a first action, but the outcomes are very different.

### 3.2.5 Restarting the application

A Node application compiles before running, so if you make changes to the application code while it's running, they won't be picked up until the Node process is stopped and restarted.

Note that this is only true for application code; Jade templates, CSS files, and client-side JavaScript can all be updated on-the-fly.

Restarting the Node process is a two-step procedure. First you have to stop the running process. You do this in terminal by pressing Ctrl-C. Then you have to start the process again in terminal using the same command as before: `DEBUG=loc8r:* npm start`.

This doesn't sound problematic, but when you're actively developing and testing an application, having to do these two steps every time you want to check an update actually becomes quite frustrating. Fortunately there's a better way.

#### **AUTOMATICALLY RESTARTING THE APPLICATION WITH NODEMON**

There are some services out there that have been developed to monitor application code that will restart the process when it detects that changes have been made. One such service, and the one we'll be using in this book, is `nodemon`. `nodemon` simply wraps the Node application, and other than monitoring for changes causes no interference.

To use `nodemon` you start off by installing it globally, much like you did with Express. This is done using `npm` in terminal:

```
$ npm install -g nodemon
```

When the installation has finished, you'll be able to use `nodemon` wherever you wish. Using it's really simple. Instead of typing `node` to start the application, you type `nodemon`. So, making sure you're in the `loc8r` folder in terminal—and that you've stopped the Node process if it's still running—enter the following command:

```
$ nodemon
```

You should see that a few extra lines are output to terminal confirming that `nodemon` is running, and that it has started `node ./bin/www`. If you head back over to your browser and refresh, you should see that the application is still there.

**NOTE** `nodemon` is only intended for easing the development process in your development environment, and shouldn't really be used in a live production environment.

### **3.3 Modifying Express for MVC**

First off, what is MVC architecture? MVC stands for model-view-controller, and it's an architecture that aims to separate out the data (model), the display (view), and the application logic (controller). This separation aims to remove any tight coupling between the components, theoretically making code more maintainable and reusable. A bonus is that these components fit very nicely into our rapid prototype development approach and allow us to concentrate on one aspect at a time as we discuss each part of the MEAN stack.

There are whole books dedicated to the nuances of MVC, but we're not going to go into that depth here. We'll keep the discussion of MVC at a high level, and see how we can use it with Express to build our Loc8r application.

### 3.3.1 A bird's eye view of MVC

Most applications or sites that you build will be designed to take an incoming request, do something with it, and return a response. At a simple level, this loop in an MVC architecture works like this:

1. A request comes into the application.
2. The request gets routed to a controller.
3. The controller, if necessary, makes a request to the model.
4. The model responds to the controller.
5. The controller sends a response to a view.
6. The view sends a response to the original requester.

In reality, depending on your setup, the controller may actually compile the view before sending the response back to the visitor. The effect is the same though, so keep this simple flow in mind as a visual for what will happen in our Loc8r application. See figure 3.5 for an illustration of this loop.

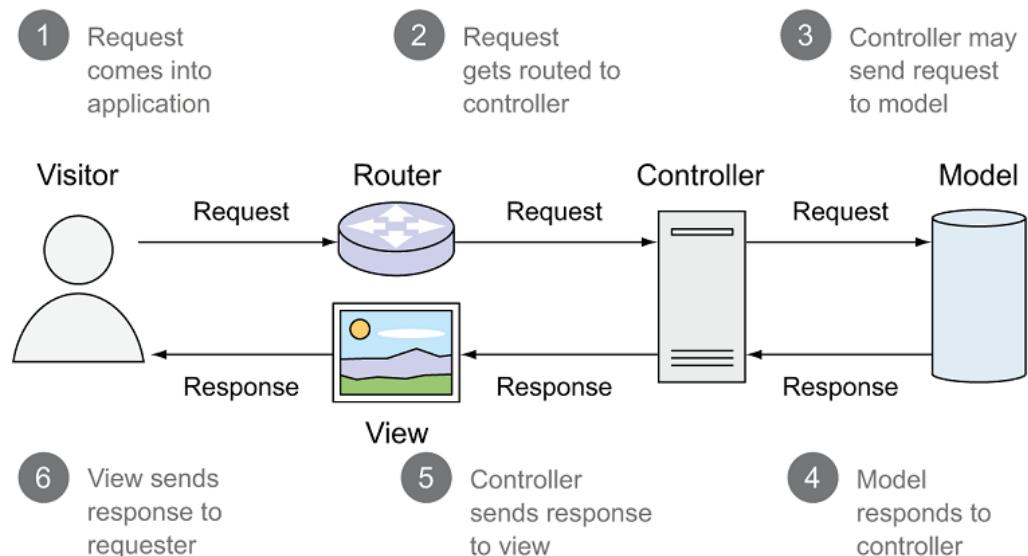


Figure 3.5 Request-response flow of a basic MVC architecture

Figure 3.5 highlights the individual parts of the MVC architecture and how they link together. It also illustrates the need for a routing mechanism along with the model, view, and controller components. So now that you've seen how we want the basic flow of our Loc8r application to work, it's time to modify the Express setup to make this happen.

### 3.3.2 Changing the folder structure

If you look inside the newly created Express project in the loc8r folder, you should see a file structure including a views folder, and even a routes folder, but no mention of models or controllers. Rather than going ahead and cluttering up the root level of the application with some new folders, let's keep things tidy with one new folder for all of our MVC architecture. Follow three quick steps here:

1. Create a new folder called app\_server.
2. In app\_server create two new folders called models and controllers.
3. Move the views and routes folders from the root of the application into the app\_server folder.

Figure 3.6 illustrates these changes and shows the folder structures before and after the modifications.

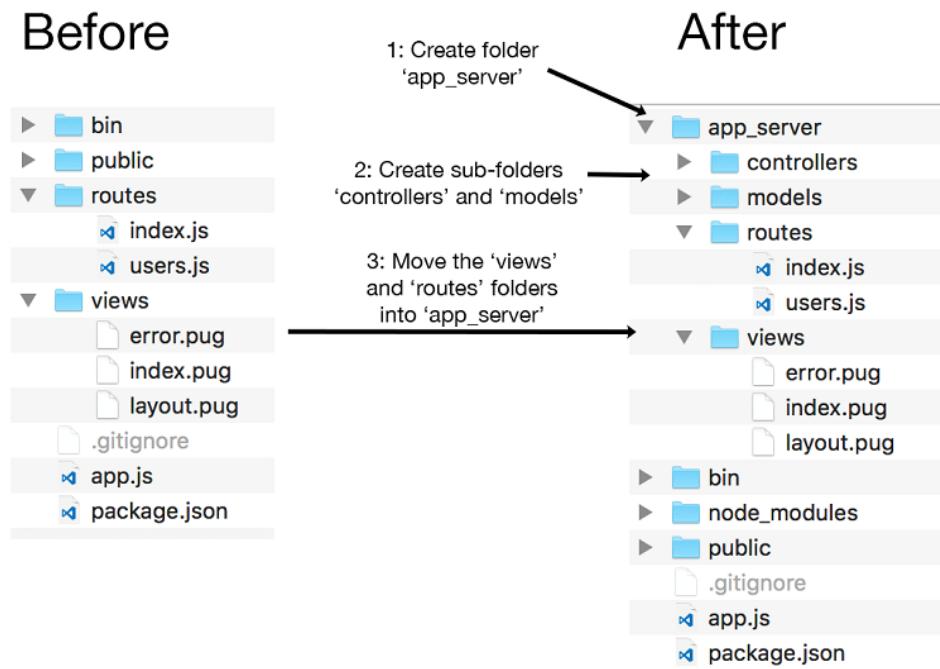


Figure 3.6 Changing the folder structure of an Express project into an MVC architecture

Now we have a really obvious MVC setup in the application, which makes it easier to separate our concerns. But if you try to run the application now it won't work, as we've just broken it. So let's fix it. Express doesn't know that we've added in some new folders, or have any idea what we want to use them for. So we need to tell it.

### 3.3.3 Using the new views and routes folders

The first thing we need to do is tell Express that we've moved the views and routes folders, because Express will be looking for folders and files that no longer exist.

#### **USING THE NEW VIEWS FOLDER LOCATION**

Express will be looking in /views but we've just moved it to /app\_server/views. Changing it's really simple. In app.js find the following line:

```
app.set('views', path.join(__dirname, 'views'));
```

and change it to the following (modifications in bold):

```
app.set('views', path.join(__dirname, 'app_server', 'views'));
```

Our application still won't work just yet because we've moved the routes, so let's tell Express about them too.

#### **USING THE NEW ROUTES FOLDER LOCATION**

Express will be looking in /routes but we've just moved it to /app\_server/routes. Changing this is also really simple. In app.js find the following lines:

```
var index = require('./routes/index');
var users = require('./routes/users');
```

and change them to the following (modifications in bold):

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

#### **Defining variables in ES6**

One of the most fundamental changes in ES6 is deprecating the `var` keyword to define variables. It still works, but instead you should use one of the two new keywords `const` or `let`. Variables defined with `const` *cannot* be changed at a later point in the code, whereas variables defined with `let` *can* be changed.

Best practice is to always define variables with `const` unless the value is going to change. All instances of `var` in app.js can be changed to `const` – I've done this in the source code for this book, feel free to do it too.

Note that we've also changed var to const to upgrade to ES6, check out the sidebar "Defining variables in ES6" if this is new to you. If you save this and run the application again, you'll find that we've fixed it and the application works once more!

### 3.3.4 Splitting controllers from routes

In a default Express setup, controllers are very much part of the routes, but we want to separate them out. Controllers should manage the application logic, and routing should map URL requests to controllers.

#### **UNDERSTANDING ROUTE DEFINITION**

To understand how routes work let's take a look at the route already set up for delivering the default Express homepage. Inside index.js in app\_server/routes you should see the following code snippet:

```
/* GET home page. */
router.get('/', function(req, res) {           ①
  res.render('index', { title: 'Express' });      ②
});
```

- ① Where router looks for URL
- ② Controller content, albeit very basic right now

In the code at #1 you can see `router.get('/')`, which is where the router looks for a `get` request on the homepage URL path, which is just `'/'`. The anonymous function that runs the code #2 is really the controller. This is a very basic example with no application code to speak of. So #1 and #2 are the pieces we want to separate here.

Rather than diving straight in and putting the controller code into the controllers folder, we'll test out the approach in the same file first. To do this, we can take the anonymous function from the route definition and define it as a named function. We'll then pass the name of this function through as the callback in the route definition. Both of these steps are in the following listing, which you can put into place inside app\_server/routes/index.js.

#### **Listing 3.2 Taking the controller code out of the route: step 1**

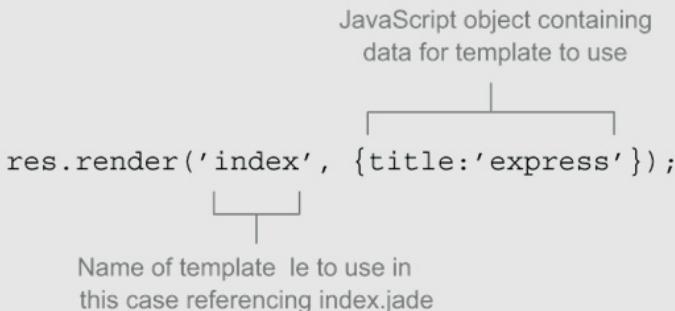
```
const homepageController = function (req, res) {    ①
  res.render('index', { title: 'Express' });          ①
};
/* GET home page. */
router.get('/', homepageController);                ②
```

- ① Take anonymous function and define it as a named function
- ② Pass name of function through as a callback in route definition

If you refresh your homepage now it should still work just as before. We haven't changed anything in how the site works, just moved a step toward separating concerns.

### **Understanding `res.render`**

We'll look at this more in chapter 4, but `render` is the Express function for compiling a view template to send as the HTML response that the browser will receive. The `render` method takes the name of the view template and a JavaScript data object in the following construct:



Note that the template file doesn't need to have the file extension suffix, so `index.pug` can just be referenced as `index`. You also don't need to specify the path to the view folder because you've already done this in the main Express setup.

Now that we're clear about how the route definition works, it's time to put the controller code into its proper place.

### **MOVING THE CONTROLLER OUT OF THE ROUTES FILE**

In Node, to reference code in an external file you create a module in your new file and then `require` it in the original file. See the following sidebar for some overarching principles behind this process.

### **Creating and using Node modules**

Taking some code out of a Node file to create an external module is fortunately pretty simple. In essence, you create a new file for your code, choose which bits of it you want to expose to the original file, and then `require` your new file in your original file.

In your new module file, you expose the parts of the code that you wish to by using the `module.exports` method, like so:

```
module.exports = function () {
  console.log("This is exposed to the requester");
};
```

You'd then `require` this in your main file like so:

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

If you want your module to have separate named methods exposed, then you can do so by defining them in your new file in the following way:

```
module.exports.logThis = function(message){
  console.log(message);
};
```

Or even better is to define a named function and export it at the end of the file. This lets you expose all the functions you need to in one place, creating a handy list for your future self (or subsequent developers).

```
const logThis = function(message) {
  console.log(message);
};

module.exports = {
  logThis
};
```

To reference this in your original file you need to assign your module to a variable name, and then invoke the method. For example, in your main file

```
const yourModule = require('./yourModule');
yourModule.logThis("Hooray, it works!");
```

This assigns your new module to the variable `yourModule`. The exported function `logThis` is now available as a method of `yourModule`.

When using the `require` function, note that you don't need to specify a file extension. The `require` function will look for a couple of things: a JavaScript file of the same name or an `index.js` file inside a folder of the given name.

So the first thing we need to do is create a file to hold the controller code. Create a new file called `main.js` in `app_server/controllers`. In this file we'll create and export a method called `index` and use it to house the `res.render` code as shown in the following listing.

### **Listing 3.3 Setting up the homepage controller in app\_server/controllers/main.js**

```
/* GET home page */
const index = function(req, res){          ①
  res.render('index', { title: 'Express' });  ②
};
module.exports = {                         ③
  index                                     ③
};
```

- ① Create an index function
- ② Include controller code for homepage

### ③ Expose the index function as a method

That's all there is to creating the controller export. The next step is to `require` this controller module in the routes file so that we can use the exposed method in the route definition. The following listing shows how the file `index.js` in `app_server/routes` should now look.

#### **Listing 3.4 Updating the routes file to use external controllers**

```
const express = require('express');
const router = express.Router();
const ctrlMain = require('../controllers/main'); ①
/* GET home page. */
router.get('/', ctrlMain.index);
module.exports = router;
```

① Require main controllers file

② Reference index method of controllers in route definition

This links the route to the new controller by "requiring" the controller file ① and referencing the controller function in the second parameter of the `router.get` function ②.

We now have the routing and controller architecture, as illustrated in figure 3.7, where `app.js` requires `routes/index.js`, which in turn requires `controllers/main.js`.

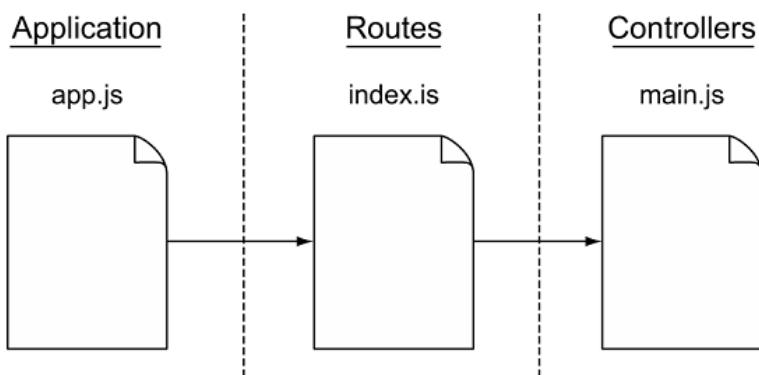


Figure 3.7 Separating the controller logic from the route definitions

If you test this out now in your browser, you should see that the default Express home-page displays correctly once again.

That's everything set up with Express for now, so it's almost time to start the building process. But before that there are a couple more things that we need to do, first of which is adding Twitter Bootstrap to the application.

## 3.4 Import Bootstrap for quick, responsive layouts

As discussed in chapter 1, our Loc8r application will make use of Twitter's Bootstrap framework to speed up the development of a responsive design. We'll also make the application stand out by adding some font icons and custom styles. The aim here is to help us keep moving forward quickly with building the application, and not get side-tracked with the semantics of developing a responsive interface.

### 3.4.1 Download Bootstrap and add it to the application

Instructions for downloading Bootstrap, downloading the font-icons (by Font Awesome), creating a custom style, and adding the files into the project folder are all found in appendix B. Note that we are using Bootstrap 4. A key point here's that the downloaded files are all static files to be sent directly to the browser; they don't need any processing by the Node engine. Your Express application will already have a folder for this purpose: the *public* folder. When you have it ready, the public folder should look something like figure 3.8.

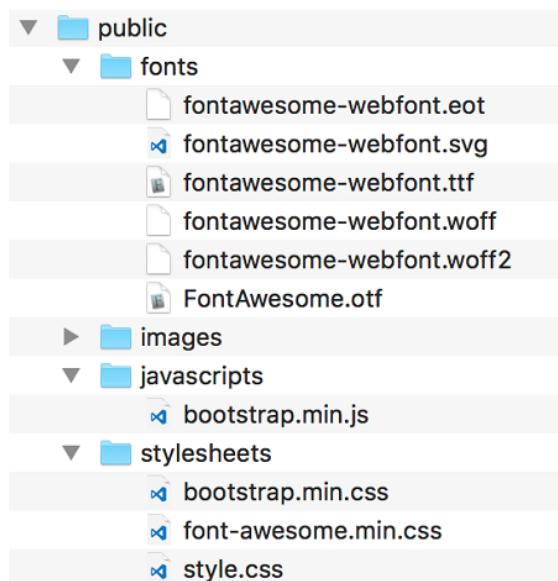


Figure 3.8 Structure of the public folder in the Express application after adding Bootstrap

Bootstrap also requires jQuery and Tether for some of the interactive components to work. Because these aren't core to our application, we'll reference these from a content delivery network (CDN) in the next step.

### 3.4.2 Using Bootstrap in the application

Now that all of the Bootstrap pieces are sitting in the application, it's time to hook it up to the front end. This means taking a look at the Pug templates.

#### **WORKING WITH PUG TEMPLATES**

Pug templates are often set up to work by having a main layout file that has defined areas for other Pug files to extend. This makes a great deal of sense when building a web application, because there are often many screens or pages that have the same underlying structure with different content in the middle.

This is how Pug appears in a default Express installation. If you look in the views folder in the application, you'll see three files, layout.pug, index.pug and error.pug. The index.pug file is controlling the content for the index page of the application. Open it up, and there's not much in there; the entire contents are shown in the following listing.

#### **Listing 3.5 The complete index.pug file**

```
extends layout
block content
  h1= title
  p Welcome to #{title.}
```

- ① Informs that this file is extending layout file
- ② Informs that the following section goes into area of layout file called content
- ③ Outputs h1 and p tags to content area

There's more going on here than meets the eye. Right at the top of the file is a statement declaring that this file is an extension of another file ①, in this case the layout file. Following this is a statement defining a block of code ② that belongs to a specific area of the layout file, an area called `content` in this instance. Finally, there's the minimal content that's displayed on the Express index page, a single `<h1>` tag and a single `<p>` tag ③.

There are no references to `<head>` or `<body>` tags here, nor any stylesheet references. These are all handled in the layout file, so that's more likely where you want to go to add in global scripts and stylesheets to the application. Open up layout.pug and you should see something similar to the following listing.

#### **Listing 3.6 Default layout.pug file**

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

①

- ① Empty named block can be used by other templates

Listing 3.6 shows the layout file being used for the basic index page in the default Express installation. You'll see that there's a `head` section and a `body` section, and within the `body` section there's a `block content` line with nothing inside it. This named `block` can be referenced by other Pug templates, such as the `index.pug` file in listing 3.5. The `block content` from the `index` file gets pushed into the `block content` area of the layout file when the views are compiled.

### **ADDING BOOTSTRAP TO THE ENTIRE APPLICATION**

If you want to add some external reference files to the entire application, then using the layout file makes sense in the current setup. So in `layout.pug`, you need to accomplish four things:

- Reference the Bootstrap and Font Awesome CSS files.
- Reference the Bootstrap JavaScript file.
- Reference jQuery and Tether, which Bootstrap requires.
- Add viewport metadata so that the page scales nicely on mobile devices.

The CSS file and the viewport metadata should both be in the head of the document, and the two script files should be at the end of the body section. The following listing shows all of this in place in `layout.pug`, with the new lines in bold.

#### **Listing 3.7 Updated layout.pug including Bootstrap references**

```
doctype html
html
  head
    meta(name='viewport', content='width=device-width, initial-scale=1.0') ①
    title= title
    link(rel='stylesheet', href='/stylesheets/bootstrap.min.css')          ②
    link(rel='stylesheet', href='/stylesheets/font-awesome.min.css')        ②
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
    script(src='https://code.jquery.com/jquery-3.1.1.slim.min.js')      ③
    script(src=
      https://cdnjs.cloudflare.com/ajax/libs/tether/1.4.0/js/tether.min.js) ③
    script(src='javascripts/bootstrap.min.js')                                ④
```

- ① Set viewport metadata for better display on mobile devices
- ② Include Bootstrap and Font Awesome CSS
- ③ Bring in jQuery and Tether, needed by Bootstrap
- ④ Bring in Bootstrap JavaScript file

With that done, any new template that you create will automatically have Bootstrap included and will scale on mobile devices—as long as your new templates extend the layout template, of course. Should you have any problems or unexpected results at this stage remember that Pug is very sensitive to indentation, spacing and new lines - all indentation must be done with spaces to get the correct nesting in the HTML output.

**TIP** If you followed along in appendix B, you'll also have some custom styles in the style.css file in /public/stylesheets/. This will prevent the default Express styles from overriding the Bootstrap files, and will help us get the look we want.

Now we're ready to test.

### VERIFY THAT IT WORKS

If the application isn't already running with nodemon, start it up and view it in your browser. The content hasn't changed, but the appearance should have. You should now have something looking like figure 3.9.

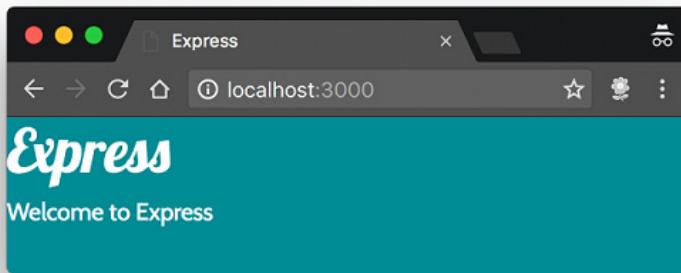


Figure 3.9 Bootstrap and our styles having an effect on the default Express index page.

If yours doesn't look like this make sure you have added the custom styles as outlined in appendix B. Remember you can get the source code of the application so far from GitHub on the chapter-03 branch. In a fresh folder in terminal, the following command will clone it:

```
$ git clone -b chapter-03 https://github.com/simonholmes/getting-MEAN-2.git
```

Now we've got something working locally; let's see how we can get it running on a live production server.

## 3.5 Make it live on Heroku

A common perceived headache with Node applications is deploying them to a live production server. We're going to get rid of that headache early on and push our Loc8r application onto a live URL already. As we iterate and build it up, we can keep pushing out the updates. For prototyping, this is great, because it makes it really easy to show our progress to others.

As mentioned in chapter 1, there are a few PaaS providers such as Google Cloud Platform, Nodejitsu, OpenShift, and Heroku. We're going to use Heroku here, but there's nothing stopping you from trying out other options.

Here we are going to get Heroku up and running and walk through a few basic Git commands to deploy our application to a live server.

### 3.5.1 Getting Heroku set up

Before you can use Heroku, you'll need to sign up for a free account and install the Heroku CLI on your development machine. Appendix B has more detailed information on how to do this. You'll also need a bash-compatible terminal; the default terminal for Mac users is fine, but the default CLI for Windows users won't do. If you're on Windows you'll need to download something like the GitHub terminal, which comes as part of the GitHub desktop application.

When you have everything set up, we can continue and get the application ready to push live.

#### **UPDATING PACKAGE.JSON**

Heroku can run applications on all different types of codebases, so we need to tell it what our application is running. As well as telling it that we're running a Node application using npm as the package manager, we need to tell it which version we're running to ensure that the production setup is the same as the development setup.

If you're not sure which versions of Node and npm you're running you can find out with a couple of terminal commands:

```
$ node --version
$ npm --version
```

At the time of writing, these commands return `v6.9.4` and `3.10.10`, respectively. Using the `~` syntax to add a wildcard for a minor version as you've seen previously, you need to add these to a new engines section in the package.json file. The complete updated package.json file is shown in the following listing, with the added section in bold.

#### **Listing 3.8 Adding an engines section to package.json**

```
{
  "name": "Loc8r",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "engines": { ①
    "node": "^6.9.4", ①
    "npm": "^3.10.10" ①
  },
  "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"
  }
}

```

- ① Add an engines section to package.json to tell Heroku which platform your application is on, and which version to use

When pushed up to Heroku, this will tell Heroku that our application uses the latest minor version of Node, 6, and the latest minor version of npm, 3.

### **CREATING A PROCFILE**

The package.json file will tell Heroku that the application is a Node application, but it doesn't tell it how to start it. For this we need to use a Procfile. A Procfile is used to declare the process types used by our application, and the commands used to start them.

For Loc8r we want a web process, and we want it to run the Node application. So in the root folder of the application, create a file called Procfile—this is case-sensitive and has no file extension. Enter the following line into the Procfile file:

```
web: npm start
```

When pushed up to Heroku, this file will simply tell Heroku that the application needs a web process and that it should run `npm start`.

### **TESTING IT LOCALLY WITH HEROKU LOCAL**

The Heroku CLI comes with a utility called Heroku Local. We can use this to verify our setup and run our application locally before pushing the application up to Heroku. If the application is currently running, stop it by pressing Ctrl-C in the terminal window running the process. Then in the terminal window make sure you are in your application folder enter the following command:

```
$ heroku local
```

All being well with the setup, this will start the application running on localhost again, but this time on a different port: 5000. The confirmation you get in terminal should be along these lines:

```
16:09:02 web.1 | > loc8r@0.0.1 start /path/to/your/application/folder
16:09:02 web.1 | > node ./bin/www
```

As well as these lines, you'll probably also see a warning that "No ENV file found", this is nothing to worry about at this stage. If you fire up a browser and head over to

localhost:5000—note that the port is 5000 instead of 3000—you should be able to see the application up and running again.

Now that we know the setup is working, it's time to push our application up to Heroku.

### 3.5.2 Pushing the site live using Git

Heroku uses Git as the deployment method. If you already use Git you'll love this approach; if you haven't you may feel a bit apprehensive about it, because the world of Git can be quite complex. But it doesn't need to be, and when you get going, you'll love this approach too!

#### **STORING THE APPLICATION IN GIT**

The first action is to store the application in Git, on your local machine. This is a three-step process, because you need to do the following:

1. Initialize the application folder as a Git repository.
2. Tell Git which files you want to add to the repository.
3. Commit these changes to the repository.

This might sound complex, but it really isn't. You just need a single, short terminal command for each step. If the application is running locally, stop it in terminal (Ctrl-C). Then, ensuring you're still in the root folder of the application, stay in terminal and run the following commands:

```
$ git init          1
$ git add --all      2
$ git commit -m "First commit" 3
```

- 1 Initializes folder as a local Git repository
- 2 Adds everything in folder to repository
- 3 Commits changes to repository with a message

These three things together will create a local Git repository containing the entire codebase for the application. When we go to update the application later on, and we want to push some changes live, we'll use the second two commands, with a different message, to update the repository.

Your local repository is now ready. It's time to create the Heroku application.

#### **CREATING THE HEROKU APPLICATION**

This next step will create an application on Heroku as a remote Git repository of your local repository. All this is done with a single terminal command:

```
$ heroku create
```

When this is done, you'll see a confirmation in terminal of the URL that the application will be on, the Git repository address, and the name of the remote repository. For example

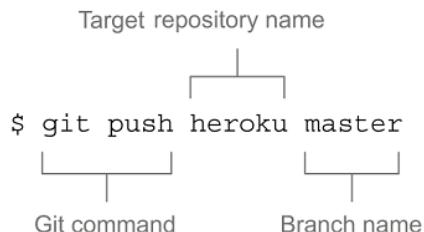
```
https://pure-temple-67771.herokuapp.com/ | git@heroku.com:pure-temple-67771.git
Git remote heroku added
```

If you log in to your Heroku account in a browser you'll also see that the application exists there. So you now have a place on Heroku for the application, and the next step is to push the application code up.

### DEPLOYING THE APPLICATION TO HEROKU

By now you have the application stored in a local Git repository, and you've created a new remote repository on Heroku. The remote repository is currently empty, so you need to push the contents of your local repository into the `heroku` remote repository.

If you don't know Git, there's a single command to do this, which has the following construct:



This command will push the contents of your local Git repository to the `heroku` remote repository. Currently, you have only a single branch in your repository, which is the `master` branch, so that's what you'll push to Heroku. See the following sidebar for more information on Git branches.

When you run this, terminal will display a load of log messages as it goes through the process, eventually showing – about 5 lines from the end – a confirmation that the application has been deployed to Heroku. This will be something like the following, except you'll have a different URL, of course:

```
http://pure-temple-67771.herokuapp.com deployed to Heroku
```

#### What are Git branches?

If you just work on the same version of the code and push it up to a remote repository like Heroku or GitHub periodically, you're working on the `master` branch. This is absolutely fine for linear development with just one developer. If you have multiple developers or your application is already published, then you don't really want to be doing your development on the `master` branch. Instead, you start a new branch from the `master` code in which you can continue development, add fixes, or build a new feature.

When work on a branch is complete it can be merged back into the `master` branch.

## **ABOUT WEB DYNOS ON HEROKU**

Heroku uses the concept of *dynos* for running and scaling an application. The more dynos you have, the more system resources and processes you have available to your application. Adding more dynos when your application gets bigger and more popular is really easy.

Heroku also has a great free tier, which is perfect for application prototyping and building a proof-of-concept. You get one web dyno for free with each application, which is more than adequate for our purposes here. If you have an application that needs more resources you can always log into your account and pay for more.

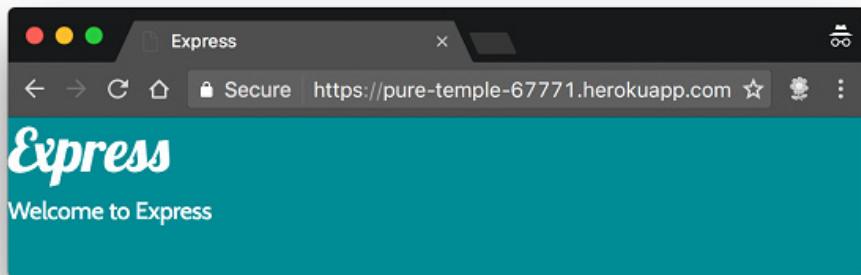
Now let's check out the live URL.

## **VIEWING THE APPLICATION ON A LIVE URL**

Everything is now in place, and the application is live on the internet! You can see it by typing in the URL given to you in the confirmation, via your account on the Heroku website, or by using the following terminal command:

```
$ heroku open
```

This will launch the application in your default browser, and you should see something like figure 3.10.



**Figure 3.10 MVC Express application running on a live URL**

Your URL will be different, of course, and within Heroku you can change it to use your domain name instead of the address it has given you. In the application settings on the Heroku website, you can also change it to use a more meaningful subdomain of herokuapp.com.

Having your prototype on an accessible URL is very handy for cross-browser and cross-device testing, as well as sending it out to colleagues and partners.

## A SIMPLE UPDATE PROCESS

Now that the Heroku application is set up, updating it will be really easy. Every time you want to push some new changes through, you just need three terminal commands:

```
$ git add --all          1
$ git commit -m "Commit message here" 2
$ git push heroku master 3
```

- 1 Add all changes to local Git repository
- 2 Commit changes to local repository with a useful message
- 3 Push changes to Heroku repository

That's all there's to it, for now at least. Things might get a bit more complex if you have multiple developers and branches to deal with, but the actual process of pushing the code to Heroku using Git remains the same.

## 3.6 Summary

In this chapter, we've covered

- Creating a new Express application
- Managing application dependencies with npm and the package.json file
- Modifying Express to meet an MVC approach to architecture
- Routes and controllers
- Creating new Node modules
- Publishing an Express application live onto Heroku using Git

In the next chapter, you'll get to know Express even more when we build out a prototype of the Loc8r application.

# 4

## *Building a static site with Node and Express*

### This chapter covers

- Prototyping an application through building a static version
- Defining routes for application URLs
- Creating views in Express using Pug and Bootstrap
- Using controllers in Express to tie routes to views
- Passing data from controllers to views

In chapter 3 you should have had an Express application running, set up in an MVC way, with Bootstrap included to help with building page layouts. Our next step is to build on this base, creating a static site that you can click through. This is a critical step in putting together any site or application. Even if you've been given a design or some wireframes to work from, there's no substitute for rapidly creating a realistic prototype that you can use in the browser. Something always comes to light in terms of layout or usability that hadn't been noticed before. From this static prototype, we'll take the data out from the views and put it into the controllers. By the end of this chapter we'll have intelligent views that can display data passed to them, and controllers passing hard-coded data to the views.

### Getting the source code

If you haven't yet built the application from chapter 3, you can get the code from GitHub on the chapter-03 branch at [github.com/simonholmes/getting-MEAN-2](https://github.com/simonholmes/getting-MEAN-2). In a fresh folder in terminal the following commands will clone it and install the npm module dependencies:

```
$ git clone -b chapter-03 https://github.com/simonholmes/getting-MEAN-2.git
$ cd getting-MEAN-2
$ npm install
```

In terms of building up the application architecture, this chapter will be focusing on the Express application as shown in figure 4.1.

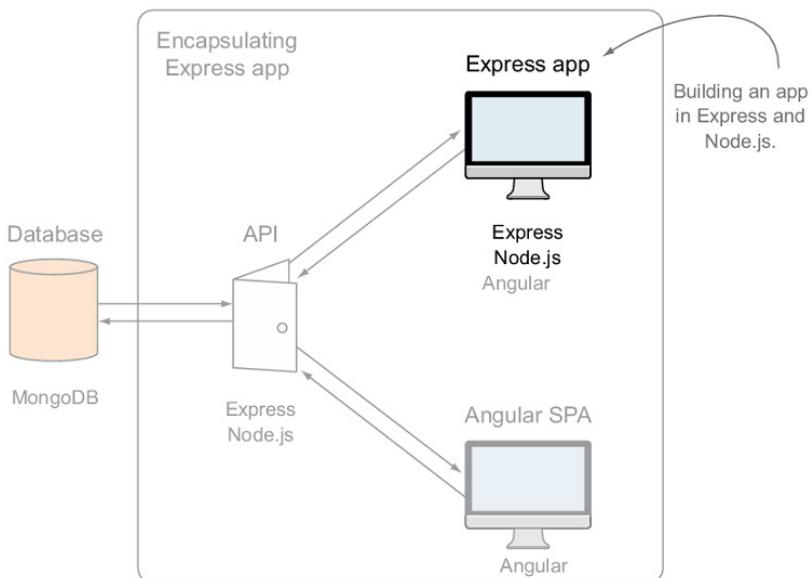


Figure 4.1 Using Express and Node to build a static site for testing views

There are two main steps being taken in this chapter, so there are two versions of the source code available. The first version contains all of the data in the views and represents the application as it stands at the end of section 4.4. This is available from GitHub on the chapter-04-views branch.

The second version has the data in the controllers, in the state the application will be at the end of this chapter. This is available from GitHub on the chapter-04 branch.

To get one of these use the following commands in a fresh folder in terminal, remembering to specify the branch that you want:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

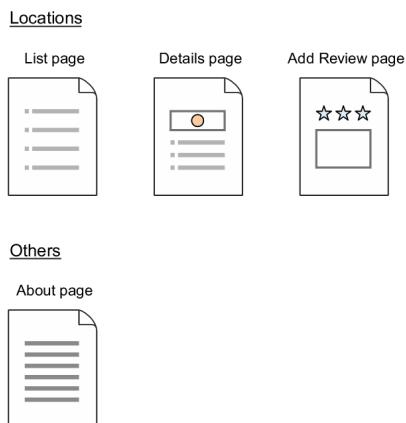
Licensed to Jacob Munkholm Hansen <au518331@uni.au.dk>

```
$ git clone -b chapter-04 https://github.com/simonholmes/getting-MEAN-2.git
$ cd getting-MEAN-2
$ npm install
```

Okay, let's get back into Express.

## 4.1 Defining the routes in Express

In chapter 2 we planned out the application and decided on the four pages we are going to build. There's a collection of Locations pages and a page in the Others collection as shown in figure 4.2.



**Figure 4.2 Collections of screens we'll be building for the Loc8r application**

Having a set of screens is great, but these need to relate to incoming URLs. Before we do any coding it's a good idea to map this out and to get a good standard in place. Take a look at table 4.1. It shows a simple mapping of the screens against URLs. These will form the basis of the routing for our application.

**Table 4.1 Defining a URL path, or route, for each of the screens in the prototype**

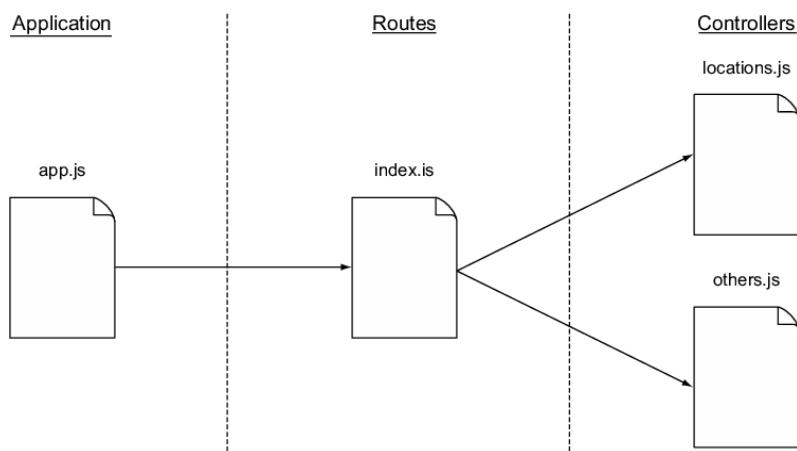
Collection	Screen	URL path
Locations	List of locations (this will be the homepage)	/
Locations	Location detail	/location
Locations	Location review form	/location/review/new
Others	About Loc8r	/about

For example, when somebody visits the homepage we want to show them a list of places, but when somebody visits the `/about` URL path we want to show them the information about Loc8r.

#### **4.1.1 Different controller files for different collections**

In chapter 3 you'll recall we moved any sense of controller logic out of the route definitions and into an external file. Looking to the future we know that our application will grow, and we don't want to have all the controllers in one file. A logical starting point for splitting them up is to divide them by collections.

So looking at the collections we've decided upon, we'll split the controllers up into Locations and Others. To see how this might work from a file architecture point of view, we can sketch out something like figure 4.3. Here the application includes the routes file, which in turn includes multiple controller files, each named according to the relevant collection.



**Figure 4.3 Proposed file architecture for routes and controllers in our application**

Here we have a single route file and one controller file for each logical collection of screens. This setup is designed to help us organize our code in line with how our application is organized. We'll look at the controllers shortly, but first we'll deal with the routes.

The time for planning is over; now it's time for action! So head back over to your development environment and open the application. We'll start off working in the routes file `index.js`.

#### **REQUIRING THE CONTROLLER FILES**

As shown in figure 4.3 we want to reference two controller files in this routes file. We haven't created these controller files yet; we'll do that shortly.

These files will be called locations.js and others.js and will be saved in app\_server/controllers. In index.js we'll require both of these files and assign each to a relevant variable name, as shown in the following listing.

#### **Listing 4.1 Requiring the controller files in app\_server/routes/index.js**

```
const express = require('express');
const router = express.Router();
const ctrlLocations = require('../controllers/locations'); ①
const ctrlOthers = require('../controllers/others'); ②
```

- ① Replace existing ctrlMain reference with two new requires

Now we have two variables we can reference in the route definitions, which will contain different collections of routes.

#### **SETTING UP THE ROUTES**

In index.js you'll need to have the routes for the three screens in the Locations collection, and also the About page in the Others collection. Each of these routes will also need a reference to the controllers. Remember that routes simply serve as a mapping service, taking the URL of an incoming request and mapping it to a specific piece of application functionality.

From table 4.1 we already know which paths we want to map, so it's a case of putting it all together into the routes/index.js file. What you need to have in the file is shown in entirety in the following listing.

#### **Listing 4.2 Defining the routes and mapping them to controllers**

```
const express = require('express');
const router = express.Router();
const ctrlLocations = require('../controllers/locations'); ①
const ctrlOthers = require('../controllers/others'); ②

/* Locations pages */
router.get('/', ctrlLocations.homelist); ②
router.get('/location', ctrlLocations.locationInfo); ②
router.get('/location/review/new', ctrlLocations.addReview); ②

/* Other pages */
router.get('/about', ctrlOthers.about); ③

module.exports = router;
```

- ① Require controller files
- ② Define location routes and map them to controller functions
- ③ Define other routes

This routing file maps the defined URLs to some specific controllers, although we haven't created those yet. So let's take care of that now and create the controllers.

## 4.2 Building basic controllers

At this point we're going to make the controllers really basic so that our application will run and we can test the different URLs and routing.

### 4.2.1 Setting up controllers

We currently have one file, the main.js file in the controllers folder (in the app\_server folder) that has a single function that's controlling the homepage. This is shown in the following code snippet:

```
/* GET 'home' page */const index = function(req, res){
  res.render('index', { title: 'Express' });
};
```

We don't actually want a "main" controller file anymore, but we can use this as a template. Start by renaming this as *others.js*.

#### ADDING THE OTHERS CONTROLLERS

Recall from listing 4.2 that we want one controller in others.js called `about`. So rename the existing `index` controller to `about`, keep the same view template for now, and update the title property to be something relevant. This will help you easily test that the route is working as expected. The following listing shows the full contents of the others.js controller file following these couple of little changes.

#### Listing 4.3 Others controller file

```
/* GET 'about' page */const about = function(req, res){ ①
  res.render('index', { title: 'About' });
}; ①
module.exports = {
  about ②
};
```

- ① Define route using same view template but changing title to About
- ② Update the export to reflect the name change

That's the first one done, but the application still won't work, as there aren't any controllers for the Locations routes yet.

#### ADDING THE LOCATIONS CONTROLLERS

Adding the controllers for the Locations routes is going to be pretty much the same process. In the routes file we specified the name of the controller file to look for, and the name of the three controller functions.

In the controllers folder create a file called locations.js, and create and export three basic controller functions: homelist, locationInfo, and addReview. The following listing shows how this should look.

```
Listing 4.4 Locations controller file
/* GET 'home' page */
const homelist = function(req, res){
  res.render('index', { title: 'Home' });
};

/* GET 'Location info' page */
const locationInfo = function(req, res){
  res.render('index', { title: 'Location info' });
};

/* GET 'Add review' page */
const addReview = function(req, res){
  res.render('index', { title: 'Add review' });
};

module.exports = {
  homelist,
  locationInfo,
  addReview
};
```

That looks like everything is in place, so let's test it.

## 4.2.2 Testing the controllers and routes

Now that the routes and basic controllers are in place you should be able to start and run the application. If you don't already have it running with nodemon, head to the root folder of the application in the terminal and start it up:

```
$ nodemon
```

---

### Troubleshooting

If you're having problems restarting the application at this point, the main thing to check is that all of the files, functions, and references are named correctly. Look at the error messages you're getting in the terminal window and see if they give you any clues. Sometimes they're more helpful than others! Take a look at the following possible error, and we'll pick out the parts that are interesting to us:

```
module.js:340
  throw err;
  ^
Error: Cannot find module '../controllers/other'1
  at Function.Module._resolveFilename (module.js:338:15)
  at Function.Module._load (module.js:280:25)
  at Module.require (module.js:364:17)
  at require (module.js:380:17)
```

```

at module.exports (/Users/sholmes/Dropbox/Manning/Getting-
MEAN/Code/Loc8r/BookCode/routes/index.js:2:3) ②
  at Object.<anonymous> (/Users/sholmes/Dropbox/Manning/Getting-
MEAN/Code/Loc8r/BookCode/app.js:26:20)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)

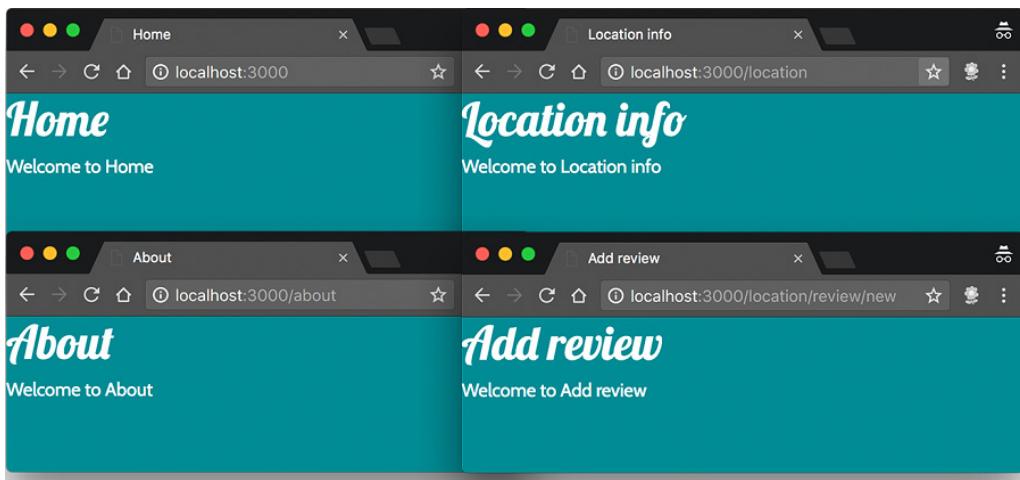
```

① Clue one: a module can't be found

② Clue two: file throwing error

First, you can see that a module called `other` can't be found #1. Further down the stack trace you can see the file where the error originated #2. So you'd then open the `routes/index.js` file and discover that you'd written `require('../controllers/other')` when the file you want to require is `others.js`. So to fix the problem you'd simply need to correct the reference by changing it to `require('../controllers/others')`.

All being well, this should give you no errors, meaning that all of the routes are pointing to controllers. So you can head over to your browser and check each of the four routes we've created, such as `localhost:3000` for the homepage and `localhost:3000/location` for the location information page. Because we changed the data being sent to the view template by each of the controllers, we'll easily be able to see that each one is running correctly because the title and heading should be different on each page. Figure 4.4 shows a collection of screenshots of the newly created routes and controllers.



**Figure 4.4** Screenshots of the four routes created so far, with different heading text coming through from the specific controllers associated with each route

From this we can see that each route is getting unique content, so we know that the routing and controller setup has worked.

The next stage in this prototyping process is to put some HTML, layout, and content onto each screen. We'll do this using views.

## 4.3 Creating some views

When you have your empty pages, paths, and routes sorted out, it's time to get some content and layout into our application. This is where we really bring it to life and can start to see our idea become reality. For this step, the technologies that we're going to use are Pug and Bootstrap. Pug is the template engine we're using in Express (although you can use others if you prefer) and Bootstrap is a front-end layout framework that makes it really easy to build a responsive website that looks different on desktop and mobile devices.

### 4.3.1 A look at Bootstrap

Before getting started, let's take a quick look at Bootstrap. We're not going to go into all the details about Bootstrap and everything it can do, but it's useful to see some of the key concepts before we try to throw it into a template file.

#### **BOOTSTRAP RESPONSIVE GRID SYSTEM**

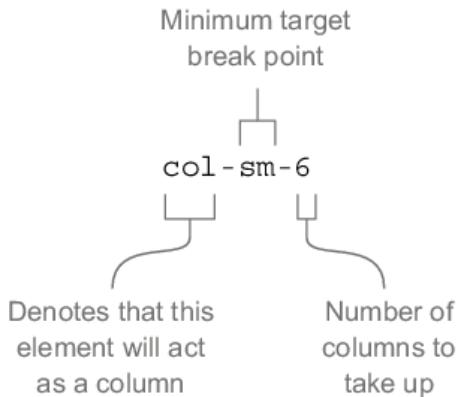
Bootstrap uses a 12-column grid. No matter the size of the display you're using, there will always be these 12 columns. On a phone each column will be narrow, and on a large external monitor each column will be wide. The fundamental concept of Bootstrap is that you can define how many columns an element will use, and this can be a different number for different screen sizes.

Bootstrap has various CSS references that let you target up to five different pixel-width breakpoints for your layouts. These breakpoints are noted in table 4.2 along with the example device that you'd be targeting at each size.

**Table 4.2 Breakpoints that Bootstrap targets for different types of devices**

Breakpoint name	CSS reference	Example device	Width in pixels
Extra-small devices	(none)	Small phones	Less than 576
Small devices	sm	Smart phones	576 or more
Medium devices	md	Tablets	768 or more
Large devices	lg	Laptops	992 or more
Extra large devices	xl	External monitors	1,200 or more

To define the width of an element you combine a CSS reference from table 4.2 with the number of columns you wish it to span. A class denoting a column is constructed like this:



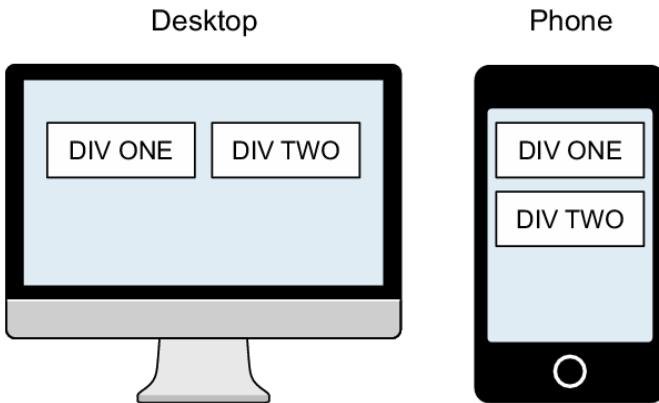
This class of `col-sm-6` will make the element it's applied to take up 6 columns on screens of size `sm` and larger. So on tablets, laptops, and monitors this column will take up half of the available width.

To get the responsive side of things to work, you can apply multiple classes to a single element. So if you wanted a `div` to span the entire width of the screen on the phone, but only half of the width on tablets and larger, you could use the following code snippet:

```
<div class="col-12 col-md-6"></div>
```

The `col-12` class tells the layout to use 12 columns on extra-small devices, and the `col-md-6` class tells the layout to use 6 columns for medium devices and above. Figure 4.5 illustrates the effect this has on different devices if you have two of these, one after another on the page, like this:

```
<div class="col-12 col-md-6">DIV ONE</div>
<div class="col-12 col-md-6">DIV TWO</div>
```

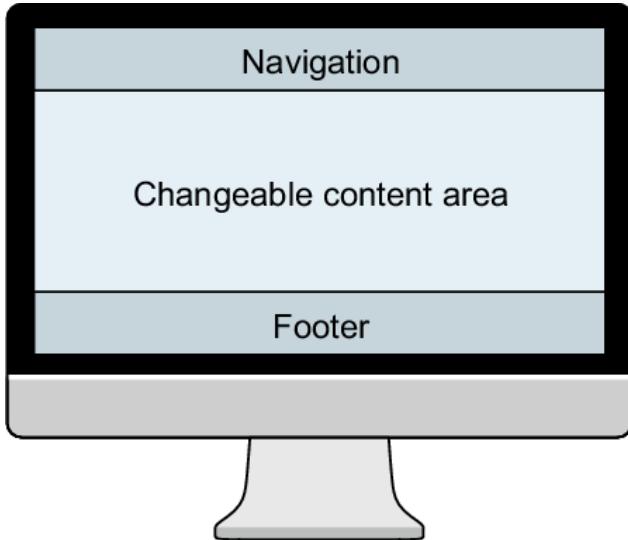


**Figure 4.5** Bootstrap's responsive column system on a desktop and mobile device. CSS classes are used to determine the number of columns out of 12 that each element should take up at different screen resolutions.

This approach allows for a very semantic way of putting together responsive templates, and we'll rely heavily on this for the Loc8r pages. Speaking of which, let's make a start.

### 4.3.2 Setting up the HTML framework with Pug templates and Bootstrap

There are some common requirements across all of the pages we'll have in the application. At the top of the page we'll want a navigation bar and logo, at the bottom of the page we'll have a copyright notice in the footer, and we'll have a content area in the middle. What we're aiming for here is something like figure 4.6.



**Figure 4.6 Basic structure of the reusable layout, comprising a standard navigation bar and footer with an extendable and changeable content area in between**

This framework for a layout is pretty simple, but it suits our needs. It will give a consistent look and feel, while allowing for all types of content layouts to go in the middle.

As you saw in chapter 3, Pug templates use the concept of extendable layouts, enabling you to define this type of repeatable structure just once in a layout file. In the layout file you can specify which parts can be extended; once you have this layout file set up you can extend it as many times as you want. Creating the framework in a layout file means that you only have to do it once, and only have to maintain it in one place.

### **LOOKING AT THE LAYOUT**

To build the common framework then, we're mainly going to be working with the **layout.pug** file in the `app_server/views` folder. This is currently pretty minimal and looks like this code snippet:

```
doctype html
html
head
  meta(name='viewport', content='width=device-width, initial-scale=1.0')
  title= title
  link(rel='stylesheet', href='/stylesheets/bootstrap.min.css')
  link(rel='stylesheet', href='/stylesheets/font-awesome.min.css')
  link(rel='stylesheet', href='/stylesheets/style.css')
body
  block content
    script(src='https://code.jquery.com/jquery-3.1.1.slim.min.js')
```

```
script(src='https://cdnjs.cloudflare.com/ajax/libs/tether/1.4.0/js/tether.min.js')
script(src='/javascripts/bootstrap.min.js')
```

There isn't any HTML content in the body area at all yet, just a single extendable block called `content` and a couple of script references. We want to keep all of this, but need to add a navigation section above the `content` block, and a footer below it.

### BUILDING THE NAVIGATION

Bootstrap offers a collection of elements and classes that can be used to create a sticky navigation bar that's fixed to the top, and collapses the options into a dropdown menu on mobile devices. We're not going to explore the details of Bootstrap's CSS classes here, as we really just need to grab the example code from the Bootstrap website, tweak it a little, and update it with the correct links.

In the navigation we want to have

4. The Loc8r logo linking to the homepage
5. An About link on the left, pointing to the `/about` URL page

The code to do all of this is in the following snippet, and can be placed in the **layout.pug** file above the `block content` line:

```
nav.navbar.fixed-top.navbar-toggleable-md.navbar-inverse ①
  .container
    button.navbar-toggler.navbar-toggler-right(type='button', data-toggle='collapse',
      data-target='#navbarMain') ②
      span.navbar-toggler-icon ②
    a.navbar-brand(href='/' ) Loc8r ③
    #navbarMain.navbar-collapse.collapse ③
      ul.navbar-nav.mr-auto ④
        li.nav-item ④
          a.nav-link(href='/about/') About ④
```

- ① Set up a Bootstrap navigation bar fixed to top of window
- ② Set up collapsing navigation for smaller screen resolutions
- ③ Add a brand-styled link to homepage
- ④ Set up collapsing navigation for smaller screen resolutions
- ④ Add About link to left side of bar

If you pop that in and run it you'll notice that the navigation now overlays the page heading. This will be fixed when we build the layouts for the content area in sections 4.3.3 and 4.4, so it's nothing to worry about.

**TIP** Remember that Pug doesn't include any HTML tags, and that **correct Indentation Is critical** to provide the expected outcome.

And that's it for the navigation bar; it's all we'll need for a while. If Pug and Bootstrap are new to you it might take a little while to get used to the approach and the syntax, but as you can see, you can achieve a lot with little code.

### **WRAPPING THE CONTENT**

Working down the page from top to bottom the next area is the `content` block. There isn't much to do with this, as other Pug files will decide the contents. As it stands though, the `content` block is anchored to the left margin and is unconstrained, meaning that it will stretch the full width of any device.

Addressing this is easy with Bootstrap. Still in `layout.pug` you simply need to wrap the `content` block in a container `div` like so, remembering to ensure that the indentation is correct:

```
.container
  block content
```

The `div` with a class of `container` will be centered in the window, and constrained to sensible maximum widths on large displays. The contents of a container `div` will remain aligned to the left as normal though.

### **ADDING THE FOOTER**

At the bottom of the page we want to add a standard footer. We could add a bunch of links in here, terms and conditions, or a privacy policy. For now we'll just add a copyright notice and keep things simple. As it's going in the layout file it will be really easy to update this across all of the pages should we need to at a later date.

The following code snippet shows all the code needed for our simple footer in `layout.pug`:

```
footer
  .row
    .col-12
      small &copy; Simon Holmes 2017
```

This will be best placed inside the container `div` that holds the `content` block, so when you add it in make sure that the `footer` line is at the same level of indentation as the `block content` line.

### **ALL TOGETHER NOW**

Now that the navigation bar, content area, and footer are all dealt with, that's the complete layout file. The full code for `layout.pug` is shown in the following listing (modifications in bold).

#### **Listing 4.5 Final code for the layout framework in app\_server/views/layout.pug**

```
doctype html
html
```

```

head
  meta(name='viewport', content='width=device-width, initial-scale=1.0')
  title= title
  link(rel='stylesheet', href='/stylesheets/bootstrap.min.css')
  link(rel='stylesheet', href='/stylesheets/font-awesome.min.css')
  link(rel='stylesheet', href='/stylesheets/style.css')
body
  nav.navbar.fixed-top.navbar-toggleable-md.navbar-inverse      ①
    .container
      button.navbar-toggler.navbar-toggler-right(type='button', data-
        toggle='collapse', data-target='#navbarMain')
        span.navbar-toggler-icon
      a.navbar-brand(href=' / ') Loc8r
      #navbarMain.navbar-collapse.collapse
        ul.navbar-nav.mr-auto
          li.nav-item
            a.nav-link(href='/about/') About
  .container
    block content
    footer
      .row
        .col-12
          small &copy; Simon Holmes 2017
script(src='https://code.jquery.com/jquery-3.1.1.slim.min.js')
script(src='https://cdnjs.cloudflare.com/ajax/libs/tether/1.4.0/js/tether.min.js')
script(src='/javascripts/bootstrap.min.js')

```

- ① Starting layout with fixed navigation bar
- ② Extendable content block now wrapped in a container div
- ③ Simple copyright footer in same container as content block

That's all it takes to create a responsive layout framework using Bootstrap, Pug, and Express. If you've got that all in place and run the application you should see something like the screenshots in figure 4.7, depending on your device.

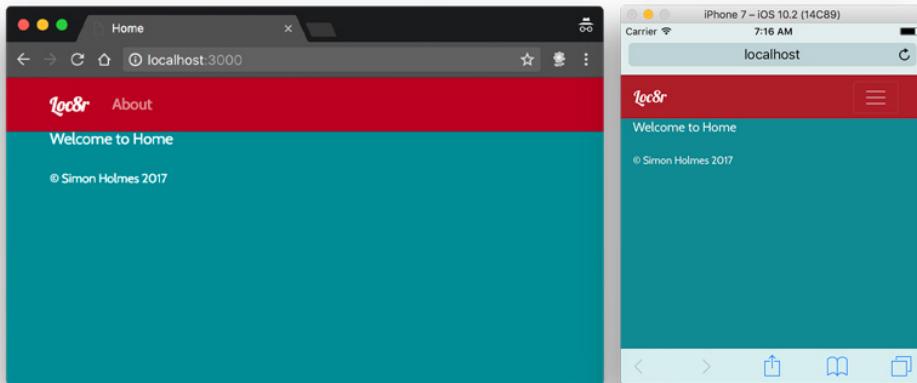


Figure 4.7 The homepage after the layout template has been set up. Bootstrap has automatically collapsed the navigation on the small screen size of the phone. The navigation bar overlaps the content, but that will be fixed when the content layouts are created.

You'll see that the navigation still overlaps the content, but we'll address that very soon when we start looking at the content layouts. It's a good indication that the navigation is working as we want it to though—we want the navigation to be ever-present, fixed to the top of the window. Also notice how Bootstrap has collapsed the navigation into a dropdown menu on the smaller screen of the phone. Pretty nice isn't it, for very little effort on our part?

**TIP** If you can't access your development site on a phone you can always try resizing your browser window, or Google Chrome and Firefox both allow you to emulate various different mobile devices through their built in developer tools.

Now that the generic layout template is complete, it's time to start building out the actual pages of our application.

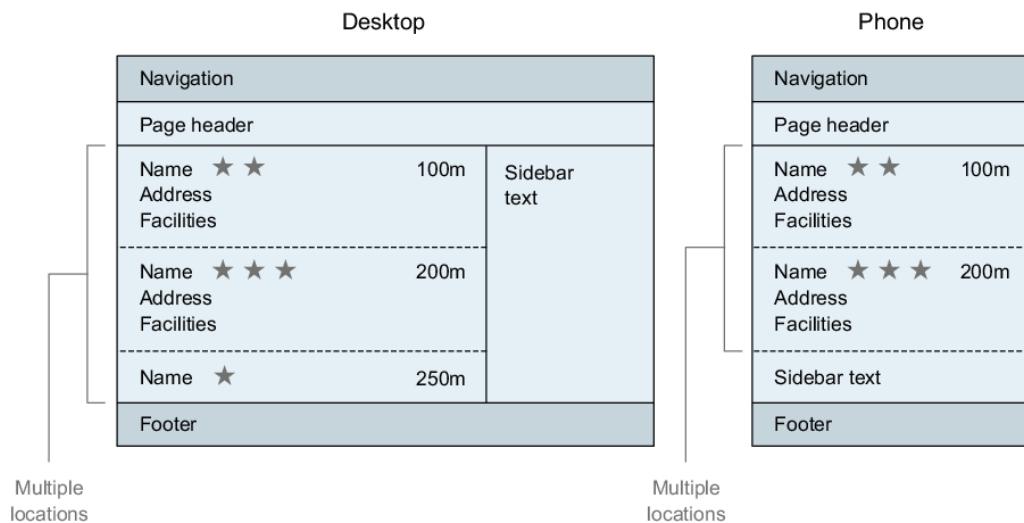
### 4.3.3 Building a template

When building templates, start with whichever one makes the most sense to you. This might be the most complicated or the most simple, or just the first in the main user journey. For Loc8r a good place to start is the homepage; this is the example we'll go through in most detail.

## DEFINING A LAYOUT

The primary aim for the homepage is to display a list of locations. Each location will need to have a name, an address, the distance away, users' ratings, and a facilities list. We'll also want to add a header to the page, and some text to put the list in context, so that users know what they're looking at when they first visit.

You may find it useful, as I do, to sketch out a layout or two on a piece of paper or a whiteboard. I find this really helpful for creating a starting point for the layout, making sure you've got all of the pieces you need on a page without getting bogged down in the technicalities of any code. Figure 4.8 shows what I've sketched for the home-page of Loc8r.



**Figure 4.8** Desktop and mobile layout sketches for the homepage. Sketching out the layouts for a page can give you a quick idea of what you're going to build, without getting distracted by the intricacies of Photoshop or technicalities of code.

You'll see that there are two layouts, a desktop and a phone. It's worth making the responsive distinction at this point, with your understanding of what Bootstrap can do and how it works in the back of your mind.

At this stage, the layouts are by no means final and we may well find that we'll tweak them and change them as we build the code. But any journey is easier if there's a destination and a map; this is what the sketches give us. We can start off our coding in the right direction. The few minutes it takes to do this upfront can save us hours later on—moving parts around, or even throwing them out and starting again, is much easier with a sketch than with a load of code.

Now that we've got an idea of the layout and the pieces of content required, it's time to put it together in a new template.

### **SETTING UP THE VIEW AND THE CONTROLLER**

The first step is to create a new view file and link it to the controller. So in the `app_server/views` folder make a copy of the `index.pug` view and save it in the same folder as `locations-list.pug`. It's best not calling it "homepage" or something similar, as at some point we may change our mind about what should be displayed on the homepage. This way, the name of the view is clear, and it can be used anywhere without confusion.

The second step is to tell the controller for the homepage that we want to use this new view. The controller for the homepage is in the `locations.js` file in `app_server/controllers`. Update this to change the view called by the `homelist` controller. This should look like the following code snippet (modifications in bold):

```
const homelist = function(req, res){
  res.render('locations-list', { title: 'Home' });
};
```

Now let's build the view template.

### **CODING THE TEMPLATE: PAGE LAYOUT**

When actually writing the code for the layouts, I prefer to start with the big pieces, and then move toward the detail. As we're extending the layout file, the navigation bar and footer are already done, but there's still the page header, the main area for the list, and the sidebar to consider.

At this point we need to take a first stab at how many of the 12 Bootstrap columns we want each element to take up on which devices. The following code snippet shows the layout of the three distinct areas of the Loc8r List page in `locations-list.pug`:

```
.row.banner
  .col-12
    h1 Loc8r
    small &nbsp;Find places to work with wifi near you!
  .row
    .col-12.col-md-8
      p List area.
    .col-12.col-md-4
      p.lead Loc8r helps you find places to work when out and about.
```

- ① Page header that fills entire width of the screen
- ② Container for list of locations, spanning all 12 columns on extra-small and small devices and 8 columns on medium devices and larger
- ③ Container for secondary or sidebar information, spanning all 12 columns on extra-small and small devices and 4 columns on medium devices and larger

I did go back and forth a bit, testing the columns at various resolutions until I was happy with them. Having device simulators can make this process easier, but a really simple method is to just change the width of your browser window to force the different Bootstrap breakpoints. When you've got something you think is probably okay, you can push it up to Heroku and test it out for real on your phone or tablet.

### CODING THE TEMPLATE: LOCATIONS LIST

Now that the containers for the homepage are defined, it's time for the main area. We have an idea of what we want in here from the sketches drawn for the page layout. Each place should show the name, address, its rating, how far away it is, and the key facilities.

Because we're creating a clickable prototype all of the data will be hard-coded into the template for now. It's the quickest way of putting a template together and ensuring that we have the information we want displayed the way we want. We'll worry about the data side of it later. If you're working from an existing data source, or have constraints around what data you can use, then naturally you'll have to bear that in mind when creating the layouts.

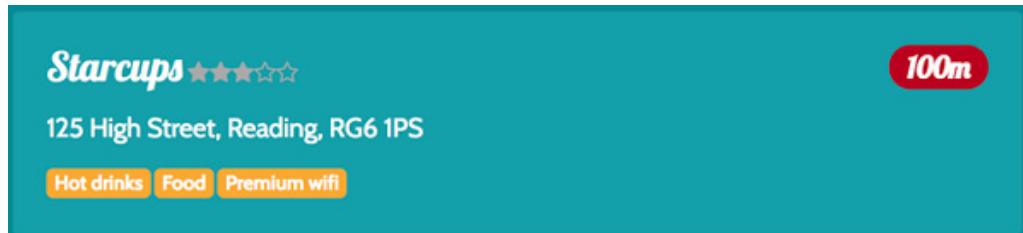
Again, getting a layout you're happy with will take a bit of testing, but Pug and Bootstrap working together make this process considerably easier than it might be. The following code snippet shows what I've come up with for a single location to replace the `p List` area placeholder in `locations-list.pug`:

```
.card
  .card-block
    h4
      a(href="/location") Starcups
      small &nbsp;
        i.fa.fa-star
        i.fa.fa-star
        i.fa.fa-star
        i.fa.fa-star-o
        i.fa.fa-star-o
      span.badge.badge-pill.badge-default.float-right 100m
      p.address 125 High Street, Reading, RG6 1PS
      .facilities
        span.badge.badge-warning Hot drinks
        span.badge.badge-warning Food
        span.badge.badge-warning Premium wifi
```

- 1 Create a new Bootstrap card and “card block” to wrap the content
- 2 Name of listing and a link to location
- 3 Use Font Awesome icons to output a star rating
- 4 Use Bootstrap's badge helper class to hold distance away
- 5 Address of location
- 6 Facilities of location, output using Bootstrap's badge classes

Once again you can see how much you can achieve with relatively little effort and code, all thanks to the combination of Pug and Bootstrap. Remember, there are some custom classes to

help with styling in the `styles.css` file in `public/stylesheets` available in the GitHub repo. Without it, your visuals will look much different. To give you an idea of what the preceding code snippet does, it will render like figure 4.9.



**Figure 4.9** Onscreen rendering of a single location on the List page

This section is set to go across the full width of the available area, so 12 columns on all devices. Remember, though, that this section is nested inside a responsive column, so that full width is the full width of the containing column, not necessarily the browser viewport.

This will probably make more sense when we put it all together and see it in action.

## CODING THE TEMPLATE: PUTTING IT TOGETHER

So we've got the layout of page elements, the structure of the list area, and some hard-coded data. Let's see what it looks like all together. To get a better feel of the layout in the browser it will be a good idea to duplicate and modify the List page so that we have a number of locations showing up. The code, including just a single location for brevity, is shown in the following listing.

#### **Listing 4.6 Complete template for app\_server/views/locations-list.pug**

```
extends layout

block content
  .row.banner
    .col-12
      h1 Loc8r
      small &nbsp;Find places to work with wifi near you!
  .row
    .col-12.col-md-8
      .card
        .card-block
          h4
            a(href="/location") Starcups
            small &nbsp;
              i.fa.fa-star
              i.fa.fa-star
              i.fa.fa-star
              i.fa.fa-star-o
              i.fa.fa-star-o
```

```

100m

address 125 High Street, Reading, RG6 1PS



facilities



- Hot drinks
- Food
- Premium wifi



.col-12.col-md-4



lead Looking for wifi and a seat? Loc8r helps you find places to work when out and about. Perhaps with coffee, cake or a pint? Let Loc8r help you find the place you're looking for.


```

- 1 Start header area
- 2 Start responsive main listing column section
- 3 An individual listing; duplicate this section to create list of multiple items
- 4 Set up sidebar area and populate with some content

When you've got this all in place, you've got the homepage listing template all done. If you run the application and head to localhost:3000 you should see something like figure 4.10.

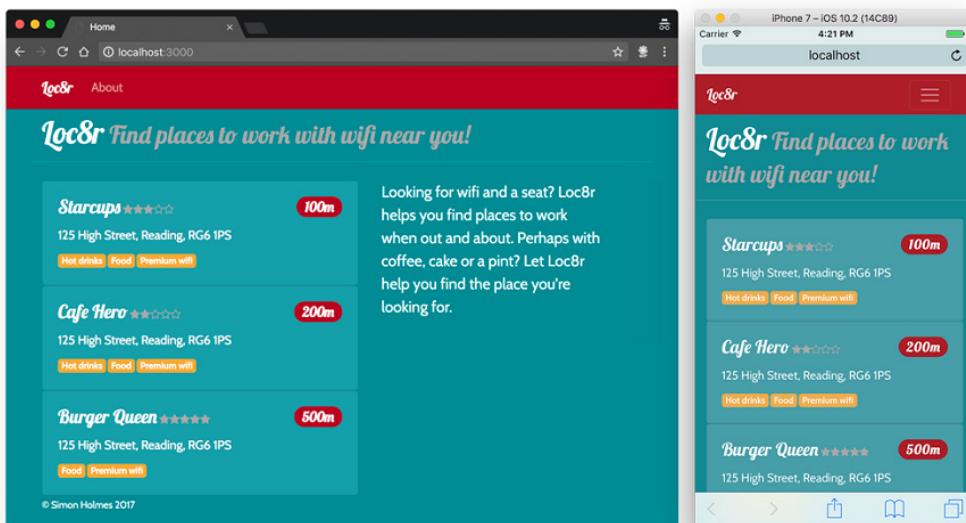


Figure 4.10 Responsive template for the homepage in action on different devices

See how the layout changes between a desktop view and a mobile view? That's all thanks to Bootstrap's responsive framework and our choice of CSS classes. Scrolling down in the mobile view you'll see the sidebar text content between the main list and the footer. On the smaller screen it's more important to display the list in the available space than the text.

So that's great; we've got a responsive layout for the homepage using Pug and Bootstrap in Express and Node. Let's quickly add the other views.

## 4.4 Adding the rest of the views

The Locations List page is done, so we now need to look at creating the other pages to give users a site they can click through. In this section we'll cover adding these pages:

1. Details
2. Add Review
3. About

We won't go through the process in so much detail for all of them though, just a bit of explanation, the code, and the output. You can always download the source code from GitHub if you prefer.

### 4.4.1 Details page

The logical step, and arguably the next most important page to look at, is the Details page for an individual location.

This page needs to display all of the information about a location, including

- Name
  - Address
  - Rating
  - Opening hours
  - Facilities
  - Location map
  - Reviews, each with
    - Rating
    - Reviewer name
    - Review date
  - Review text
  - Button to add a new review
- Text to set the context of the page

That's quite a lot of information! This is the most complicated single template that we'll have in our application.

#### **PREPARATION**

The first step is to update the controller for this page to use a different view. Look for the `locationInfo` controller in the `locations.js` file in `app_server/controllers`. Change the name of the view to be `location-info`, as per the following code snippet:

```
const locationInfo = function(req, res){
  res.render('location-info', { title: 'Location info' });
};
```

Remember, if you run the application at this point it won't work because Express can't find the view template. Not surprising really, as we haven't created it yet. That's the next part.

### THE VIEW

Create a new file in `app_server/views` and save it as `location-info.pug`. The content for this is shown in listing 4.7. This is the largest listing in this book. Remember that for the purposes of this stage in the prototype development, we're generating clickable pages with the data hard-coded directly into them.

#### **Listing 4.7 View for the Details page, `app_server/views/location-info.pug`**

```
extends layout
block content
  .row.banner
    .col-12
      h1 Starcups
  .row
    .col-12.col-lg-9
      .row
        .col-12.col-md-6
          p.rating
            i.fa.fa-star
            i.fa.fa-star
            i.fa.fa-star
            i.fa.fa-star-o
            i.fa.fa-star-o
          p 125 High Street, Reading, RG6 1PS
        .card.card-primary
          .card-block
            h2.card-title Opening hours
            p.card-text Monday - Friday : 7:00am - 7:00pm
            p.card-text Saturday : 8:00am - 5:00pm
            p.card-text Sunday : closed
        .card.card-primary
          .card-block
            h2.card-title Facilities
            span.badge.badge-warning
              i.fa.fa-check
              | &nbsp;Hot drinks
            span.badge.badge-warning
              i.fa.fa-check
              | &nbsp;Food
            span.badge.badge-warning
              i.fa.fa-check
              | &nbsp;Premium wifi
            .col-12.col-md-6.location-map
```

```

.card.card-primary
  .card-block
    h2.card-title Location map
    img.img-
      fluid.rounded(src='http://maps.googleapis.com/maps/api/staticmap?center=51.455
        041,-0.9690884&zoom=17&size=400x350&sensor=false&markers=51.455041,-
        0.9690884&scale=2') 4
  .row
    .col-12
      .card.card-primary.review-card
        .card-block
          a.btn.btn-primary.float-right(href='/location/review/new') Add review 5
          h2.card-title Customer reviews
          .row.review
            .col-12.no-gutters.review-header
              span.rating
                i.fa.fa-star
                i.fa.fa-star
                i.fa.fa-star
                i.fa.fa-star-o
                i.fa.fa-star-o
              span.reviewAuthor Simon Holmes
              small.reviewTimestamp 16 February 2017
            .col-12
              p What a great place.
          .row.review
            .col-12.no-gutters.review-header
              span.rating
                i.fa.fa-star
                i.fa.fa-star
                i.fa.fa-star
                i.fa.fa-star-o
                i.fa.fa-star-o
              span.reviewAuthor Charlie Chaplin
              small.reviewTimestamp 14 February 2017
            .col-12
              p It was okay. Coffee wasn't great.
  .col-12.col-lg-3 6
  p.lead
    | Starcups is on Loc8r because it has accessible wifi and space to sit down
    with your laptop and get some work done.
  p
    | If you've been and you like it - or if you don't - please leave a review to
    help other people just like you.

```

- 1 Start with page header
- 2 Set up nested responsive columns needed for template
- 3 One of several Bootstrap card components used to define information areas, in this case opening hours
- 4 Use static Google Maps image, including coordinates in the query string 51.455041,-0.9690884
- 5 Create link to Add Review page using Bootstrap's button helper class
- 6 Final responsive column for sidebar contextual information

So that's a pretty long template, and we'll look at how we can shorten that soon. But the page itself is pretty complex and contains a lot of information, and a few nested responsive columns. Imagine how much longer it would be if it was written in full HTML, though!

Make sure you have the full version of style.css from GitHub included, as we're using that to give a bit of life on top of the standard Bootstrap theme. With that all done, the Details page layout is complete, and you can head over to localhost:3000/location to check it out. Figure 4.11 shows how this layout looks in a browser and on a mobile device.

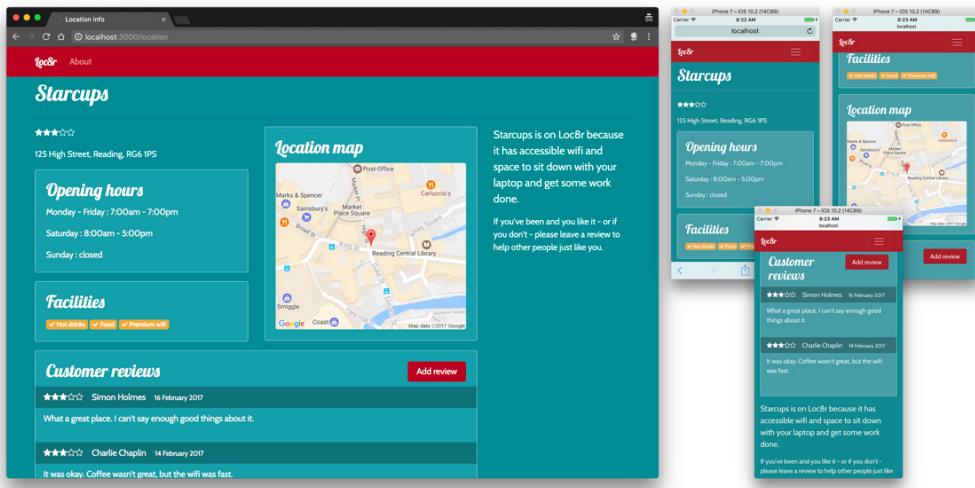


Figure 4.11 Details page layout on desktop and mobile devices

The next step in this user journey is the Add Review page, which has much simpler requirements.

#### 4.4.2 Adding Review page

This will be a pretty straightforward page. It only really needs to hold a form containing the user's name and a couple of input fields for the rating and review.

The first step is to update the controller to reference a new view. In app\_server/controllers/locations.js change the addReview controller to use a new view location-review-form, like in the following code snippet:

```
const addReview = function(req, res){
  res.render('location-review-form', { title: 'Add review' });
};
```

The second step is to create the view itself. In the views folder app\_server/views create a new file called location-review-form.pug. Because this is designed to be a clickable prototype we're not going to be posting the form data anywhere, so the aim is just to get the action to redirect to the Details page that displays the review data. In the form then, we'll set the action to be /location and the method to get. Later this will need to be a post method, but this will give

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>

us the functionality we need for now. The entire code for the review form page is shown in the following listing.

#### **Listing 4.8 View for the Add Review page, app\_server/views/location-review-form.pug**

```

extends layout
block content
  .row.banner
    .col-12
      h1 Review Starcups
  .row
    .col-12.col-md-8
      form(action="/location", method="get", role="form") ①
        .form-group.row
          label.col-10.col-sm-2.col-form-label(for="name") Name
          .col-12.col-sm-10
            input#name.form-control(name="name") ②
        .form-group.row
          label.col-10.col-sm-2.col-form-label(for="rating") Rating
          .col-12.col-sm-2
            select#rating.form-control.input-sm(name="rating") ③
              option 5 ③
              option 4 ③
              option 3 ③
              option 2 ③
              option 1 ③
        .form-group.row
          label.col-sm-2.col-form-label(for="review") Review
          .col-sm-10
            textarea#review.form-control(name="review", rows="5") ④
            button.btn.btn-primary.float-right Add my review ⑤
        .col-12.col-md-4

```

- ① Set form action to /location and method to get
- ② Input field for reviewer to leave name
- ③ Dropdown select box for rating 1 to 5
- ④ Text area for text content of review
- ⑤ Submit button for form

Bootstrap has a lot of helper classes for dealing with forms, which are evident in listing 4.8. But it's a pretty simple page, and when you run it, it should look like figure 4.12.

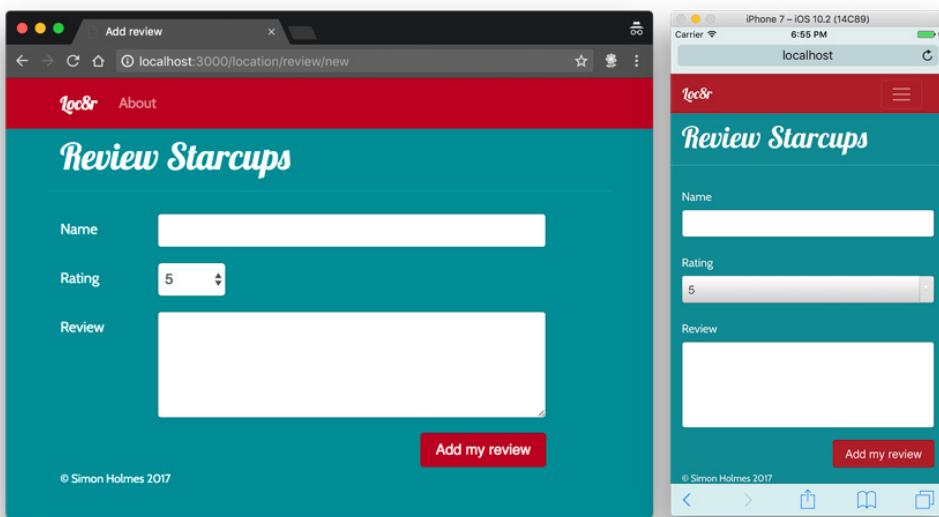


Figure 4.12 Complete Add Review page in a desktop and mobile view

The Add Review page marks the end of the user journey through the Locations collection of screens. There is just the About page left to do.

#### 4.4.3 The About page

The final page of the static prototype is the About page. This is just going to be a page with a header and some content, so nothing complicated. The layout might be useful for other pages further down the line, such as a privacy policy or terms and conditions, so we're best off creating a generic, reusable view.

The controller for the About page is in the `others.js` file in `app_server/controllers`. You're looking for the controller called `about`, and you want to change the name of the view to `generic-text`, like in the following code snippet:

```
const about = function(req, res){
  res.render('generic-text', { title: 'About' });
};
```

Next, create the view `generic-text.pug` in `app_server/views`. It's a pretty small template, and should look like the following listing.

#### **Listing 4.9 View for text only pages, app\_server/views/generic-text.pug**

```
extends layout
block content
```

```
.row.banner
  .col-12      h1= title
.row
.col-12.col-lg-8
  p
    | Loc8r was created to help people find places to sit down and get a bit of
    work done.
    | <br /><br />
    | Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc sed lorem ac
    nisi dignissim accumsan.
```

- ① Use | to create lines of plain text within a <p> tag

Listing 4.9 is a very simple layout. Don't worry about including page-specific content into a generic view at this point; we'll take that on soon and make the page reusable. For now, for the purposes of finishing the clickable static prototype, it's okay.

You'll probably want to add some additional lines in there, so that the page looks like it has real content. Notice that the lines starting with the pipe character (|) can contain HTML tags if you want them to. Figure 4.13 shows how this might look in the browser with a bit more content in it.

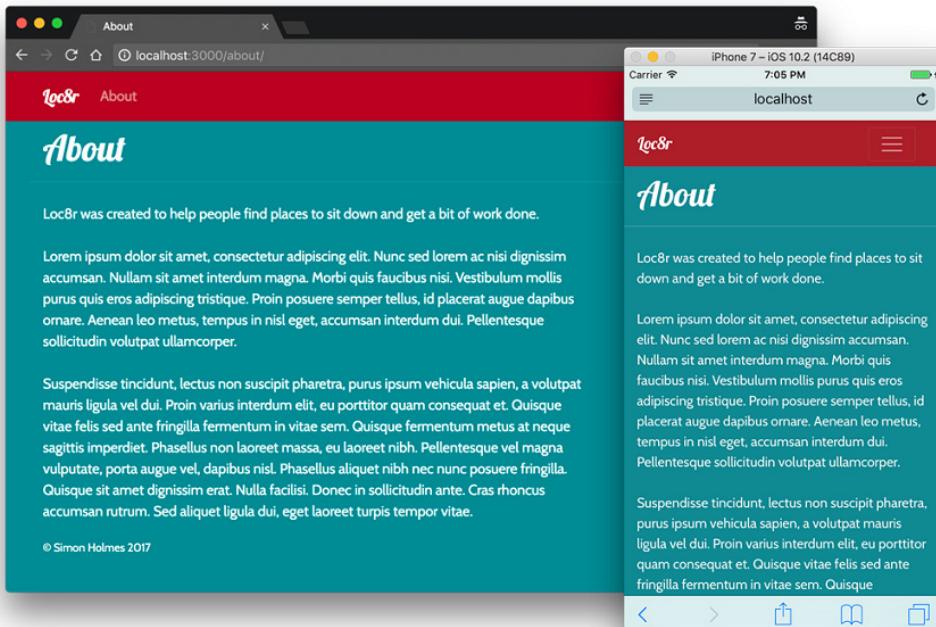


Figure 4.13 Generic text template rendering the About page

And that's the last one of the four pages we need for the static site. You can now push this up to Heroku and have people visit the URL and click around. If you've forgotten how to do this, the following code snippet shows the terminal commands you need, assuming you've already set up Heroku. In terminal, you need to be in the root folder of the application.

```
$ git add --all
$ git commit -m "Adding the view templates"
$ git push heroku master
```

### Get the source code

The source code for the application as it stands at this point is available on GitHub in the chapter-04-views branch. In a fresh folder in terminal the following commands will clone it and install the npm module dependencies:

```
$ git clone -b chapter-04-views https://github.com/simonholmes/getting-MEAN-2.git
$ cd getting-MEAN-2
$ npm install
```

So what's next? The routes, views, and controllers are set up for a static site that you can click through. And you've just pushed it up to Heroku so that others can also try it out. In some ways this is the end goal for this stage, and you can stop here while you play with the journeys and get feedback. This is definitely the easiest point in the process to make large sweeping changes.

If you were definitely going to be building an Angular SPA, and assuming you're happy with what you've done to this point, then you probably wouldn't go any further with creating a static prototype. Instead, you'd start to create an application in Angular.

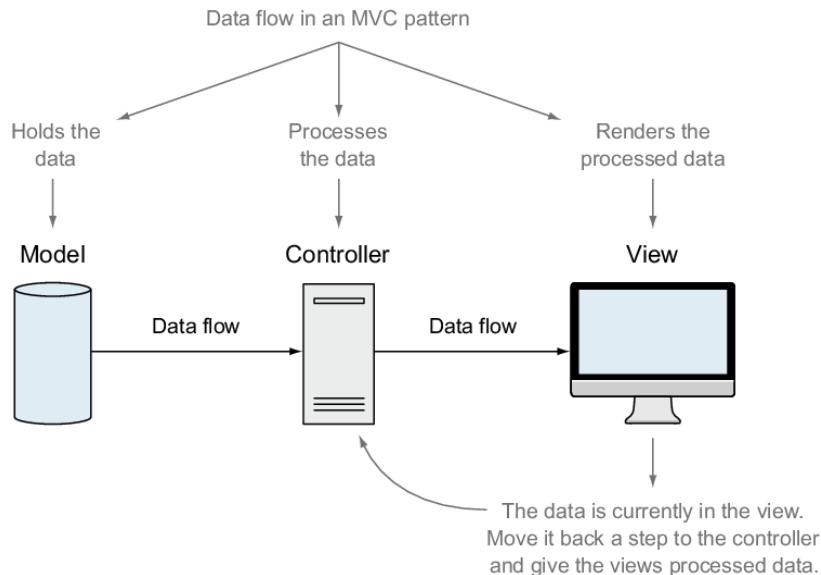
But the next step we're going to take now will continue down the road of creating the Express application. So while keeping with the static site, we'll be removing the data from the views and putting them into the controllers.

## 4.5 Take the data out of the views and make them smarter

At the moment, all of the content and data is held in the views. This is perfect for testing stuff out and moving things around, but we need to move forward. An end goal of the MVC architecture is to have views without content or data. The views should simply be fed data that they present to the end users, while being agnostic of the data they're fed. The views will need a data structure, but what is inside the data doesn't matter to the view itself.

If you think about the MVC architecture, the model holds the data, the controller then processes the data, and finally the view renders the processed data. We're not dealing with the model yet—that will come soon, starting in chapter 5. For now we're working with the views and controllers. To make the views smarter, and do what they're intended to do, we need to take the data and content out of the views and into the controllers. Figure 4.14

illustrates the data flow in an MVC architecture, and the changes we want to make to get us a step closer.



**Figure 4.14** How the data should flow in an MVC pattern, from the model through the controller to the view. At this point in the prototype our data is in the view, but we want to move it a step back into the controller.

Making these changes now will allow us to finalize the views and be ready for the next step. As a bonus, we'll start thinking about how the processed data should look in the controllers. So rather than starting with a data structure, we'll start off with the ideal front end and slowly reverse-engineer the data back through the MVC steps as our understanding of the requirements solidifies.

So how are we going to do this? Starting with the homepage we'll take every piece of content out of the Pug view. We'll update the Pug file to contain variables in place of the content, and put the content as variables into the controller. The controller can then pass these values into the view. The result should end up looking the same in the browser, and end users shouldn't be able to spot a difference. The roles of the various parts and the movement and use of data are shown in figure 4.15.

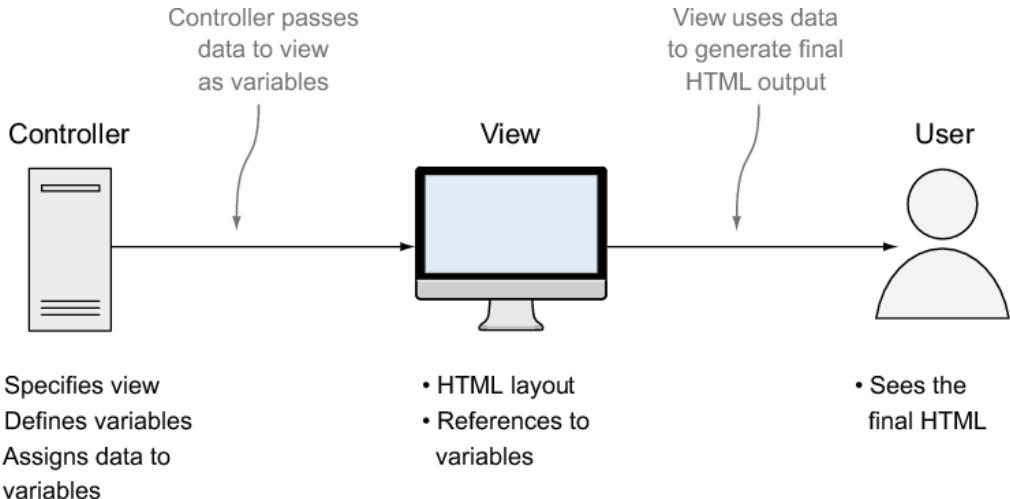


Figure 4.15 When the controller specifies the data, it passes the data to the view as variables; the view then uses that data to generate the final HTML that's delivered to the user.

At the end of this stage, the data will still be hard-coded, but in the controllers instead of the views. The views will now be smarter and able to accept and display whatever data is sent to them, providing it's in the correct format, of course.

#### 4.5.1 How to move data from the view to the controller

We're going to start with the homepage, and take the data out of the `locations-list.pug` view into the `homelist` function in the `locations.js` controllers file. Let's start at the top with something pretty simple, the page header. The following code snippet shows the page header section of the `locations-list.pug` view, which has two pieces of content:

```
.row.banner .col-12 h1 Loc8r
  small &nbsp;Find places to work with wifi near you! ②
```

- ① Large font page title
- ② Smaller font strapline for page

These two pieces of content are the first that we'll move into the controller. The homepage controller currently looks like the following:

```
const homelist = function(req, res){
  res.render('locations-list', { title: 'Home' });
};
```

This is already sending one piece of data to the view. Remember that the second parameter in the `render` function is a JavaScript object containing the data to send to the view. Here the `homelist` controller is sending the data object `{ title: 'Home' }` to the view. This is being

used by the layout file to put the string `Home` into the HTML `<title>`, which isn't necessarily the best choice of text.

### **UPDATE THE CONTROLLER**

So let's change the title to something more appropriate for the page, and also add in the two data items for the page header. Make these changes to the controller first, as follows (modifications in bold):

```
const homelist = function(req, res){
  res.render('locations-list', {
    title: 'Loc8r - find a place to work with wifi',
    pageHeader: {
      title: 'Loc8r',
      strapline: 'Find places to work with wifi near you!'
    }
  });
};
```

①  
②  
③  
④  
⑤

- ① New nested `pageHeader` object containing properties for title and strapline of page

For neatness and future manageability the title and the strapline are grouped together within a `pageHeader` object. This is a good habit to get into, and will make the controllers easier to update and maintain further down the line.

### **UPDATE THE VIEW**

Now that the controller is passing these pieces of data to the view, we can update the view to reference them in place of the hard-coded content. Nested data items like this are referenced using the dot syntax, just like you do when getting data out of nested object in JavaScript. So to reference the page header strapline in the `locations-list.pug` view we'll use `pageHeader.strapline`. The following code snippet shows the page header section of the view (modifications in bold):

```
.row.banner .col-12 h1= pageHeader.title
small &nbsp;#{pageHeader.strapline}
```

①  
②

- ① = signifies that following content is buffered code, in this case a JavaScript object  
 ② #{} delimiters are used to insert data into a specific place, like part of a piece of text

The code is outputting `pageHeader.title` and `pageHeader.strapline` in the relevant places in the view. See the following sidebar for details about the different methods of referencing data in Pug templates.

## Referencing data in Pug templates

There are two key syntaxes for referencing data in Pug templates. The first is called *interpolation*, and it's typically used to insert data into the middle of some other content. Interpolated data is defined by the opening delimiter `#{` and the end delimiter `}`. You'd normally use it like this:

```
h1 Welcome to #{pageHeader.title}
```

If your data contains HTML, this will be escaped for security reasons. This means that the end users will see any HTML tags displayed as text, and the browser will not interpret them as HTML. If you want the browser to render any HTML contained in the data you can use the following syntax:

```
h1 Welcome to !{pageHeader.title}
```

This poses potential security risks and should only be done from data sources that you trust. You shouldn't allow user inputs to display like this, for example, without some additional security checks.

The second method of outputting the data is with *buffered code*. Instead of inserting the data into a string, you build the string using JavaScript. This is done by using the `=` sign directly after the tag declaration, like this:

```
h1= "Welcome to " + pageHeader.title
```

Again, this will escape any HTML for security reasons. If you want to have unescaped HTML in your output you can use slightly different syntax like this:

```
h1!= "Welcome to " + pageHeader.title
```

Once again, be careful using this. Whenever possible you should try to use one of the escaped methods just to be on the safe side.

For this buffered code approach, you can also use JavaScript template strings like this:

```
h1= `Welcome to ${pageHeader.title}`
```

Of course, template strings will require that your version of Node can run and understand ES2015 code.

---

If you run the application now and head back to the homepage, the only change you should notice is that the `<title>` has been updated. Everything else still looks the same; it's just that some of the data is now coming from the controller.

This section served as a simple example of what we're doing at this point, and how we're doing it. The complicated part of the homepage is the listing section, so let's look at how we can approach that.

Dealing with complex, repeating data patterns

This first thing to bear in mind with the listing section is that there are multiple entries, all following the same data pattern and layout pattern. Like we've just done with the page header, we'll start with the data, taking it from the view to the controller.

In terms of JavaScript data, a repeatable pattern lends itself nicely to the idea of an array of objects. We'll look to have one array holding multiple objects, with each single object containing all of the relevant information for an individual listing.

### **ANALYZING THE DATA IN THE VIEW**

Let's take a look at a listing and see what information we need the controller to send. Figure 4.16 reminds us how a listing looks in the homepage view.

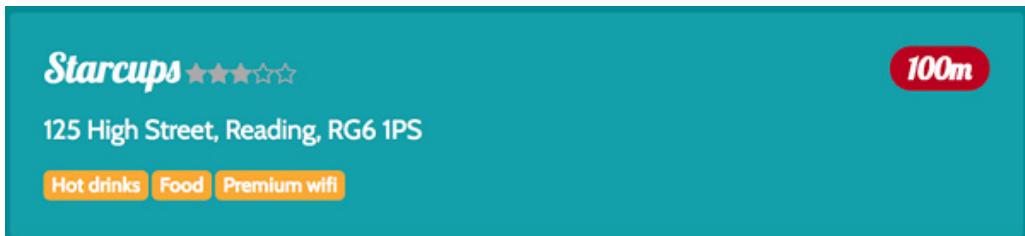


Figure 4.16 An individual listing, showing the data that we need

From this screenshot we can see that an individual listing on the homepage has the following data requirements:

- Name
- Rating
- Distance away
- Address
- List of facilities

Taking the data from the screenshot in figure 4.16 and creating a JavaScript object from it, we could come up with something simple like the following code snippet:

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

➊ List of facilities is sent as an array of string values

That's all the data needed for a single location. For multiple locations we'll need an array of these.

## ADDING THE REPEATING DATA ARRAY TO THE CONTROLLER

So we just need to create an array of the single location objects—taking the data you currently have in the view if you want—and add it to the data object passed to the `render` function in the controller. The following code snippet shows the updated `homelist` controller including the array of locations:

```
const homelist = function(req, res){
  res.render('locations-list', {
    title: 'Loc8r - find a place to work with wifi',
    pageHeader: {
      title: 'Loc8r',
      strapline: 'Find places to work with wifi near you!'
    },
    locations: [
      {
        name: 'Starcups',
        address: '125 High Street, Reading, RG6 1PS',
        rating: 3,
        facilities: ['Hot drinks', 'Food', 'Premium wifi'],
        distance: '100m'
      },
      {
        name: 'Cafe Hero',
        address: '125 High Street, Reading, RG6 1PS',
        rating: 4,
        facilities: ['Hot drinks', 'Food', 'Premium wifi'],
        distance: '200m'
      },
      {
        name: 'Burger Queen',
        address: '125 High Street, Reading, RG6 1PS',
        rating: 2,
        facilities: ['Food', 'Premium wifi'],
        distance: '250m'
      }
    ]
  });
};
```

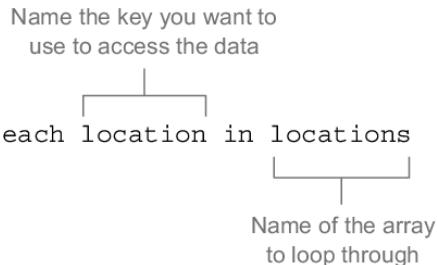
①

- ① Array of locations is being passed as locations to view for rendering

Here we've got the details for three locations being sent in the array. You can add many more, of course, but this is as good a start as any. Now we need to get the view to render this information, instead of the data currently hard-coded inside it.

## LOOPING THROUGH ARRAYS IN A PUG VIEW

The controller is sending an array to Pug as the variable `locations`. Pug offers a very simple syntax for looping through an array. In one line we specify which array to use and what variable name we want to use as the key. The key is simply a named reference to the current item in the array, so the contents of it change as the loop progresses through the array. The construct of a Pug loop is like so:



Anything nested inside this line in Pug will be iterated through for each item in the array. Let's take a look at an example of using this, using the locations data and part of the view we want. In the view file – locations-list.pug – each location starts off with the code in the following snippet, just with a different name each time:

```
.card
  .card-block
    h4
      a(href="/location") Starcups
```

We can use Pug's each/in syntax to loop through all of the locations in the `locations` array, and output the name of each. How this works is shown in the next code snippet:

```
each location in locations
  .card
    .card-block
      h4
        a(href="/location")= location.name
```

- 1
- 2
- 2
- 2
- 3

- 1 Set up loop, defining variable `location` as key
- 2 Nested items are all looped through
- 3 Output name of each location, accessing the `name` property of each location

Given the controller data we've got, with three locations in it, using that with the preceding code would result in the following HTML:

```
<div class="card">
  <div class="card-block">
    <h4>
      <a href="/location">Starcups</a>
    </h4>
  </div>
</div>
<div class="card">
  <div class="card-block">
    <h4>
      <a href="/location">Cafe Hero</a>
    </h4>
  </div>
</div>
<div class="card">
  <div class="card-block">
```

```

<h4>
  <a href="/location">Burger Queen</a>
</h4>
</div>
</div>

```

As you can see, the HTML construct—the `div`'s, `h4`, and `a` tags—are repeated three times. But the name of the location is different in each one, corresponding to the data in the controller.

So looping through arrays is pretty easy, and with that little test we've already got the first few lines of the updated view text we need. Now we just need to follow this through with the rest of the data used in the listings. We can't deal with the ratings stars like this, so we'll ignore those for now and deal with them shortly.

Dealing with the rest of the data we can produce the following code snippet, which will output all of the data for each listing. As the facilities are being passed as an array, we'll need to loop through that array for each listing:

```

each location in locations
  .card
    .card-block
      h4
        a(href="/location")= location.name
        small &nbsp;
          i.fa.fa-star
          i.fa.fa-star
          i.fa.fa-star
          i.fa.fa-star-o
          i.fa.fa-star-o
        span.badge.badge-pill.badge-default.float-right= location.distance
      p.address= location.address
      .facilities
        each facility in location.facilities
          span.badge.badge-warning= facility

```

① ②

#### ① Looping through a nested array to output facilities for each location

Looping through the facilities array is no problem, and Pug handles this with ease. Pulling out the rest of the data like the distance and the address is pretty straightforward, using the techniques we've already looked at.

The only part left to deal with is the ratings stars. For that, we're going to need a bit of inline JavaScript code.

### 4.5.2 Manipulating the data and view with code

For the star ratings the view is outputting `spans` with different classes using Font Awesome's icon system. There are a total of five stars, which are either solid or empty, depending on the rating. For example, a rating of five will show five solid stars, a rating of three will show three solid stars and two empty stars, as shown in figure 4.17, and a rating of zero will show five empty stars.



Figure 4.17 The Font Awesome star rating system in action, showing a rating of three out of five stars

To generate this type of output, we're going to use some code inside the Pug template. The code is essentially JavaScript, with some Pug-specific conventions thrown in. To add a line of inline code to a Pug template we prefix the line with a dash or hyphen. This tells Pug to run the JavaScript code rather than passing it through to the browser.

To generate the output for the stars we're going to use a couple of `for` loops. The first loop will output the correct number of solid stars and the second loop will output the remaining empty stars. The following code snippet shows how this looks and works in Pug:

```
small &nbsp;
- for (let i = 1; i <= location.rating; i++)
  i.fa.fa-star
- for (let i = location.rating; i < 5; i++)
  i.fa.fa-star-o
```

Notice that the syntax is very familiar JavaScript, but there are no curly brackets defining the block of code to run. Instead, the block of code is defined by indentation, like the rest of Pug. Also notice the mixture of code and Pug. The lines of code are saying “for every time I evaluate as true, render the indented Pug content.” This is a really nice approach, as you don't have to try to construct your HTML using JavaScript.

That's all of the content and layout for the homepage sorted, so we can move on. Except, there's one more thing we can do to improve what we've got and make some of the code reusable.

#### 4.5.3 Using includes and mixins to create reusable layout components

The star rating code that we've just written is going to be quite useful on other layouts. We're going to want it on the Details page, for example, and maybe in more places in the future. We don't want to have to manually add it to every page. What if we decide that we don't like the Font Awesome icons anymore and want to change the markup? We certainly don't want to have to change it separately on every single page that shows a rating, not if we can help it.

Fortunately, Pug enables you to create reusable components using *mixins* and *includes*.

##### **DEFINING PUG MIXINS**

A *mixin* in Pug is essentially a function. You can define a mixin at the top of your file and use it in multiple places. A mixin definition is really straightforward: you simply define the name of

the mixin, and then nest the content of it with indentation. The following code snippet shows a basic mixin definition:

```
mixin welcome
  p Welcome
```

This will output the "Welcome" text inside a `<p>` tag wherever it's invoked.

Mixins are also able to accept parameters, just like a JavaScript function. This is going to be very useful for creating the mixin we need to display the rating, as the HTML output will be different depending on the actual rating. The following code snippet shows how this can work, defining the mixin we want to use on the homepage to output the ratings stars:

```
mixin outputRating(rating)
  - for (let i = 1; i <= rating; i++)
    i.fa.fa-star
  - for (let i = rating; i < 5; i++)
    i.fa.fa-star-o
```

- ① Define mixin `outputRating` expecting a single parameter `rating`
- ② Use `rating` parameter inside for loops to output correct HTML

In a sense, this works just like a JavaScript function. When you define the mixin you can specify the parameters that it expects. Within the mixin you can then use this parameter. You can take the preceding code snippet and pop it into the top of the `locations-list.pug` file, between the `extends layout` and `block content` lines.

## CALLING PUG MIXINS

After defining the mixin, you're going to want to use it, of course. The syntax for calling a mixin is to simply place a `+` before its name. If you have no parameters, such as the `welcome` mixin we've just looked at, this looks like the following:

```
+welcome
```

This will call the `welcome` mixin and output the text "Welcome" inside a `<p>` tag.

Calling a mixin with parameters is just as easy. You simply send the values of the parameters through inside brackets, just like you'd do when calling a JavaScript function. In the `locations-list.pug` file, at the point where we're outputting the ratings, the value of the rating is held in the variable `location.rating`, as shown in the following code snippet:

```
small &nbsp;
  - for (let i = 1; i <= location.rating; i++)
    i.fa.fa-star
  - for (let i = location.rating; i < 5; i++)
    i.fa.fa-star-o
```

We can replace this code with a call to our new mixin `outputRating`, sending the `location.rating` variable as the parameter. This looks like the following code snippet:

```
small &nbsp;
+outputRating(location.rating)
```

This will now output the exact same HTML as before, but we've taken a part of the code outside of the contents of the layout. Right now, this is only reusable within the same file, but next we're going to use includes to make it accessible to other files.

### **USING INCLUDES IN PUG**

To allow our new mixin to be called from other Pug templates, we need to make it an include file. This is super easy.

Within the `app_server/views` folder, create a subfolder called `_includes` (the `_` prefix is a convention I find useful for keeping folders like this at the top). Within this folder create a new file called `sharedHTMLfunctions.pug`, and paste into it the `outputRating` mixin definition, as follows:

```
mixin outputRating(rating)
- for (let i = 1; i <= rating; i++)
  i.fa.fa-star
- for (let i = rating; i < 5; i++)
  i.fa.fa-star-o
```

Save the file, and that's your include created. To use an include file in a Pug layout there is a very simple syntax. Simply use the keyword `include`, followed by the relative path to the include file. The following code snippet shows the line of code that we can use to replace the mixin code at the top of `locations-list.pug`:

```
include _includes/sharedHTMLfunctions
```

Now, rather than having the mixin code inline in the template, we're calling it in from an include file. Notice that you can omit the `.pug` file extension when calling the include. So now, when we create a new template that needs to have ratings stars on it, we can easily reference this include file and call the `outputRating` mixin.

And now, we're really done with the homepage!

#### **4.5.4 The finished homepage**

We've made quite a lot of changes to the homepage template throughout this section. So let's see what we've ended up with. First let's take a look at the updated controller. The following listing shows the final `homelist` controller, incorporating the hard-coded data for the title, page header, sidebar and location list.

##### **Listing 4.10 The homelist controller, passing hard-coded data to the viewconst homelist =**

```
function(req, res){
```

```
res.render('locations-list', {
  title: 'Loc8r - find a place to work with wifi',
  pageHeader: {
```

1

2

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

**Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>**

```

        title: 'Loc8r',
        strapline: 'Find places to work with wifi near you!'  
2
    },
    sidebar: "Looking for wifi and a seat? Loc8r helps you find places  
to work when out and about. Perhaps with coffee, cake or a pint?  
Let Loc8r help you find the place you're looking for.",  
3
    locations: [{  
        name: 'Starcups',  
        address: '125 High Street, Reading, RG6 1PS',  
        rating: 3,  
        facilities: ['Hot drinks', 'Food', 'Premium wifi'],  
        distance: '100m'  
    }, {  
        name: 'Cafe Hero',  
        address: '125 High Street, Reading, RG6 1PS',  
        rating: 4,  
        facilities: ['Hot drinks', 'Food', 'Premium wifi'],  
        distance: '200m'  
    }, {  
        name: 'Burger Queen',  
        address: '125 High Street, Reading, RG6 1PS',  
        rating: 2,  
        facilities: ['Food', 'Premium wifi'],  
        distance: '250m'  
    }]  
});  
};  


```

- ① Update text for HTML <title>
- ② Add text for page header as two items inside an object
- ③ Add text for sidebar
- ④ Create an array of one object for each location in list

Seeing this all together you can start to appreciate where we're going with this approach. We've got a clear picture of all of the data required for the homepage of Loc8r. This is going to come in handy in the next chapter. This controller contains the text for the sidebar. We didn't talk about this step, but taking it from the view to the controller is as simple as creating a new variable for it in the controller and referencing it in the view.

Something important that we've achieved through this process is the removal of data from the view. Building the view with data in was great as a first step, as it allowed us to focus on the end-user experience without getting distracted by the technicalities. Now that we've moved the data from the view into the controller we have a much smarter, dynamic view. The view knows what pieces of data it needs, but it doesn't care what is in those pieces of data. The following listing shows the final view for the homepage.

#### **Listing 4.11 Final view for the homepage, app\_server/views/locations-list.pug**

```

extends layout
include _includes/sharedHTMLfunctions  
1
block content
  .row.banner
    .col-12

```

```

h1= pageHeader.title
    small &nbsp;#{pageHeader.strapline}          2
  .row
    .col-12.col-md-8
      each location in locations
        .card
          .card-block
            h4
              a(href="/location")= location.name
              small &nbsp;
                +outputRating(location.rating)          4
              span.badge.badge-pill.badge-default.float-right= location.distance
            p.address= location.address
            .facilities
              each facility in location.facilities
                span.badge.badge-warning= facility

    .col-12.col-md-4
      p.lead= sidebar                         5

```

- ① Bring in external include file containing outputRating mixin
- ② Output page header text using different methods
- ③ Loop through array of locations
- ④ Call outputRating mixin for each location, passing value of current location's rating
- ⑤ Reference sidebar content from controller

That's a pretty small template, right? Especially considering everything it's doing. This is a testament to the power of Pug and Bootstrap working together, combined with removing all of the content.

We're one step closer to the MVC—and general development—goal of separation of concerns. With the homepage at least.

#### 4.5.5 Updating the rest of the views and controllers

We've stepped through the process for the homepage in quite some detail here, but we're not going to spend so much time on each of the other pages. Before we can move to the next stage of development—building the data model—we need to go through the process on all of the pages, though. The end goal is to have no data in any of the views; instead the views will be smarter and the data will be hard-coded into the relevant controllers.

The process for each page will be

1. Look at the data in the view.
2. Create a structure for that data in the controller.
3. Replace the data in the view with references to the controller data.
4. Look for opportunities to reuse code.

Appendix C goes through the process for each of the three remaining pages, showing what the controller and view code should look like for each one. When you've finished, none of your views should contain any hard-coded data; the controller for each page should be passing the

required data. Figure 4.18 shows a collection of screenshots of the final pages you should have at the end of this stage.

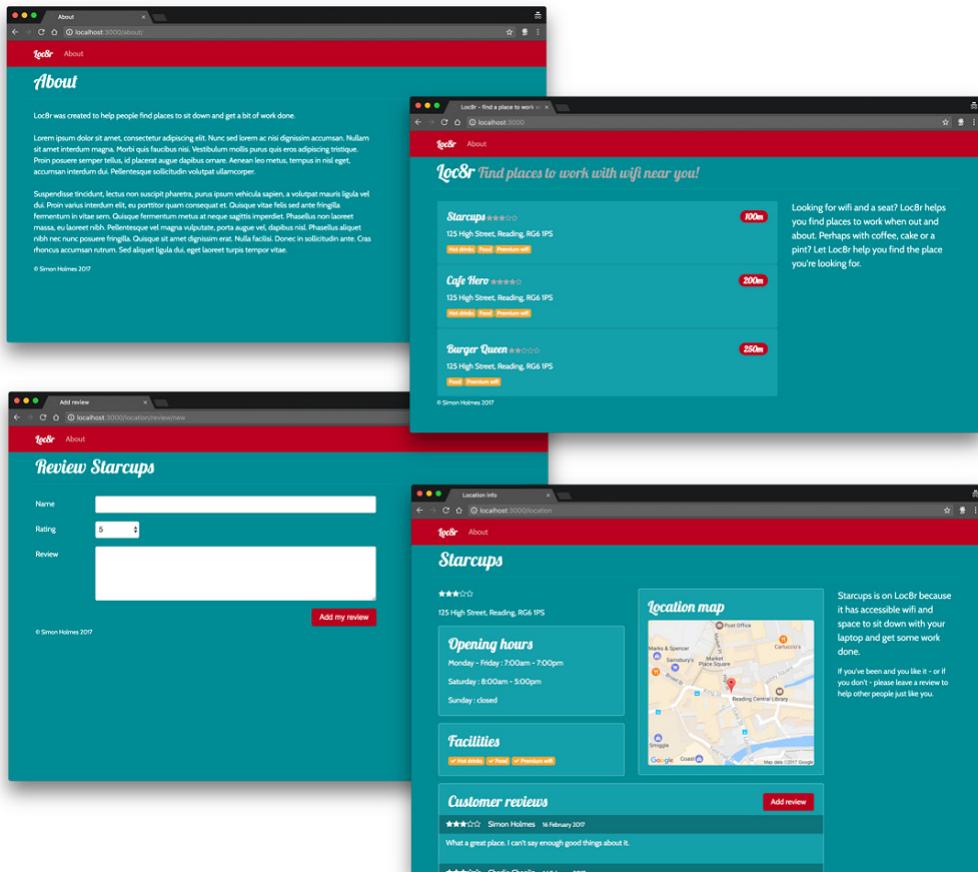


Figure 4.18 Screenshots of all four pages in the static prototype, using smart views and data hard-coded into the controllers

This puts us at the end of the first phase of our rapid prototype development, and primed to start the next phase.

### Get the source code

The source code of the application so far is available from GitHub on the chapter-04 branch of the getting-MEAN-2 repository. In a fresh folder in terminal the following commands will clone it and install the dependencies:

```
$ git clone -b chapter-04 https://github.com/simonholmes/getting-MEAN-2.git  
$ cd getting-MEAN-2  
$ npm install
```

## 4.6 Summary

In this chapter we've covered

- Defining and organizing routes in Express
- Creating Node modules to hold the controllers
- Using multiple sets of controllers with the routes
- Creating views using Pug and Bootstrap
- Making reusable Pug components, mixins
- Displaying dynamic data in Pug templates
- Passing data from controllers to views

Coming up in chapter 5 we're going to continue the journey of moving the data back up through the MVC architecture by using MongoDB and Mongoose to create a data model. That's right, it's database time!

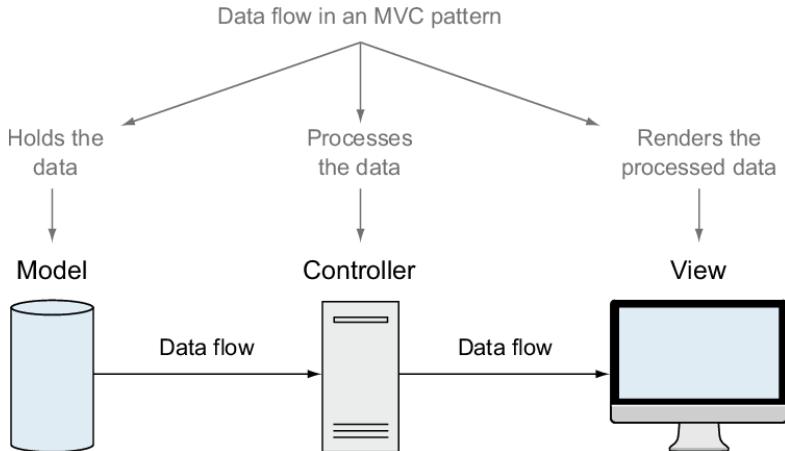
# 5

## *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.



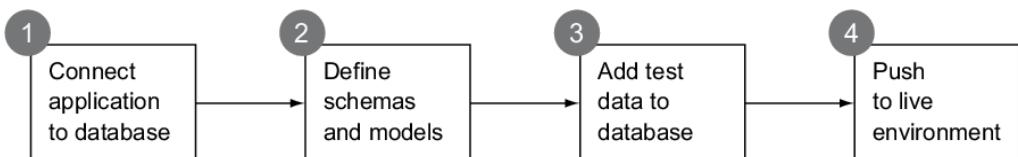
**Figure 5.1** In an MVC pattern, data is held in the model, processed by a controller, and then rendered by a view.

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.

**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-2](https://github.com/simonholmes/getting-MEAN-2). In a fresh folder in terminal the following command will clone it:

```
$ git clone -b chapter-04 https://github.com/simonholmes/getting-MEAN-2.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.

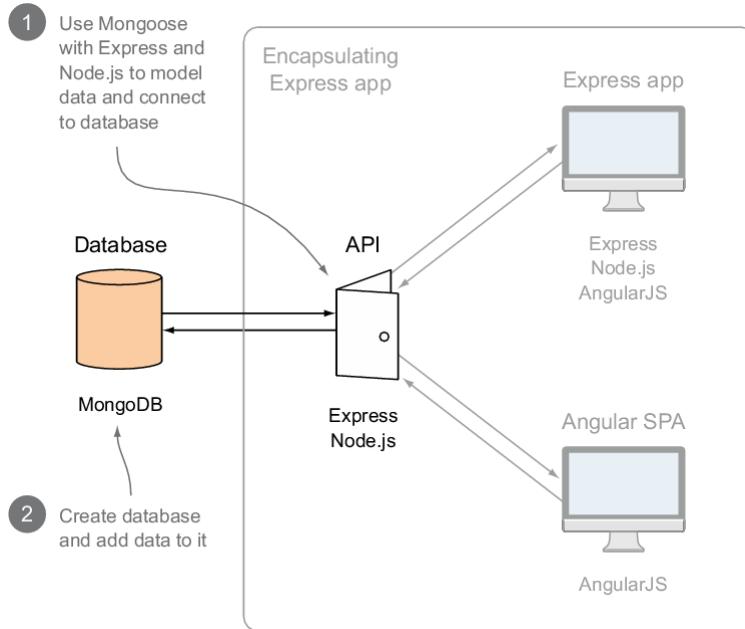


**Figure 5.2** Four main steps in this chapter, from connecting our application to a database to pushing the whole thing into a live environment

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.

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



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

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-2.git
$ cd getting-MEAN-2
$ 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.

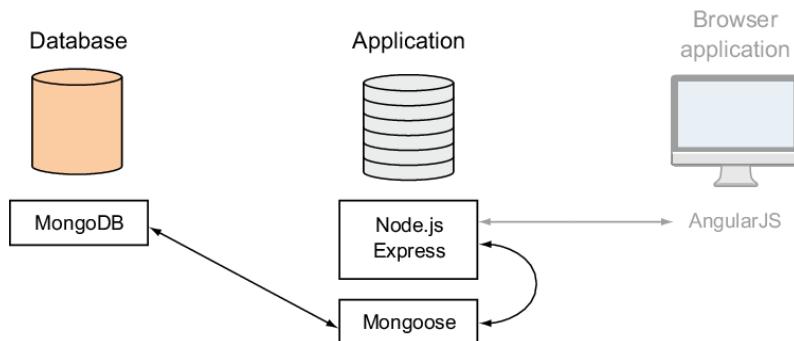


Figure 5.4 The data interactions in the MEAN stack and where Mongoose fits in. The Node/Express application interacts with MongoDB through Mongoose, and Node and Express can then also talk to Angular.

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": {
  "body-parser": "~1.15.2",
  "cookie-parser": "~1.4.3",
  "debug": "~2.2.0",
  "express": "~4.14.0",
  "morgan": "~1.7.0",
  "pug": "~2.0.0-beta6",
  "serve-favicon": "~2.3.0"
  "mongoose": "^4.9.1"
}
```

You may have slightly different version numbers, of course, but at the time of writing the latest stable version of Mongoose is 4.9.1. 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:

```
const 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):

```
const express = require('express');
const path = require('path');
const favicon = require('serve-favicon');
const logger = require('morgan');
const cookieParser = require('cookie-parser');
const 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`

```

graph LR
    A["MongoDB protocol"] --- B["Login credentials for database"]
    B --- C["Server address"]
    C --- D["Port"]
    D --- E["Database name"]
  
```

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:

```
const 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', () => {
  console.log(`Mongoose connected to ${dbURI}`);
  1
});
mongoose.connection.on('error', err => {
  console.log('Mongoose connection error:', err);
  2
});
mongoose.connection.on('disconnected', () => {
  console.log('Mongoose disconnected');
  3
});
```

- ① Monitoring for successful connection through Mongoose
- ② Checking for connection error
- ③ 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. As before, use the `npm install` command in the command line like this:

```
$ npm install --save readline
```

When that's done, in the db.js file, above the event listener code, add the following:

```
const readLine = require ('readline');
if (process.platform === 'win32'){
  const rl = readLine.createInterface ({
    input: process.stdin,
    output: process.stdout
  });
  rl.on ('SIGINT', () => {
    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:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

**Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>**

```
const gracefulShutdown = (msg, callback) => {
  mongoose.connection.close( () => {
    console.log(`Mongoose disconnected through ${msg}`);
    callback();
  });
};
```

1  
2  
3  
3

- 1 Define function to accept message and callback function
- 2 Close Mongoose connection, passing through an anonymous function to run when closed
- 3 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', () => {
  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);
  });
});
```

1  
2  
3  
4  
4  
5  
4  
4

- 1 Listen for SIGUSR2, which is what nodemon uses
- 2 Send message to gracefulShutdown and callback to kill process, emitting SIGUSR2 again
- 3 Listen for SIGINT emitted on application termination
- 4 Send message to gracefulShutdown and callback to exit Node process
- 5 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**

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

① Define database connection string and use it to open Mongoose connection

- 2 Listen for Mongoose connection events and output statuses to console
- 3 Reusable function to close Mongoose connection
- 4 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:

```
const dbURIlog = 'mongodb://localhost/Loc8rLog';
const 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', () => {                                1
  console.log(`Mongoose connected to ${dbURIlog}`);          1
});                                                               1
logDB.close( () => {                                         2
  console.log('Mongoose log disconnected');                   2
});                                                               2
① Monitoring connection event for named connection
② 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.

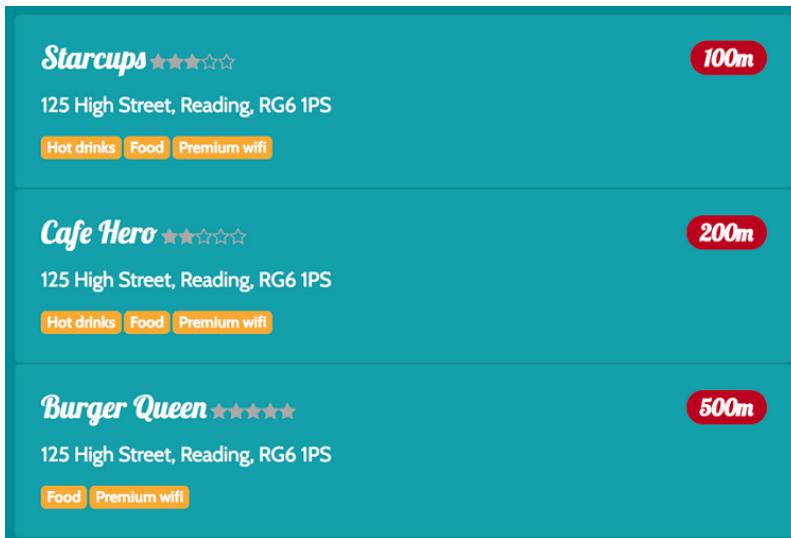
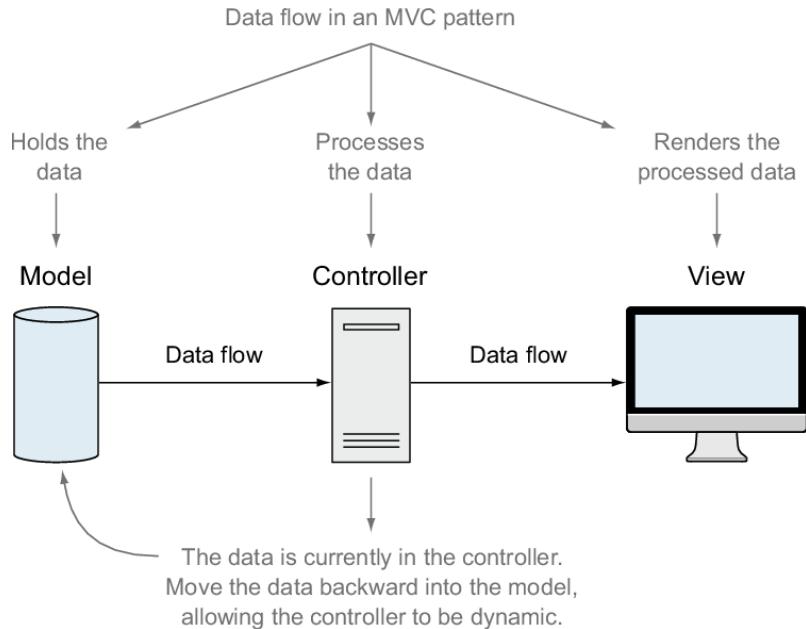


Figure 5.5 Listing section of the homepage has very defined data requirements and structure

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.



**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.

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.

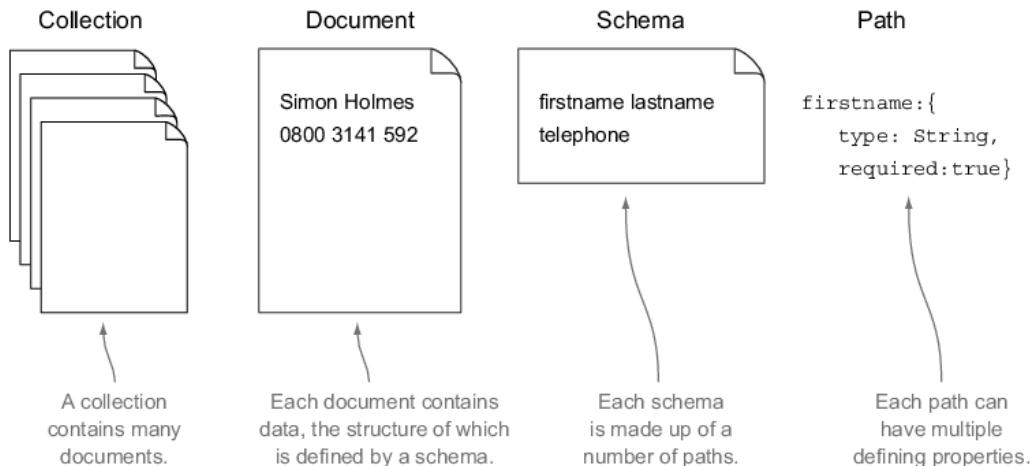
First off, let's get some naming conventions out of the way:

- 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.



**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")
}
{
  firstname : String,
  surname : String
}
```

1  
1  
1  
1  
1

2  
2  
2  
2

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>

- 1 Example MongoDB document
- 2 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
- `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:

```
const 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:

```
const 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.

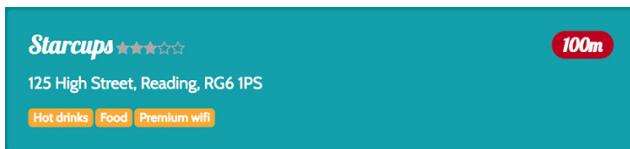


Figure 5.8 A single location as displayed on the homepage list

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

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

① name is a string
② address is another string
③ rating is a number
④ facilities is an array of strings
```

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:

```
const 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 #1. 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 `locationSchema`, 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:

```
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: [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 screenshot displays the details page for a location named "Starcups". At the top, there is a rating of ★★★☆☆ and the address "125 High Street, Reading, RG6 1PS". Below this, the "Opening hours" section lists "Monday - Friday : 7:00am - 7:00pm", "Saturday : 8:00am - 5:00pm", and "Sunday : closed". The "Facilities" section includes icons for "Hot drinks", "Food", and "Premium wifi". To the right, a "Location map" shows the area around "High St" in Reading, with landmarks like "Post Office", "Marks & Spencer", "Sainsbury's", "Market Place Square", "Carluccio's", "Abbey Square", "Reading Central Library", "Duke St", "London St", "Yield Hall Pl", "Broad St", and "Smiggle". A red pin marks the location of Starcups. Below the map, the "Customer reviews" section contains two entries: one from "Simon Holmes" dated "16 February 2017" with a review of "What a great place. I can't say enough good things about it.", and another from "Charlie Chaplin" dated "14 February 2017" with a review of "It was okay. Coffee wasn't great, but the wifi was fast.". A red "Add review" button is located in the top right corner of the reviews section.

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

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*.

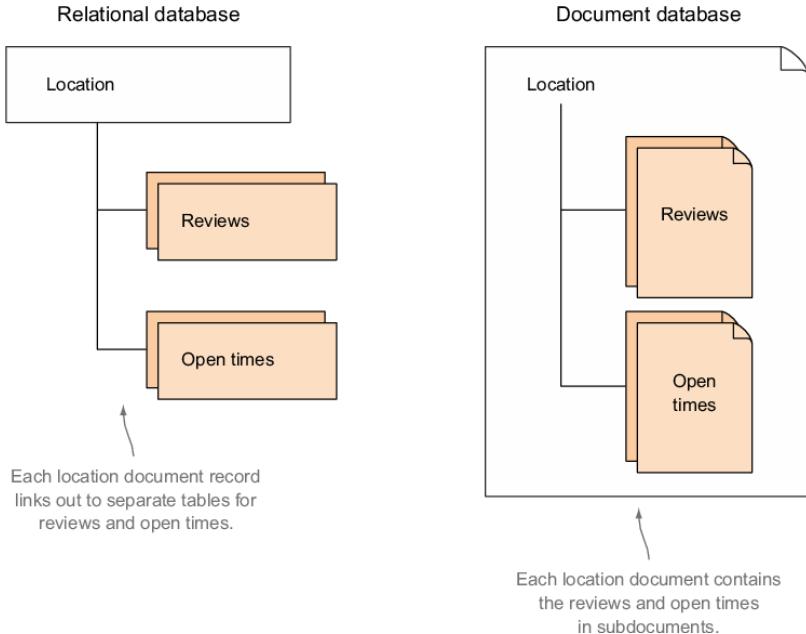
The data powering this view is currently held in the `locationInfo` controller in `app_server/controllers/locations.js`. The following listing shows the relevant portion of the data in this controller.

### **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',
      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.'③
    }
  ]
}
```

- ① Already covered with existing schema
- ② Data for opening hours is held as an array of objects
- ③ 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.



**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.

```
const 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

```

const d = new Date();
d.setHours(15);
d.setMinutes(30);      ①
① d is now Sun Mar 12 2017 15:30:40 GMT+0000 (GMT)

```

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  $(7 \times 60) + 30 = 450$ . 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`:

```

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: [Number],
  index: '2dsphere'
},
openingTimes: [openingTimeSchema] ①
});
```

① 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
    }
  ]
}
```

① 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`:

```
const 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`:

```
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: [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**

```
const mongoose = require( 'mongoose' );
const openingTimeSchema = new mongoose.Schema({
  days: {type: String, required: true},  
  1  
  opening: String,  
  closing: String,  
  closed: {  
    2  
    2  
    2  
    2  
  }  
});  
  2
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

Licensed to Jacob Munkholm Hansen <au518331@uni.au.dk>

```

        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}
});
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: [Number],
        index: '2dsphere'
    },
    openingTimes: [openingTimeSchema],
    reviews: [reviewSchema]
});

```

- ① Require Mongoose so that we can use its methods
- ② Define a schema for opening times
- ③ Define a schema for reviews
- ④ Start main location schema definition
- ⑤ Use 2dsphere to add support for GeoJSON longitude and latitude coordinate pairs
- ⑥ 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],          ①
"openingTimes": [{
  "_id": ObjectId("52ef3a9f79c44a86710fe7f6"), ②
  "days": "Monday - Friday",                  ②
  "opening": "7:00am",                        ②
  "closing": "7:00pm",                        ②
  "closed": false                           ②
}, {
  "_id": ObjectId("52ef3a9f79c44a86710fe7f7"), ②
  "days": "Saturday",                       ②
  "opening": "8:00am",                        ②
  "closing": "5:00pm",                        ②
  "closed": false                           ②
}, {
  "_id": ObjectId("52ef3a9f79c44a86710fe7f8"), ②
  "days": "Sunday",                          ②
  "closed": true                           ②
}],                                         ②
"reviews": [{
  "_id": ObjectId("52ef3a9f79c44a86710fe7f9"), ③
  "author": "Simon Holmes",                 ③
  "rating": 5,                            ③
  "createdOn": ISODate("2013-07-15T23:00:00Z"), ③
  "reviewText": "What a great place. I can't say enough good
things about it."                         ③
}, {
  "_id": ObjectId("52ef3a9f79c44a86710fe7fa"), ③
  "author": "Charlie Chaplin",              ③
  "rating": 3,                            ③
  "createdOn": ISODate("2013-06-15T23:00:00Z"), ③
  "reviewText": "It was okay. Coffee wasn't great, but the wifi was fast." ③
}]                                         ③
}

```

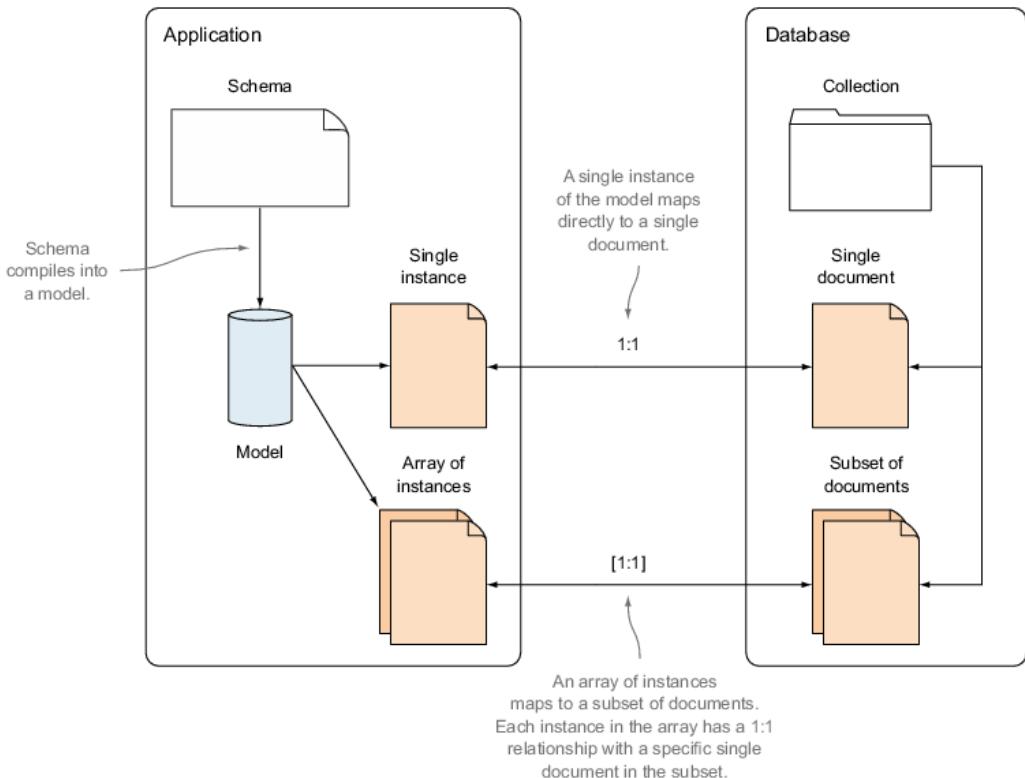
- ① Coordinates are stored as a GeoJSON pair [longitude, latitude]
- ② Opening times are stored as nested array of objects—these are subdocuments
- ③ Reviews are also array of subdocuments

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.



**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.

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');
  └─────────┘   └─────────┘   └─────────┘   └─────────┘
  Connection    The name of   The schema   MongoDB collection
  name          the model      to use       name (optional)
```

**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 compiled 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 few lines, confirming:

1. The shell version
2. The server and port that it's connecting to
3. The server version it has connected to

These confirmation lines should look like this:

```
MongoDB shell version v3.4.2
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.2
```

If you're using an older version of MongoDB you might see different messages, but it's normally obvious if it has worked or failed. You might also see a few lines starting with `Server` has startup warnings going on to state `Access control is not enabled for the database.` This isn't anything to worry about on your local development machine.

**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:

```
admin      0.000GB
local      0.000GB
```

### 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 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)
```



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, and you won't even see it if you run `show dbs`. But you will be able to see it after saving some data to it

## 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({
  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
    }
  ]
})
```

①

① Note collection name specified as part of `save` command

In one step this will have created the `Loc8r` database, a new `locations` collection, and also the first document within 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. For example:

```
> show dbs
Loc8r      0.000GB
admin      0.000GB
```

```
local    0.000GB
```

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

```
> show collections
locations
```

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,
}
```

- ① Remember to run the `find` operation on collection itself
- ② 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 prettyify 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("Mar 12, 2017"),
      reviewText: "What a great place."
    }
  }
})
```

- ① Start with query object to find correct document
- ② When document is found, push a subdocument into the reviews path
- ③ 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.

You may have noticed that here we are specifying the `_id` property and assigning it the value of `ObjectId()`. MongoDB doesn't automatically add `_id` to subdocuments like it does documents, but it will be very useful for us later. Giving it the value of `ObjectId()` tell MongoDB to create a new unique identifier for this review subdocument.

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 mLab, 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 mLab, but the following parts—migrating the data and setting the connection strings in the Node application—are platform-specific.

### 5.5.1 Setting up mLab 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 mLab 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 mLab. 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 mLab will not incur any charges. If you're not comfortable doing this, check out the following sidebar for setting up mLab manually.

---

#### Setting up mLab 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 mLab documentation to guide you through this: <http://docs.mlab.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.mlab.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 MLAB_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 `MLAB_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 MLAB TO THE HEROKU APPLICATION**

The quickest way to add mLab as a Heroku add-on is through terminal. Make sure you're in the root folder of your application and run the following command (note that here we need to use mLab's old name *MongoLab*):

```
$ heroku addons:create 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 MONGODB_URI
```

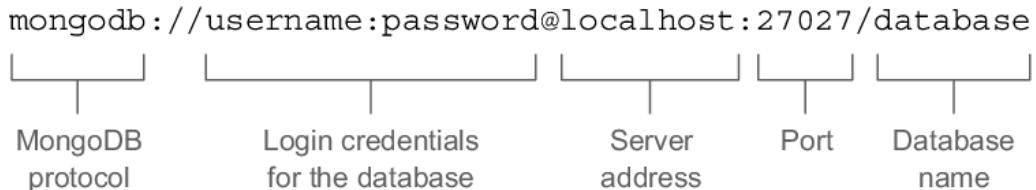
This will output the full connection string, which looks something like this:

```
mongodb://heroku_t0zs37gc:1k3t3pg08sb5enovqd9sk314gj@ds159330.mlab.com:59330/heroku_t0zs37gc
```

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:



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

- Username: `heroku_t0zs37gc`
- Password: `1k3t3pg08sb5enovqd9sk314gj`
- Server address: `ds159330.mlab.com`
- Port: `59330`
- Database name: `heroku_t0zs37gc`

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. Navigate to a directory on your machine that is suitable to hold a 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.

#### **NAVIGATE TO A SUITABLE DIRECTORY**

When we run the data dump command from the command line it will create a folder called `/dump` in the current directory and place the data dump inside it. The first step then is to navigate in terminal to a suitable location on your hard drive - your home directory or documents folder will do, or you can create a specific folder if you prefer.

## 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 can accept many parameters, of which we need these two:

- `-h` —The host server (and port)
- `-d` —The database name

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
```

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 ds159330.mlab.com:59330 -d heroku_t0zs37gc -u heroku_t0zs37gc -p  
1k3t3pg08sb5enovqd9sk314gj dump/
```

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:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>

```
$ 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 ds159330.mlab.com:59330/heroku_t0zs37gc -u heroku_t0zs37gc -p  
1k3t3pg08sb5enovqd9sk314gj
```

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.

### 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 mLab (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.

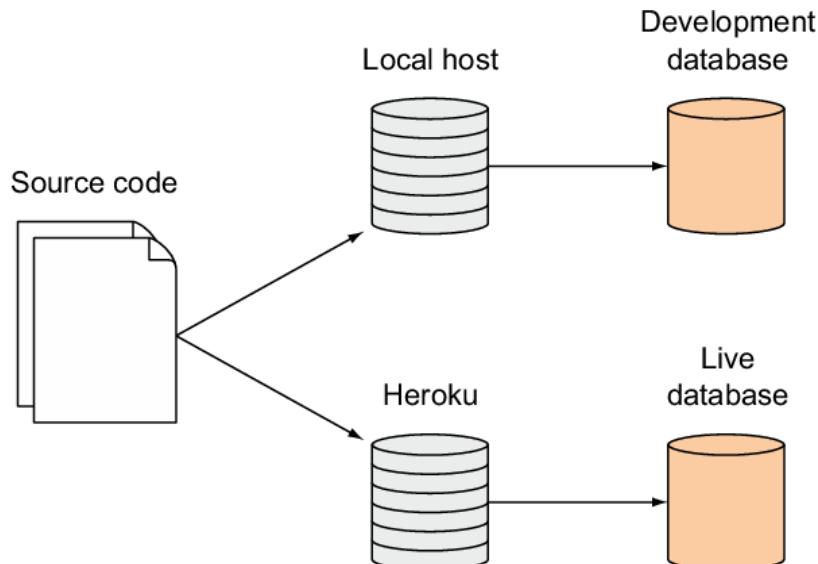


Figure 5.12 The source code runs in two locations, each of which needs to connect to a different database.

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 validate this setting by using a `get` version of this command, like so:

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

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:

```
const 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.

```
let dbURI = 'mongodb://localhost/Loc8r';
if (process.env.NODE_ENV === 'production') {
  dbURI =
    'mongodb://heroku_t0zs37gc:1k3t3pg08sb5enovqd9sk314gj@ds159330.mlab.com:59330/heroku_t0zs37gc';
}
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 mLab on Heroku you automatically have one set up—it's how we originally got access to the connection string (if you set your mLab 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:

```
let dbURI = 'mongodb://localhost/Loc8r';
if (process.env.NODE_ENV === 'production') {
  dbURI = process.env.MONGODB_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.

To do this we need to add both the `NODE_ENV` and `MONGODB_URI` environment variables in front of the `nodemon` command like this (note that all of the following should be entered as one line):

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

Now our console log on startup should look like this:

```
Mongoose connected to
```

```
mongodb://heroku_t0zs37gc:1k3t3pg08sb5enovqd9sk314gj@ds159330.mlab.com:59330/heroku_t0zs37gc
```

When running this command 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.

### 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 --all
$ 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:

```
2017-04-14T07:01:22.066997+00:00 app[web.1]: Mongoose connected to
      mongodb://heroku_t0zs37gc:1k3t3pg08sb5enovqd9sk314gj@ds159330.mlab.com:59330/heroku_t0zs37gc
```

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-2 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-2.git
$ cd getting-MEAN-2
$ 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.

# 6

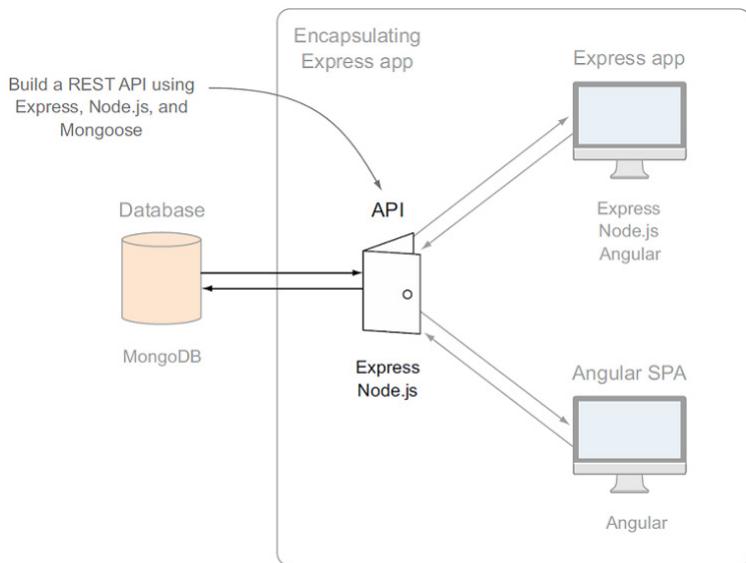
## *Writing a REST API: Exposing the MongoDB database to the application*

### This chapter covers

- Rules of REST APIs
- API patterns
- Typical CRUD functions (create, read, update, delete)
- Using Express and Mongoose to interact with MongoDB
- Testing API endpoints

As we come in to this chapter we have a MongoDB database set up, but we can only interact with it through the MongoDB shell. During this chapter we'll build a REST API so that we can interact with our database through HTTP calls and perform the common CRUD functions: create, read, update, and delete.

We'll mainly be working with Node and Express, using Mongoose to help with the interactions. Figure 6.1 shows where this chapter fits into the overall architecture.



**Figure 6.1** This chapter will focus on building the API that will interact with the database, exposing an interface for the applications to talk to.

We'll start off by looking at the rules of a REST API. We'll discuss the importance of defining the URL structure properly, 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. Once we have that knowledge under our belts we'll move on to building our API for Loc8r, covering all of the typical CRUD operations. As we go, we'll discuss a lot about Mongoose, and get into some Node programming and more Express routing.

**NOTE** If you haven't yet built the application from chapter 5, you can get the code from GitHub on the chapter-05 branch at [github.com/simonholmes/getting-MEAN-2](https://github.com/simonholmes/getting-MEAN-2). 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-2.git
$ cd getting-MEAN-2
$ npm install
```

## 6.1 The rules of a REST API

Let's start with a recap of what a REST API is. From chapter 2 you may remember:

- 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 each other.

So a REST API is a stateless interface to your application. 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.

REST APIs have an associated set of standards. While you don't have to stick to these for your own API it's generally best to, as it means that any API you create will follow the same approach. It also means you're used to doing things in the "right" way if you decide you're going to make your API public.

In basic terms a REST API takes an incoming HTTP request, does some processing, and always sends back an HTTP response, as shown in figure 6.2.

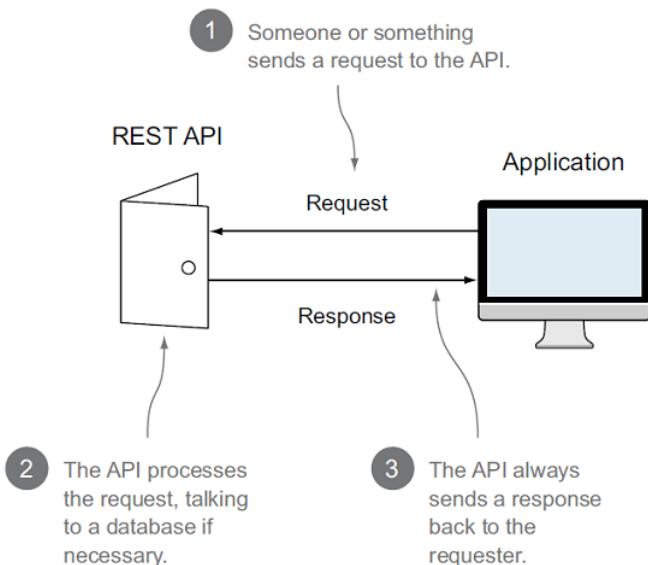


Figure 6.2 A REST API takes incoming HTTP requests, does some processing, and returns HTTP responses.

The standards that we're going to follow for Loc8r revolve around the requests and the responses.

### 6.1.1 Request URLs

Request URLs for a REST API have a simple standard. Following this standard will make your API easy to pick up, use, and maintain.

The way to approach this is to start thinking about the collections in your database, as you'll typically have a set of API URLs for each collection. You may also have a set of URLs for

each set of subdocuments. Each URL in a set will have the same basic path, and some may have additional parameters.

Within a set of URLs you need to cover a number of actions, generally based around the standard CRUD operations. The common actions you'll likely want are

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

Using Loc8r as an example, the database has a Locations collection that we want to interact with. Table 6.1 shows how the URLs and parameters might look for this collection-.

**Table 6.1 URL paths and parameters for an API to the Locations collection; all have the same base path, and several have the same location ID parameter**

Action	URL path	Parameters	Example
Create new location	/locations		<a href="http://loc8r.com/api/locations">http://loc8r.com/api/locations</a>
Read list of locations	/locations		<a href="http://loc8r.com/api/locations">http://loc8r.com/api/locations</a>
Read a specific location	/locations	locationid	<a href="http://loc8r.com/api/locations/123">http://loc8r.com/api/locations/123</a>
Update a specific location	/locations	locationid	<a href="http://loc8r.com/api/locations/123">http://loc8r.com/api/locations/123</a>
Delete a specific location	/locations	locationid	<a href="http://loc8r.com/api/locations/123">http://loc8r.com/api/locations/123</a>

As you can see from table 6.1, each action has the same URL path, and three of them expect the same parameter to specify a location. This poses a very obvious question: How do you use the same URL to initiate different actions? The answer lies in request methods.

### 6.1.2 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—this is the method used when you enter a URL into the address bar of your browser. Another common method is POST, often used when submitting form data.

Table 6.2 shows the methods we'll be using in our API, their typical use cases, and what you'd expect returned.

**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

The four HTTP methods that we'll be using are POST, GET, PUT, and DELETE. If you look at the first word in the "Use" column you'll notice that there's a different method for each of the four CRUD operations.

**TIP** Each of the four CRUD operations uses a different request method.

The method is important, because a well-designed REST API will often have the same URL for different actions. In these cases it's the method that tells the server which type of operation to perform. We'll discuss how to build and organize the routes for this in Express later in this chapter.

So if we take the paths and parameters and map across the appropriate request method we can put together a plan for our API, as shown in table 6.3.

**Table 6.3 Request method is used to link the URL to the desired action, enabling the API to use the same URL for different actions**

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

Table 6.3 shows the paths and methods we'll use for the requests to interact with the location data. As there are five actions but only two different URL patterns, we can use the request methods to get the desired results.

Loc8r only has one collection right now, so this is our starting point. But the documents in the Locations collection do have reviews as subdocuments, so let's quickly map those out too.

## API URLs FOR SUBDOCUMENTS

Subdocuments are treated in a similar way, but require an additional parameter. Each request will need to specify the ID of the location, and some will also need to specify the ID of a review. Table 6.4 shows the list of actions and their associated methods, URL paths, and parameters.

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

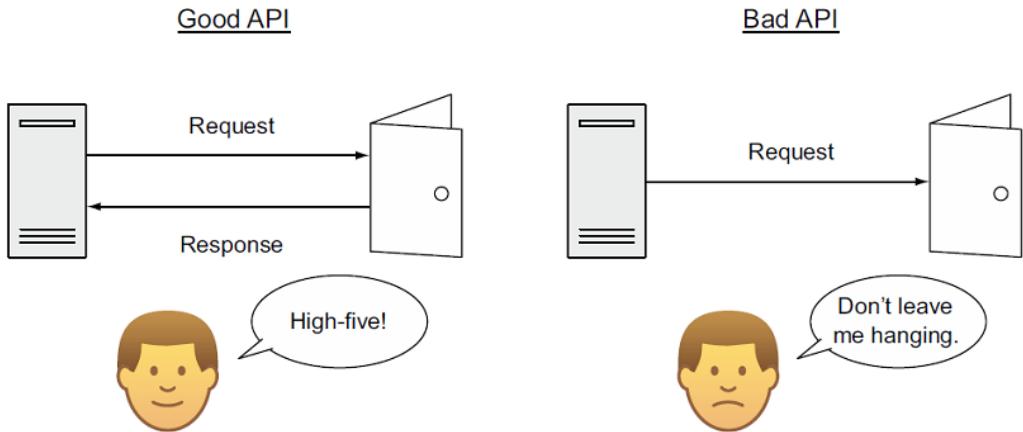
Action	Method	URL path	Parameters	Example
Create new review	POST	/locations/locationid/reviews	locationid	<a href="http://loc8r.com/api/locations/123/reviews">http://loc8r.com/api/locations/123/reviews</a>
Read a specific review	GET	/locations/locationid/reviews	locationid reviewid	<a href="http://loc8r.com/api/locations/123/reviews/abc">http://loc8r.com/api/loc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

You may have noticed that for the subdocuments we don't have a "read a list of reviews" action. This is because we'll be retrieving the list of reviews as part of the main document. The preceding tables should give you an idea of how to create basic API request specifications. The URLs, parameters, and actions will be different from one application to the next, but the approach should remain consistent.

That's requests covered. The other half of the flow, before we get stuck in some code, is responses.

### 6.1.3 Responses and status codes

A good API is like a good friend. If you go for a high-five a good friend will not leave you hanging. The same goes for a good API. If you make a request, a good API will always respond and not leave you hanging. Every single API request should return a response. This contrast is shown in figure 6.3.



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

For a successful REST API, standardizing the responses is just as important as standardizing 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 correctly should give the requester all of the information required to continue.

### ***RETURNING DATA FROM AN API***

Your API should return a consistent data format. Typical formats for a REST API are XML and/or JSON. We'll be using JSON for our 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 up the response times and efficiency of an API by reducing the bandwidth required.

Our 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

During this chapter we'll discuss how to do all of these things as we build the Loc8r API. As well as responding with data, any REST API should return the correct HTTP status code.

### ***USING HTTP STATUS CODES***

A good REST API should return the correct HTTP status code. The status code most people are familiar with is 404, which is what is returned by a web server when a user requests a page

that can't be found. This is probably the most prevalent error code on the internet, but there are dozens of other codes relating to client errors, server errors, redirections, and successful requests. Table 6.5 shows the 10 most popular HTTP status codes and where they might be useful when building an API.

**Table 6.5 Most popular HTTP status codes and how they might be used when sending 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 already exists with the same data
500	Internal server error	Problem with your server or the database server

As we go through this chapter and build the Loc8r API we'll make use of several of these status codes, while also returning the appropriate data.

## 6.2 Setting up the API in Express

We've already got a good idea about the actions we want our API to perform, and the URL paths needed to do so. As we know from chapter 4, to get Express to do something based on an incoming URL request we need to set up controllers and routes. The controllers will do the action, and the routes will map the incoming requests to the appropriate controllers.

We have files for routes and controllers already set up in the application, so we could use those. A better option, though, is to keep the API code separate so that we don't run the risk of confusion and complication in our application. In fact, this is one of the reasons for creating an API in the first place. Also, by keeping the API code separate it makes it easier to strip it out and put it into a separate application at a future point, should you choose to do so. We really do want easy decoupling here.

So the first thing we want to do here is create a separate area inside the application for the files that will create the API. At the top level of the application *create a new folder called*

`app_api`. If you've been following along and building up the application as you go, this will sit alongside the `app_server` folder.

This folder will hold everything specific to the API: routes, controllers, and models. When you've got this all set up we'll have a look at some ways to test these API placeholders.

### 6.2.1 Creating the routes

Like we did with the routes for the main Express application, we'll have an `index.js` file in the `app_api/routes` folder that will hold all of the routes we'll use in the API. Let's start by referencing this file in the main application file `app.js`.

#### **INCLUDING THE ROUTES IN THE APPLICATION**

The first step is to tell our application that we're adding more routes to look out for, and when it should use them. We already have a line in `app.js` to require the server application routes, which we can simply duplicate and set the path to the API routes as follows:

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

You may also have a line in `app.js` that still brings the example `user` routes - you can go ahead and delete this now if so, because we don't need it. Next, we need to tell the application when to use the routes. We currently have the following line in `app.js` telling the application to check the server application routes for all incoming requests:

```
app.use('/', routes);
```

Notice the `'/'` as the first parameter. This enables us to specify a subset of URLs for which the routes will apply. For example, we'll define all of our API routes starting with `/api/`. By adding the line shown in the following code snippet we can tell the application to use the API routes only when the route starts with `/api`:

```
app.use('/', routes);
app.use('/api', apiRoutes);
```

As before, you can delete the similar line for `user` routes if it's there. Okay, let's set up these URLs.

#### **SPECIFYING THE REQUEST METHODS IN THE ROUTES**

Up to now we've only used the `GET` method in the routes, like in the following code snippet from our main application routes:

```
router.get('/location', ctrlLocations.locationInfo);
```

Using the other methods of `POST`, `PUT`, and `DELETE` is as simple as switching out the `get` with the respective keywords of `post`, `put`, and `delete`. The following code snippet shows an example using the `POST` method for creating a new location:

```
router.post('/locations', ctrlLocations.locationsCreate);
```

Note that we don't specify `/api` at the front of the path. We specify in `app.js` that these routes should only be used if the path starts with `/api`, so it's assumed that all routes specified in this file will be prefixed with `/api`.

### **SPECIFYING REQUIRED URL PARAMETERS**

It's quite common for API URLs to contain parameters for identifying specific documents or subdocuments—locations and reviews in the case of Loc8r. Specifying these parameters in routes is really simple; you just prefix the name of the parameter with a colon when defining each route.

Say 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
```

With a path like this Express will only match URLs that match that pattern. So a location ID must be specified and must be in the URL between `locations/` and `/reviews`, and a review ID must also be specified at the end of the URL. When a path like this is assigned to a controller the parameters will be available to use in the code, with the names specified in the path, `locationid` and `reviewid` in this case.

We'll review exactly how you get to them in just a moment, but first we need to set up the routes for our Loc8r API.

### **DEFINING THE LOC8R API ROUTES**

Now we know how to set up routes to accept parameters, and we also know what actions, methods, and paths we want to have in our API. So we can combine all of this to create the route definitions for the Loc8r API.

If you haven't done so yet, you should create an `index.js` file in the `app_api/routes` folder. To keep the size of individual files under control we'll separate the locations and reviews controllers into different files.

We'll also make use of a slightly different way of defining routes in Express, which is ideal for managing multiple different methods on a single route. With this approach, you define the route first and then chain on the different HTTP methods. This really streamlines route definitions, making them much easier to read.

The following listing shows how the defined routes should look.

**Listing 6.1 Routes defined in app\_api/routes/locations.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)          1
    .post(ctrlLocations.locationsCreate);                 1

  router
    .route('/locations/:locationid')
      .get(ctrlLocations.locationsReadOne)                2
      .put(ctrlLocations.locationsUpdateOne)              2
      .delete(ctrlLocations.locationsDeleteOne);           2

// reviews
router
  .route('/locations/:locationid/reviews')
    .post(ctrlReviews.reviewsCreate);                   3

  router
    .route('/locations/:locationid/reviews/:reviewid')
      .get(ctrlReviews.reviewsReadOne)                  3
      .put(ctrlReviews.reviewsUpdateOne)                3
      .delete(ctrlReviews.reviewsDeleteOne);             3

module.exports = router;

```

- ① Include controller files (we'll create these next)
- ② Define routes for locations
- ③ Define routes for reviews
- ④ Export routes

In this router file we need to `require` the related controller files. We haven't created these controller files yet, and will do so in just a moment. This is a good way to approach it, because by defining all of the routes and declaring the associated controller functions here we develop a high-level view of what controllers are needed.

The application now has two sets of routes: the main Express application routes and the new API routes. The application won't start at the moment though, because none of the controllers referenced by the API routes exist.

### **6.2.2 Creating the controller placeholders**

To enable the application to start we can create placeholder functions for the controllers. These functions won't really do anything, but they will stop the application from falling over while we're building the API functionality.

The first step, of course, is to create the controller files. We know where these should be and what they should be called because we've already declared them in the `app_api/routes`

folder. We 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, like in the following code snippet.

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

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

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

To test the routing and the functions, though, we'll need to return a response.

### **RETURNING JSON FROM AN EXPRESS REQUEST**

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

```
res
  .status(status)
  .json(content);
```

①  
②  
③

- ① Use the Express response object
- ② Send response status code, such as 200
- ③ Send 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 = function (req, res) {
  res
    .status(200)
    .json({ "status" : "success" });
};
```

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

#### **6.2.3 Including the model**

It's vitally important that the API can talk to the database; without it the API isn't going to be much use! To do this with Mongoose, we first need to require Mongoose into the controller

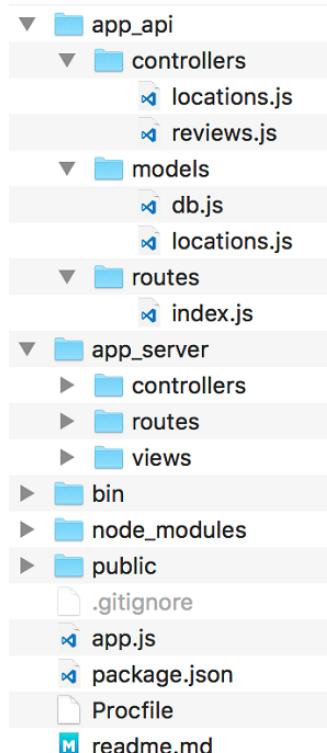
files, and then bring in the Location model. Right at the top of the controller files, above all of the placeholder functions, add the following two lines:

```
const mongoose = require('mongoose');
const Loc = mongoose.model('Location');
```

The first line gives the controllers access to the database connection, and the second brings in the Location model so that we can interact with the Locations collection.

If we take a look at the file structure of our application, we see the app\_api/models folder containing the database connection and the Mongoose setup is inside the app\_server folder. But it's the API that's dealing with the database, not the main Express application. If the two applications were separate the model would be kept part of the API, so that's where it should live.

Just move the app\_api/models folder from the app\_server folder into the app\_api folder, giving the folder structure like that shown in figure 6.4.



**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

We need to tell the application that we've moved the app\_api/models folder, of course, so we need to update the line in app.js that requires the model to point to the correct place:

```
require('./app_api/models/db');
```

With that done, the application should start again and still connect to your database. The next question is, how can we test the API?

#### 6.2.4 Testing the API

You can quickly test the GET routes in your browser 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.

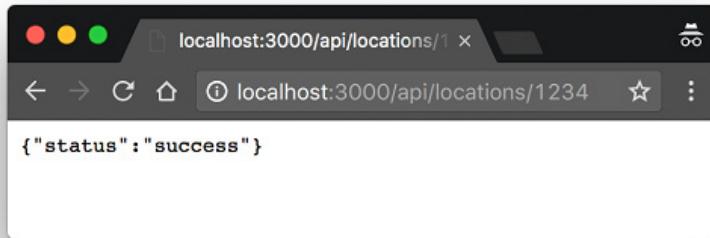
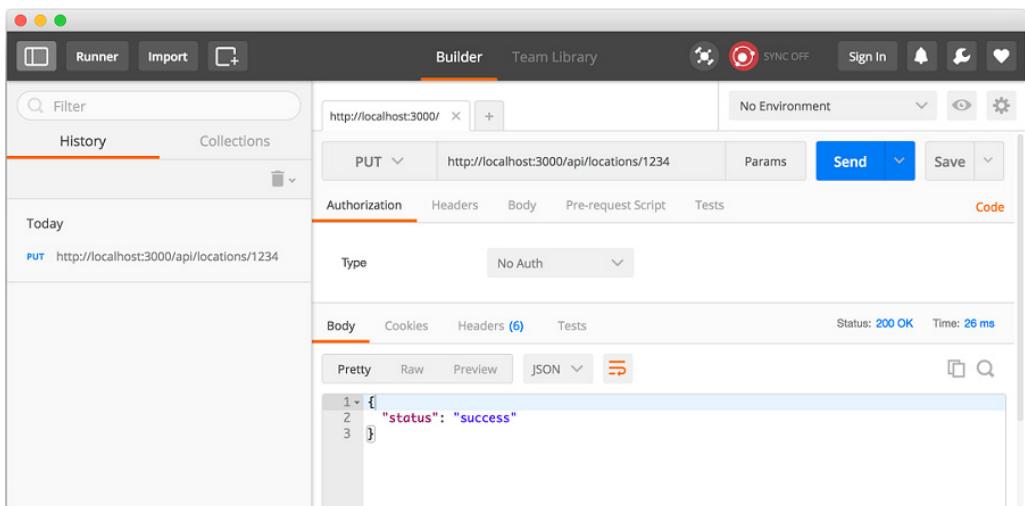


Figure 6.5 Testing a GET request of the API in the browser

This is okay for testing GET requests, but it doesn't get you very far with POST, PUT, and DELETE methods. There are a few tools to help you test API calls like this, but my current favorite is a free application called Postman REST Client; this is available as a standalone application or browser extension.

Postman enables you to test API URLs with a number of different request methods, allowing you to specify additional query string parameters or form data. After you click the Send button it will make a request to the URL you've specified and display the response data and status code.

Figure 6.6 shows a screenshot of Postman making a PUT request to the same URL as before.



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

It's a good idea to get Postman or another REST client up and running now. You'll need to use one a lot during this chapter as we build up a REST API. Let's get started on the workings of the API by using the GET requests to read data from MongoDB.

### 6.3 GET methods: Reading data from MongoDB

GET methods are all about querying the database and returning some data. In our routes for Loc8r we have three GET requests doing different things, as listed in table 6.6.

**Table 6.6** Three GET requests of the Loc8r API

Action	Method	URL path	Parameters	Example
Read list of locations	GET	/locations		<a href="http://loc8r.com/api/locations">http://loc8r.com/api/locations</a>
Read a specific location	GET	/locations	locationid	<a href="http://loc8r.com/api/locations/123">http://loc8r.com/api/locations/123</a>
Read a specific review	GET	/locations/locationid/reviews	locationid reviewid	<a href="http://loc8r.com/api/locations/123/reviews/abc">http://loc8r.com/api/locations/123/reviews/abc</a>

We'll look at how to find a single location first, because it provides a good introduction to the way Mongoose works. Next we'll locate a single document using an ID, and then we'll expand into searching for multiple documents.

### 6.3.1 Finding a single document in MongoDB using Mongoose

Mongoose interacts with the database through its models, which is why we imported the Locations model as `Loc` at the top of the controller files. A Mongoose model has several associated methods to help manage the interactions as noted in the following sidebar.

#### **Mongoose query methods**

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

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

We'll use some of these but not all of them in this book.

For finding a single database document with a known ID in MongoDB, Mongoose has the `findById` method.

#### **APPLYING THE FINDBYID METHOD TO THE MODEL**

The `findById` method is relatively straightforward, accepting a single parameter, the ID to look for. As it's a model method, it's applied to the model like this:

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

This will not start the database query operation; it just tells the model what the query will be. To start the database query, Mongoose models have an `exec` method.

#### **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. As it's a callback function the names of these parameters can be whatever you like.

The methods can be chained as follows:

```
Loc
  .findById(locationid)
  .exec((err, location) => {
    console.log("findById complete");
  });
  1
  2
  3
```

- 1 Apply `findById` method to `Location` model using `Loc`
- 2 Execute query
- 3 Log message when complete

This approach ensures that the database interaction is asynchronous, and therefore doesn't block the main Node process.

### **USING THE FINDBYID METHOD IN A CONTROLLER**

The controller we're working with to find a single location by ID is `locationsReadOne`, in the `locations.js` file in `app_api/controllers`.

We know the basic construct of the operation: apply the `findById` and `exec` methods to the `Location` model. To get this working in the context of the controller we need to do two things:

- Get the `locationid` parameter from the URL and pass it to the `findById` method.
- Provide an output function to the `exec` method.

Express makes it really easy to get the URL parameters we defined in the routes. The parameters are held inside a `params` object attached to the `request` object. With our route being defined like so

```
router
  .route('/api/locations/:locationid')
```

we can access the `locationid` parameter from inside the controller like this:

```
req.params.locationid
```

For the output function we can use a simple callback that sends the found locations as a JSON response. Putting this all together gives us the following:

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

- 1
- 2
- 3
- 3

- 1 Get `locationid` from URL parameters and give it to `findById` method
- 2 Define callback to accept possible parameters
- 3 Send document found as a JSON response with HTTP status 200

And now we have a very basic API controller. You can try it out by getting the ID of one of the locations in MongoDB and going to the URL in your browser, or by calling it in Postman. To get one of the ID values you can run the command `db.locations.find()` in the Mongo shell and it will list all of the locations you have, which will each include the `_id` value. When you've put

the URL together the output should be a full location object as stored in MongoDB; you should see something like figure 6.7.

```
{
  "_id": "58eb9d2e65891c331a84a554",
  "name": "Starcups",
  "address": "125 High Street, Reading, RG6 1PS",
  "coords": [-0.9690884, 51.455041],
  "reviews": [
    {
      "author": "Simon Holmes",
      "_id": "58ec721865891c331a84a555",
      "rating": 5,
      "timestamp": "2017-04-10T23:00:00.000Z",
      "reviewText": "What a great place.",
      "createdOn": "2017-05-02T06:11:15.135Z"
    }
  ],
  "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
    }
  ],
  "facilities": ["Hot drinks", "Food", "Premium wifi"],
  "rating": 3
}
```

**Figure 6.7 Basic controller for finding a single location by ID returns a JSON object to the browser if the ID is found**

Did you try out the basic controller? Did you put an invalid location ID into the URL? If you did you'll have seen that you got nothing back. No warning, no message, just a 200 status telling you that everything is okay, but no data returned.

### CATCHING ERRORS

The problem with that basic controller is that it only outputs a success response, regardless of whether it was successful or not. This isn't good behavior for an API. A good API should respond with an error code when something goes wrong.

To respond with error messages the controller needs to be set up to trap potential errors and send an appropriate response. Error trapping in this fashion typically involves `if` statements. Every `if` statement must either have a corresponding `else` statement, or it must include a `return` statement.

**TIP** Your API code must never leave a request unanswered.

With our basic controller there are three errors we need to trap:

- 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. Bearing this in mind the final code for the controller to find and return a single location looks like the following listing.

### **Listing 6.2 locationsReadOne controller**

```
const locationsReadOne = function(req, res) {
  if (req.params && req.params.locationid) {          ①
    Loc
      .findById(req.params.locationid)
      .exec((err, location) => {
        if (!location) {                                ②
          res
            .status(404)
            .json({
              "message": "locationid not found"
            });
          return;
        } else if (err) {                               ③
          res
            .status(404)
            .json(err);
          return;
        }
        res
          .status(200)
          .json(location);
      });
  } else {                                         ⑤
    res
      .status(404)
      .json({
        "message": "No locationid in request"
      });
  }
};
```

- ① Error trap 1: check that locationid exists in request parameters
- ② Error trap 2: if Mongoose doesn't return a location, send 404 message and exit function scope using return statement
- ③ Error trap 3: if Mongoose returned an error, send it as 404 response and exit controller using return statement
- ④ If Mongoose didn't error, continue as before and send location object in a 200 response
- ⑤ If request parameters didn't include locationid, send appropriate 404 response

Listing 6.2 uses both of the two methods of trapping with `if` statements. Error trap 1 ① uses an `if` to check that the `params` object exists in the request object, and that the `params` object contains a `locationid` value. This loop is closed off with an `else` ⑤ for when either the `params` object or the `locationid` value isn't found. Error trap 2 ② and error trap 3 ③ both use an `if` to check for an error returned by Mongoose. Each `if` includes a `return` statement, which will prevent any following code in the callback scope from running. If no error was found the `return` statement is ignored and the code moves on to send the successful response ④.

Each of these traps provides a response for success and failure, leaving no room for the API to leave a requester hanging. If you wish you can also throw in a few `console.log`

statements so that it's easier to track what's going on in terminal; the source code in GitHub will have some.

Figure 6.8 shows the difference between a successful request and a failed request, using the Postman extension in Chrome.

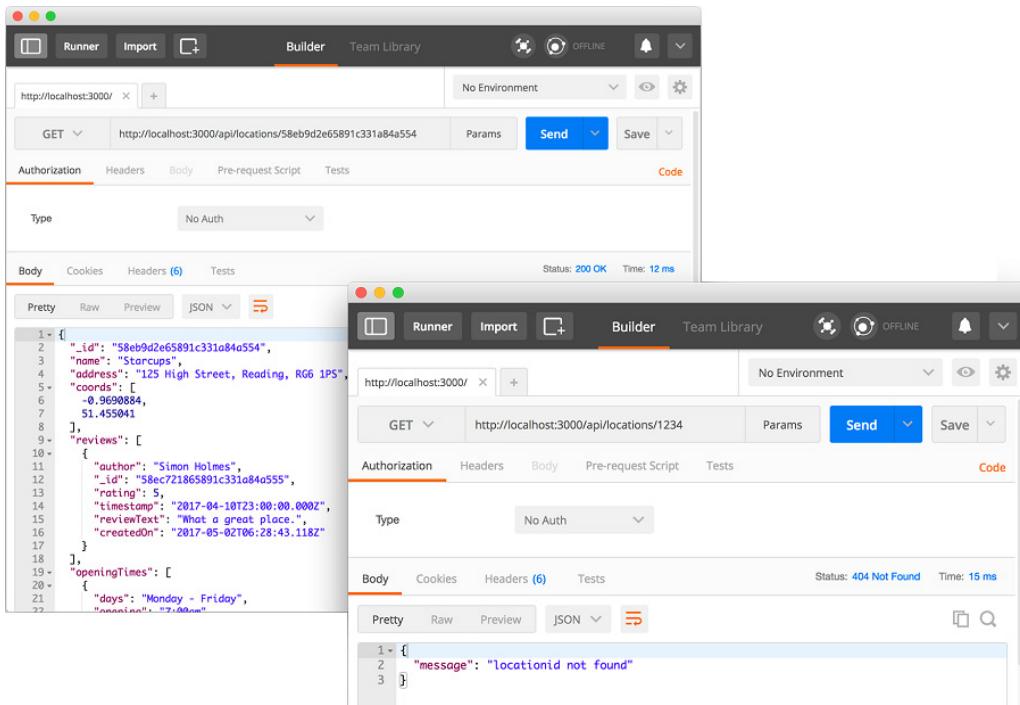


Figure 6.8 Testing successful (left) and failed (right) API responses using Postman

That's one complete API route dealt with. Now it's time to look at the second GET request to return a single review.

### 6.3.2 Finding a single subdocument based on IDs

To find a subdocument you first have to find the parent document like we've just done to find a single location by its ID. Once you've found the document you can look for a specific subdocument.

This means that we can take the `locationsReadOne` controller as the starting point and add a few modifications to create the `reviewsReadOne` controller. These modifications are

- Accept and use an additional `reviewid` URL parameter.
- Select only the name and reviews from the document, rather than having MongoDB

- return the entire document.
- Look for a review with a matching ID.
- Return the appropriate JSON response.

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

### LIMITING THE PATHS RETURNED FROM MONGODB

When you retrieve a document from MongoDB you don't always need the full document; sometimes you just want some specific data. Limiting the data being passed around is also better for bandwidth consumption and speed.

Mongoose does this through a `select` method chained to the model query. For example, the following code snippet will tell MongoDB that we only want to get the name and the reviews of a location:

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

The `select` method accepts a space-separated string of the paths we want to retrieve.

### 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 will return 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);      ①
  }
);
```

**①** Pass `reviewid` from parameters into `id` method

In this code snippet a single review would be returned to the `review` variable in the callback.

### ADDING SOME ERROR TRAPPING AND PUTTING IT ALL TOGETHER

Now we've got the ingredients needed to make the `reviewsReadOne` controller. Starting with a copy of the `locationsReadOne` controller we can make the modifications required to return just a single review.

The following listing shows the `reviewsReadOne` controller in `review.js` (modifications in bold).

**Listing 6.3 Controller for finding a single review**

```

const reviewsReadOne = function(req, res) {
  if (req.params && req.params.locationid && req.params.reviewid) {    ①
    Loc
      .findById(req.params.locationid)
      .select('name reviews')                                ②
      .exec((err, location) => {
        if (!location) {
          res
            .status(404)
            .json({
              "message": "locationid not found"
            });
          return;
        } else if (err) {
          res
            .status(400)
            .json(err);
          return;
        }
        if (location.reviews && location.reviews.length > 0) {      ③
          const review = location.reviews.id(req.params.reviewid);   ④
          if (!review) {                                              ⑤
            res
              .status(404)
              .json({
                "message": "reviewid not found"
              });
          } else {                                                 ⑥
            response = {
              location : {
                name : location.name,
                id : req.params.locationid
              },
              review : review
            };
            res
              .status(200)
              .json(response);
          }
        } else {                                                 ⑦
          res
            .status(404)
            .json({
              "message": "No reviews found"
            });
        }
      });
  } else {
    res
      .status(404)
      .json({
        "message": "Not found, locationid and reviewid are both required"
      });
  }
}

```

```
};
```

- ① Verify that reviewid exists as a parameter
- ② Add Mongoose select method to model query, stating that we want to get name of location and its reviews
- ③ Check that returned location has reviews
- ④ Use Mongoose subdocument .id method as a helper for searching for matching ID
- ⑤ If review isn't found return an appropriate response
- ⑥ If review is found build response object returning review and location name and ID
- ⑦ If no reviews are found return an appropriate error message

When this is saved and ready you can test it using Postman again. You need to have correct ID values, which you can get from the Postman query we made to check for a single location or directly from MongoDB via the Mongo shell. The Mongo command `db.locations.find()` will return all of the locations and their reviews. Remember that the URL is in the structure:

```
/locations/:locationid/reviews/:reviewed
```

With this you can also test what happens if you put in a false ID for a location or a review, or try a review ID from a different location.

### 6.3.3 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 query methods to help find nearby places.

Here we'll use the Mongoose method `geoNear` to find a list of locations close to a specified point, up to a specified maximum distance. `geoNear` is a model method that accepts three parameters:

- A geoJSON geographical point
- An options object
- A callback function

The following code snippet shows the basic construct:

```
Loc.geoNear(point, options, callback);
```

Unlike the `findById` method, `geoNear` doesn't have an `exec` method. Instead, `geoNear` is executed immediately and the code to run on completion is sent through in the callback.

#### **CONSTRUCTING A GEOJSON POINT**

The first parameter of the `geoNear` method is a geoJSON point. A geoJSON point is a simple JSON object containing a latitude and a longitude in an array. The construct for a geoJSON point is shown in the following code snippet:

```
var point = {  
    type: "Point",  
    coordinates: [lng, lat]  
};
```

- ①
- ②
- ③

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

Licensed to Jacob Munkholm Hansen <au518331@uni.au.dk>

- 1 Declare object
- 2 Define it as type "Point"
- 3 Set longitude and latitude coordinates in an array, longitude first

The route set up here to get a list of locations doesn't have the coordinates in the URL parameters, meaning that they'll have to be specified in a different way. A query string is ideal for this data type, meaning that the request URL will look more like this:

```
api/locations?lng=-0.7992599&lat=51.378091
```

Express, of course, gives you access to the values in a query string, putting them into a `query` object attached to the `request` object—for example, `req.query.lng`. The longitude and latitude values will be strings when retrieved, but they need to be added to the `point` object as numbers. JavaScript's `parseFloat` function can see to this. Putting it all together, the following code snippet shows how to get the coordinates from the query string and create the geoJSON point required by the `geoNear` function:

```
const locationsListByDistance = function(req, res) {
  const lng = parseFloat(req.query.lng);          1
  const lat = parseFloat(req.query.lat);          1
  const point = {                                2
    type: "Point",                            2
    coordinates: [lng, lat]                  2
  };                                              2
  Loc.geoNear(point, options, callback);        3
};
```

- 1 Get coordinates from query string and convert from strings to numbers
- 2 Create geoJSON point
- 3 Send point as first parameter in geoNear method

Naturally, this controller will not work yet as `options` and `callback` are both currently undefined. We'll work on these now, starting with the `options`.

### **ADDING REQUIRED QUERY OPTIONS TO GEONEAR**

The `geoNear` method only has one required option: `spherical`. This determines whether the search will be done based on a spherical object or a flat plane. It's generally accepted these days that Earth is round, so we'll set the `spherical` option to be `true`.

In creating an object to hold the options we have the following code snippet:

```
const geoOptions = {
  spherical: true
};
```

Now the search will be based on coordinates on a sphere.

## LIMITING GEONEAR RESULTS BY NUMBER

You'll often want to look after the API server—and the responsiveness seen by end users—by limiting the number of results when returning a list. In the `geoNear` method adding an option called `num` does this. You simply specify the maximum number of results you want to have returned.

The following code snippet shows this added to the previous `geoOptions` object, limiting the size of the returned data set to 10 objects:

```
const geoOptions = {
  spherical: true,
  num: 10
};
```

Now the search will bring back no more than the 10 closest results.

## LIMITING GEONEAR RESULTS BY DISTANCE

When returning location-based data, another way to keep the processing of the API under control is to limit the list of results by distance from the central point. This is just a case of adding another option called `maxDistance`. When using the `spherical` option, MongoDB actually does the calculations in meters for us, making life very simple. This wasn't always the case: older versions of MongoDB used radians, which made things much more complicated.

If you want to output in miles, you'll need to do a little calculation but we'll stick to meters and kilometers. We'll put a limit of 20km, which is 20000m. We can now add the `maxDistance` value to the options, and add these options to the controller as follows:

```
const locationsListByDistance = function(req, res) {
  const lng = parseFloat(req.query.lng);
  const lat = parseFloat(req.query.lat);
  const point = {
    type: "Point",
    coordinates: [lng, lat]
  };
  const geoOptions = {1
    spherical: true,1
    maxDistance: 20000,1
    num: 101
  };
  Loc.geoNear(point, geoOptions, callback);2
};
```

- ① Create options object, including setting maximum distance to 20 km
- ② Update `geoNear` function to use `geoOptions` object

### Extra credit

Try taking the maximum distance from a query string value instead of hard-coding it into the function. The code on GitHub for this chapter has the answer to this.

That's the last of the options we need for our `geoNear` database search, so now it's time to start working with the output.

### **LOOKING AT THE GEONEAR OUTPUT**

The completion callback for the `geoNear` method has three parameters, in this order:

1. An error object
2. A results object
3. A stats object

With a successful query the error object will be undefined, the results object will contain an array of results, and the stats object will contain information about the query, like time taken, number of documents scanned, the average distance, and the maximum distance of the documents returned. We'll start by working with a successful query before adding in the error trapping.

Following a successful `geoNear` query MongoDB returns an array of objects. Each object contains a distance value and a returned document from the database. In other words, MongoDB doesn't add the distance to the data. The following code snippet shows an example of the returned data, truncated for brevity:

```
[{
  dis: 687.5430052130049,
  obj: {
    name: 'Starcups',
    address: '125 High Street, Reading, RG6 1PS'
  }
}]
```

This array only has one object, but a successful query is likely to have several objects returned at once. The `geoNear` method actually returns the entire document in the `obj` object. There are two problems here:

- The API shouldn't return more data than necessary.
- We want to return the distance as an integral part of the returned data set.

So rather than simply sending the returned data back as the response there's some processing to do first.

### **PROCESSING THE GEONEAR OUTPUT**

Before the API can send a response we need to make sure it's sending the right thing, and only what's needed. We know what data is needed by the homepage listing as we've already built the homepage controller in `app_server/controllers/location.js`. The `homelist` function sends a number of location objects like the following example:

```
{  
  name: 'Starcups',  
  address: '125 High Street, Reading, RG6 1PS',
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

**Licensed to Jacob Munkholm Hansen <au518331@uni.au.dk>**

```

    rating: 3,
    facilities: ['Hot drinks', 'Food', 'Premium wifi'],
    distance: '100m'
}

```

To create an object along these lines from the results, we simply need to loop through the results and push the relevant data into a new array. This processed data can then be returned with a status 200 response. The following code snippet shows how this might look:

```

Loc.geoNear(point, options, (err, results, stats) => {
  let locations = [];
  results.forEach((doc) => {
    locations.push({
      distance: (doc.dis) ①
      name: doc.obj.name, ②
      address: doc.obj.address, ③
      rating: doc.obj.rating, ④
      facilities: doc.obj.facilities, ④
      _id: doc.obj._id ④
    });
  });
  res
    .status(200)
    .json(locations); ⑤
});

```

- ① Create new array to hold processed results data
- ② Loop through geoNear query results
- ③ Get distance and convert from radians to kilometers, using helper function previously created
- ④ Push rest of required data into return object
- ⑤ Send processed data back as a JSON response

If you test this API route with Postman—remembering to add longitude and latitude coordinates to the query string—you'll see something like figure 6.10.

### **Extra credit**

Try passing the results to an external named function to build the list of locations. This function should return the processed list, which can then be passed into the JSON response.

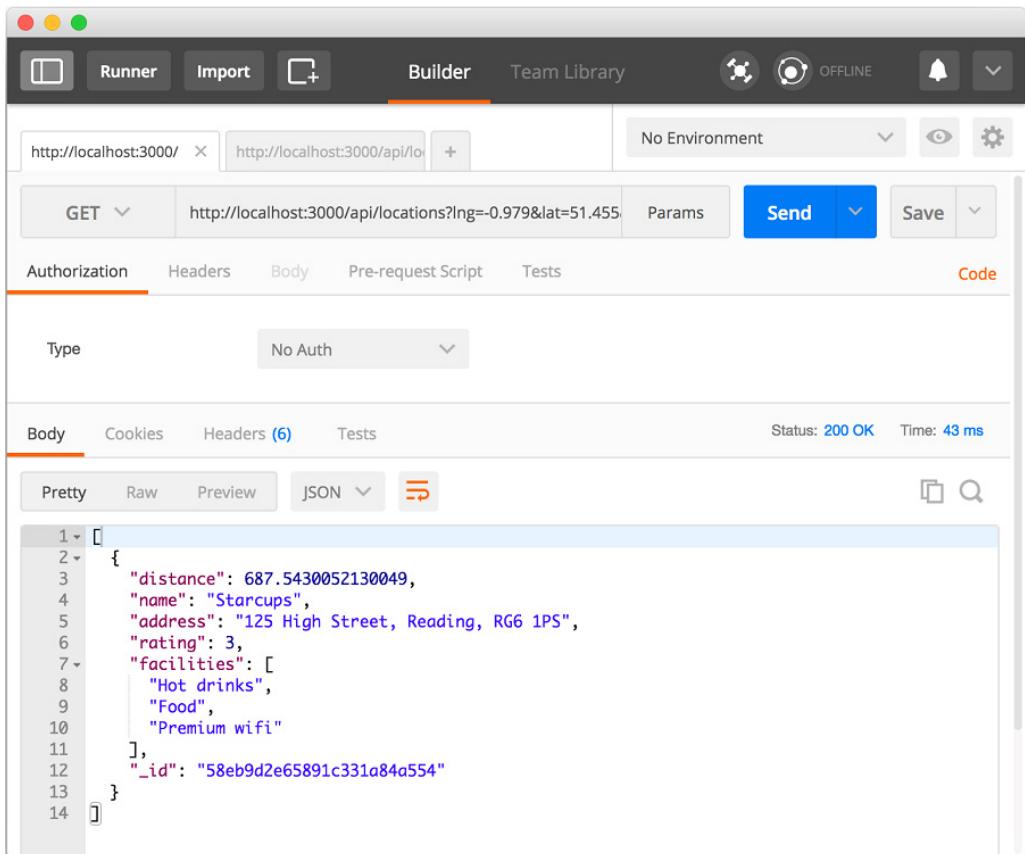


Figure 6.9 Testing the location list route in Postman should give a 200 status and a list of results, depending on the geographical coordinates sent in the query string.

If you test this by sending coordinates too far away from the test data you should still get a 200 status, but the returned array will be empty.

### **ADDING THE ERROR TRAPPING**

Once again we've started by building the success functionality, and now we need to add in some error traps to make sure that the API always sends the appropriate response.

The traps we need to set should check that

- The parameters have all been sent correctly.
- The `geoNear` function hasn't returned an error.

The listing on the next page shows the final controller all put together, including these error traps.

#### **Listing 6.4 Locations list controller locationsListByDistance**

```
const locationsListByDistance = function(req, res) {
  const lng = parseFloat(req.query.lng);
  const lat = parseFloat(req.query.lat);
  const point = {
    type: "Point",
    coordinates: [lng, lat]
  };
  const geoOptions = {
    spherical: true,
    maxDistance: 20000,
    num: 10
  };
  if (!lng || !lat) {
    res
      .status(404)
      .json({
        "message": "lng and lat query parameters are required"
      });
    return;
  }
  Loc.geoNear(point, geoOptions, (err, results, stats) => {
    let locations = [];
    if (err) {
      res
        .status(404)
        .json(err);
    } else {
      results.forEach((doc) => {
        locations.push({
          distance: doc.dis,
          name: doc.obj.name,
          address: doc.obj.address,
          rating: doc.obj.rating,
          facilities: doc.obj.facilities,
          _id: doc.obj._id
        });
      });
      res
        .status(200)
        .json(locations);
    }
  });
};
```

- ① Check lng and lat query parameters exist in right format; return a 404 error and message if not
- ② If geoNear query returns error, send this as response with 404 status

This completes the GET requests that our API needs to service, so moving forward it's time to tackle the POST requests.

## 6.4 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. In the routes for Loc8r we have two POST requests doing different things, listed in Table 6.7.

**Table 6.7 Two POST requests of the Loc8r API**

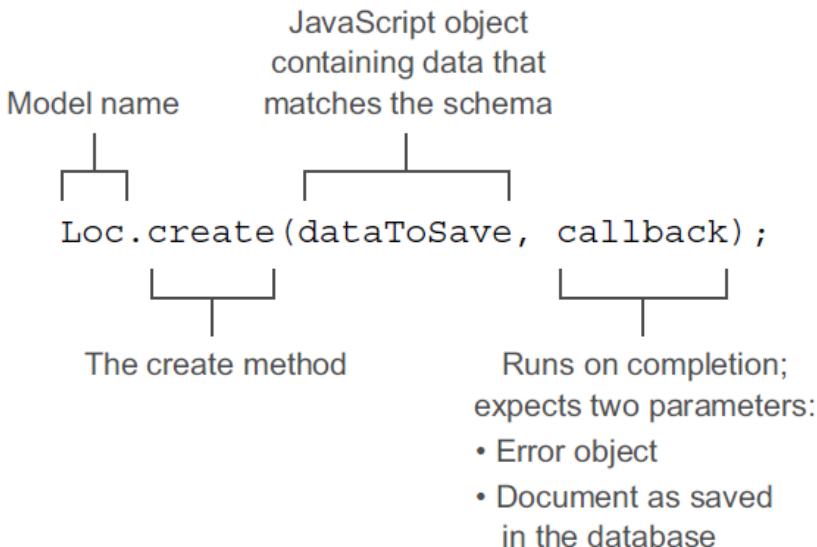
Action	Method	URL path	Parameters	Example
Create new location	POST	/locations		http://api.loc8r.com/locations
Create new review	POST	/locations/locationid/reviews	locationid	http://api.loc8r.com/locations/123/reviews

POST methods work by taking form data posted to them and adding it to the database. In the same way that URL parameters are accessed using `req.params` and query strings are accessed via `req.query`, Express controllers access posted form data via `req.body`.

Let's make a start by looking at how to create documents.

### 6.4.1 Creating new documents in MongoDB

In the database for Loc8r each location is a document, so this is what we'll be creating in this section. Mongoose really couldn't make the process of creating MongoDB documents much easier for you. You take your model, apply the `create` method, and send it some data and a callback function. This is the minimal construct, as it would be attached to our `Loc` model:



So that's pretty simple. There are two main steps to the creation process:

1. Take the posted form data and use it 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.

Looking at step 1 we already know that we can get data sent to us in a form by using `req.body`, and step 2 should be pretty familiar by now. So let's jump straight into the code. The following listing shows the full `locationsCreate` controller for creating a new document.

#### **Listing 6.5 Complete controller for creating a new location**

```

const locationsCreate = function(req, res) {
  Loc.create({
    name: req.body.name,
    address: req.body.address,
    facilities: req.body.facilities.split(","),
    coords: [parseFloat(req.body.lng), parseFloat(req.body.lat)], ①
    openingTimes: [
      {
        days: req.body.days1,
        opening: req.body.opening1,
        closing: req.body.closing1,
        closed: req.body.closed1,
      },
      {
        days: req.body.days2,
        opening: req.body.opening2,
        closing: req.body.closing2,
        closed: req.body.closed2,
      }
    ]
}

```

```

}, (err, location) => {
  if (err) {
    res
      .status(400)
      .json(err);
  } else {
    res
      .status(201)
      .json(location);
  }
});
};

```

④

- ① Apply create method to model
- ② Create array of facilities by splitting a comma-separated list
- ③ Parse coordinates from strings to numbers
- ④ Supply callback function, containing appropriate responses for success and failure

This shows how easy it can be to create a new document in MongoDB and save some data. For the sake of brevity we've limited the `openingTimes` array to two entries, but this could easily be extended, or better yet put into a loop checking for the existence of the values.

You might also notice that there's no `rating` being set. Remember in the schema that we set a default of `0`, as in the following snippet:

```

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

```

This is applied when the document is created, setting the initial value to be `0`. Something else about this code might be shouting out at you. There's no validation!

## 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? Again, we started this off in the Mongoose schemas. In the schemas we set a `required` flag to `true` in a few of the paths. When this flag is set, Mongoose will not send the data to MongoDB.

Given the following base schema for locations, for example, we can see that only `name` is a required field:

```

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: [Number],
  index: '2dsphere'
},
openingTimes: [openingTimeSchema],
reviews: [reviewSchema]
});
```

If this field is missing, the `create` method will raise an error and not attempt to save the document to the database.

Testing this API route in Postman looks like figure 6.11. Note that the method is set to `post`, and that the data type selected (above the list of names and values) is `x-www-form-urlencoded`.

key	value
name	Costy
address	High Street, Reading
facilities	hot drinks,food,power
lng	-0.9630884
lat	51.451041
days1	Monday - Friday
opening1	8:00am
closing1	5:00pm
closed1	false
days2	Saturday - Sunday
closed2	true

```

1  {
2    "__v": 0,
3    "name": "Costy",
4    "address": "High Street, Reading",
5    "coords": [
6      -0.9630884,
7      51.451041
8    ],
9    "_id": "590d8dc7a7cb5b8e3f1bfc48",
10   "reviews": [],
11   "openingTimes": [
12     {
13       "days": "Monday - Friday",
14       "opening": "8:00am",
15       "closing": "5:00pm",

```

Figure 6.11 Testing a POST method in Postman, ensuring that the method and form data settings are correct

#### 6.4.2 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>

### 3. Save the parent document.

Finding the correct parent isn't a problem as we've already done that, and can use it as the skeleton for the next controller, `reviewsCreate`. When we've found the parent, we can call an external function to do the next part (we'll write this function very soon), as shown in the following listing.

#### **Listing 6.6 Controller for creating a review**

```
const reviewsCreate = function(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); ①
        }
      });
  } else {
    res
      .status(404)
      .json({
        "message": "Not found, locationid required"
      });
  }
};
```

① Successful find operation will call new function to add review, passing request, response, and location objects

This isn't doing anything particularly new; we've seen it all before. By putting in a call to a new function we can keep the code neater by reducing the amount of nesting and indentation, and also make it easier to test.

#### **ADDING AND SAVING A SUBDOCUMENT**

Having found the parent document, and retrieved the existing list of subdocuments, we then need to add a new one. Subdocuments are essentially arrays of objects, and the easiest way to add a new object to an array is to create the data object and use the JavaScript `push` method. The following code snippet demonstrates this:

```
location.reviews.push({
  author: req.body.author,
  rating: req.body.rating,
  reviewText: req.body.reviewText
});
```

This is getting posted form data, hence using `req.body`.

Once the subdocument has been added, the parent document must be saved because subdocuments cannot be saved on their own. To save a document Mongoose has a model method `save`, which expects a callback with an error parameter and a returned object parameter. The following code snippet shows this in action:

```
location.save((err, location) => {
  if (err) {
    res
      .status(400)
      .json(err);
  } else {
    let thisReview = location.reviews[location.reviews.length - 1]; ①
    res
      .status(201)
      .json(thisReview);
  }
});
```

① Find last review in returned array, as MongoDB will return entire parent document, not just new subdocument

The document returned by the `save` method is the full parent document, not just the new subdocument. To return the correct data in the API response—that is, the subdocument—we need to retrieve the last subdocument from the array ①.

When adding documents and subdocuments you need to keep in mind any impact this may have on other data. In Loc8r adding a review will add a new rating. This new rating will impact the overall rating for the document. So on the successful save of a review we'll call another function to update the average rating.

Putting everything we have together in the `_doAddReview` function, plus a little extra error trapping, gives us the following listing.

### **Listing 6.7 Adding and saving a subdocument**

```
const _doAddReview = function(req, res, location) { ①
  if (!location) {
    res
      .status(404)
      .json({
        "message": "locationid not found"
      });
  } else {
    location.reviews.push({ ②
      author: req.body.author,
      rating: req.body.rating,
      reviewText: req.body.reviewText
    });
    location.save((err, location) => { ③
      if (err) {
        res
          .status(400)
          .json(err);
      }
    });
  }
};
```

```

    } else {
      _updateAverageRating(location._id);          4
      let thisReview = location.reviews[location.reviews.length - 1]; 5
      res
        .status(201)                                5
        .json(thisReview);
    }
  });
}
;

```

- 1 When provided with a parent document ...
- 2 push new data into subdocument array...
- 3 before saving it
- 4 On successful save operation call a function to update average rating
- 5 Retrieve last review added to array and return it as JSON confirmation response

### UPDATING THE AVERAGE RATING

Calculating the average rating isn't particularly complicated, so we won't dwell on it too long. The steps are

1. Find the correct document given a provided ID.
2. Add up the ratings from all of the review subdocuments.
3. Calculate the average rating value.
4. Update the rating value of the parent document.
5. Save the document.

Turning this list of steps into code gives us something along the lines of the following listing, which should be placed in the reviews.js controller file along with the review-based controllers.

#### **Listing 6.8 Calculating and updating the average rating**

```

const _updateAverageRating = function(locationid) {
  Loc
    .findById(locationid)                      1
    .select('rating reviews')                  1
    .exec((err, location) => {                1
      if (!err) {
        _doSetAverageRating(location);          1
      }
    });                                         1
};

const _doSetAverageRating = function(location) {
  if (location.reviews && location.reviews.length > 0) {
    const reviewCount = location.reviews.length; 2
    const ratingTotal = location.reviews.reduce((total, review) => { 2
      return total + review.rating;              2
    }, 0);                                     2
    let ratingAverage = parseInt(ratingTotal / reviewCount, 10); 3
    location.rating = ratingAverage;            4
    location.save((err) => {                   5
      if (err) {

```

```

        console.log(err);
    } else {
        console.log("Average rating updated to", ratingAverage);
    }
});
}
};

```

- ① Find correct document given supplied ID
- ② Use the JavaScript array reduce method to add up ratings from the subdocuments
- ③ Calculate average rating value
- ④ Update rating value of parent document
- ⑤ Save parent document

You might have noticed that we're not sending any JSON response here, and that's because we've already sent it. This entire operation is asynchronous and doesn't need to impact sending the API response confirming the saved review.

Adding a review isn't the only time we'll need to update the average rating. This is why it makes extra sense to make these functions accessible from the other controllers, and not tightly coupled to the actions of creating a review.

What we've just done here offers a sneak peak at using Mongoose to update data in MongoDB, so let's now move on to the PUT methods of the API.

## 6.5 PUT methods: Updating data in MongoDB

PUT methods are all about updating existing documents or subdocuments in the database, and then returning the saved data as confirmation. In the routes for Loc8r we have two PUT requests doing different things, listed in table 6.8.

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

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

PUT methods are similar to POST methods because they work by taking form data posted to them. But instead of using the data to create new documents in the database, PUT methods use the data to update existing documents.

### 6.5.1 Using Mongoose to update a document in MongoDB

In Loc8r we might 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, following these steps:

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 that 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 will update the original document in the database with your changes.

#### **USING THE MONGOOSE SAVE METHOD**

We've actually already seen this in action, when updating the average rating value. The `save` method is applied to the model instance that the `find` function returns. It expects a callback with the standard parameters of an error object and a returned data object.

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

```
Loc
  .findById(req.params.locationid)          ①
  .exec((err, location) => {
    location.name = req.body.name;
    location.save(function(err, location) { ③
      if (err) {
        res
          .status(404)          ④
          .json(err);          ④
      } else {
        res
          .status(200)          ④
          .json(location);     ④
      }
    });
  });
};
```

- ① Find document to update
- ② Make change to model instance, changing a value of one path
- ③ Save document with Mongoose's save method
- ④ Return success or failure response

Here we can clearly see the separate steps of finding, updating, saving, and responding. Fleshing out this skeleton into the `locationsUpdateOne` controller with some error trapping and the data we want to save gives us the following listing.

**Listing 6.9 Making changes to an existing document in MongoDB**

```

const locationsUpdateOne = function(req, res) {
  if (!req.params.locationid) {
    res
      .status(404)
      .json({
        "message": "Not found, locationid is required"
      });
    return;
  }
  Loc
    .findById(req.params.locationid)          ①
    .select('-reviews -rating')
    .exec((err, location) => {
      if (!location) {
        res
          .json(404)
          .status({
            "message": "locationid not found"
          });
        return;
      } else if (err) {
        res
          .status(400)
          .json(err);
        return;
      }
      location.name = req.body.name;
      location.address = req.body.address;
      location.facilities = req.body.facilities.split(',');
      location.coords = [
        parseFloat(req.body.lng),
        parseFloat(req.body.lat)
      ];
      location.openingTimes = [
        {
          days: req.body.days1,
          opening: req.body.opening1,
          closing: req.body.closing1,
          closed: req.body.closed1,
        },
        {
          days: req.body.days2,
          opening: req.body.opening2,
          closing: req.body.closing2,
          closed: req.body.closed2,
        }
      ];
      location.save((err, location) => {          ③
        if (err) {
          res
            .status(404)
            .json(err);
        } else {
          res
            .status(200)
            .json(location);                      ④
        }
      });
    });
}

```

```

    }
);
};

① Find location document by supplied ID
② Update paths with values from submitted form
③ Save instance
④ Send appropriate response, depending on outcome of save operation

```

There's clearly a lot more code here, now that it's fully fleshed out, but we can still quite easily identify the key steps of the update process.

The eagle-eyed among us may have noticed something strange in the select statement:

```
.select('-reviews -rating')
```

Previously we've used the `select` method to say which columns we *do want* to select. By adding a dash in front of a path name we're stating that we *don't want* to retrieve it from the database. So this `select` statement says to retrieve everything except the `reviews` and the `rating`.

### 6.5.2 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 then have to find the correct subdocument to make your changes. After this, the `save` method is applied to the document, not the subdocument. So the steps to updating an existing subdocument are

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

For Loc8r the subdocuments we're updating are reviews, so when a review is changed we'll have to remember to recalculate the average rating. That's the only additional thing we'll need to add in, above and beyond the five steps. The following listing shows this all put into place in the `reviewsUpdateOne` controller.

#### **Listing 6.10 Updating a subdocument in MongoDB**

```

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

```

```

.findById(req.params.locationid)      ①
.select('reviews')
.exec((err, location) => {
  if (!location) {
    res
      .status(404)
      .json({
        "message": "locationid not found"
      });
    return;
  } else if (err) {
    res
      .status(400)
      .json(err);
    return;
  }
  if (location.reviews && location.reviews.length > 0) {
    let thisReview = location.reviews.id(req.params.reviewid); ②
    if (!thisReview) {
      res
        .status(404)
        .json({
          "message": "reviewid not found"
        });
    } else {
      thisReview.author = req.body.author; ③
      thisReview.rating = req.body.rating; ③
      thisReview.reviewText = req.body.reviewText; ③
      location.save((err, location) => { ④
        if (err) {
          res
            .status(404)
            .json(err); ⑤
        } else {
          _updateAverageRating(location._id);
          res
            .status(200)
            .json(thisReview); ⑤
        }
      });
    }
  } else {
    res
      .status(404)
      .json({
        "message": "No review to update"
      });
  }
});
);

```

- ① Find parent document
- ② Find subdocument
- ③ Make changes to subdocument from supplied form data
- ④ Save parent document
- ⑤ Return a JSON response, sending subdocument object on basis of successful save

The five steps for updating are clear to see in this listing: find the document, find the subdocument, make changes, save, and respond. Once again a lot of the code here's error trapping, but it's vital for creating a stable and responsive API. You really don't want to save incorrect data, send the wrong responses, or delete data you don't want to. Speaking of deleting data, let's move on to the final of the four API methods we're using: DELETE.

## 6.6 DELETE method: Deleting data from MongoDB

The DELETE method is, unsurprisingly, all about deleting existing documents or subdocuments in the database. In the routes for Loc8r we have a DELETE request for deleting a location, and another for deleting a review. The details are listed in Table 6.9.

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

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

We'll start by taking a look at deleting documents.

### 6.6.1 Deleting documents in MongoDB

Mongoose makes deleting a document in MongoDB extremely simple by giving us the method `findByIdAndRemove`. This method expects just a single parameter—the ID of the document to be deleted.

The API should respond with a `404` in case of an error and a `204` in case of success. The following listing shows this all in place in the `locationsDeleteOne` controller.

#### Listing 6.11 Deleting a document from MongoDB given an ID

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

①

②

③

③

③

```

        .json(null);
    }
}
} else {
res
  .status(404)
  .json({
    "message": "No locationid"
  });
}

```

- ① Call `findByIdAndRemove` method, passing in `locationid`
- ② Execute method
- ③ Respond with failure or success

That's the quick and easy way to delete a document, but you can break it into a two-step process and find it then delete it if you prefer. This does give you the chance to do something with the document before deleting if you need to. This would look like the following code snippet:

```

Loc
.findById(locationid)
.exec((err, location) => {
  // Do something with the document
  Loc.remove((err, location) => {
    // Confirm success or failure
  });
});

```

So there's an extra level of nesting there, but with it comes an extra level of flexibility should you need it.

## 6.6.2 Deleting a subdocument from MongoDB

The process for deleting a subdocument is no different from the other work we'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.

Actually deleting the subdocument itself is really easy, as Mongoose gives us another helper method. You've already seen that we can find a subdocument by its ID with the `id` method like this:

```
location.reviews.id(reviewid)
```

Mongoose allows you to chain a `remove` method to the end of this statement like so:

```
location.reviews.id(reviewid).remove()
```

This will delete the subdocument from the array. Remember, of course, that the parent document will need saving after this to persist the change back to the database. Putting all the steps together—with a load of error trapping—into the `reviewsDeleteOne` controller looks like the following listing.

### **Listing 6.12 Finding and deleting a subdocument from MongoDB**

```
const reviewsDeleteOne = function(req, res) {
  if (!req.params.locationid || !req.params.reviewid) {
    res
      .status(404)
      .json({
        "message": "Not found, locationid and reviewid are both required"
      });
    return;
  }
  Loc
    .findById(req.params.locationid)          ①
    .select('reviews')
    .exec((err, location) => {
      if (!location) {
        res
          .status(404)
          .json({
            "message": "locationid not found"
          });
        return;
      } else if (err) {
        res
          .status(400)
          .json(err);
        return;
      }
      if (location.reviews && location.reviews.length > 0) {
        if (!location.reviews.id(req.params.reviewid)) {
          res
            .status(404)
            .json({
              "message": "reviewid not found"
            });
        } else {
          location.reviews.id(req.params.reviewid).remove(); ②
          location.save((err) => {                         ③
            if (err) {
              res
                .status(404)          ④
                .json(err);
            } else {
              _updateAverageRating(location._id);           ④
              res
                .status(204)          ④
            }
          });
        }
      }
    });
}
```

```

        .json(null);      ④
    });
}
} else {
res
  .status(404)
  .json({
    "message": "No review to delete"
  });
}
);
);

```

- ① Find relevant parent document
- ② Find and delete relevant subdocument in one step
- ③ Save parent document
- ④ Return appropriate success or failure response

Again, most of the code here's error trapping; there are seven possible responses the API could give and only one of them is the successful one. Actually deleting the subdocument is really easy; you just have to make absolutely sure that you're deleting the right one.

As we're deleting a review here, which will have a rating associated to it, we also have to remember to call the `_updateAverageRating` function to recalculate the average rating for the location. This should only be called if the delete operation is successful, of course.

And that is it. We've now built a REST API in Express and Node that can accept GET, POST, PUT, and DELETE HTTP requests to perform CRUD operations on a MongoDB database.

## 6.7 Summary

In this chapter we've covered

- The best practices for creating a REST API, including URLs, request methods, and response codes
- How the POST, GET, PUT, and DELETE HTTP request methods map onto common CRUD operations
- Mongoose helper methods for creating the helper methods
- Interacting with the data through Mongoose models, and how one instance of the model maps directly to one document in the database
- Managing subdocuments through their parent documents because you cannot access or save a subdocument in isolation
- Making the API robust by checking for any possible errors you can think of, so that a request is never left unanswered

Coming up next in chapter 7 we're going to see how to use this API from inside the Express application, finally making the Loc8r site database-driven!

# 7

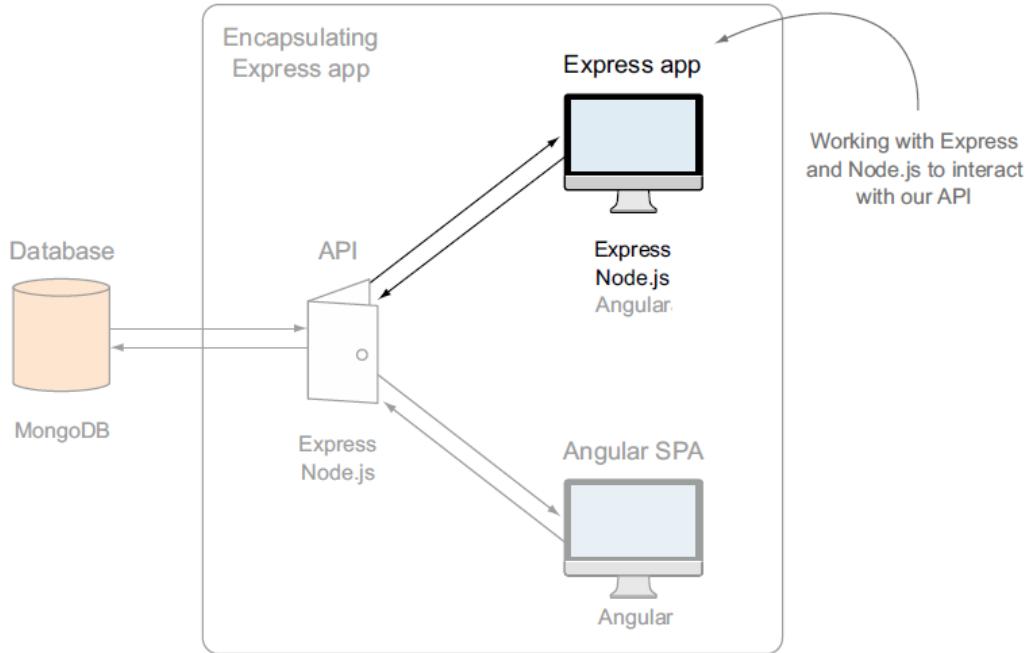
## *Consuming a REST API: Using an API from inside Express*

### This chapter covers

- Calling an API from an Express application
- Handling and using data returned by the API
- Working with API response codes
- Submitting data from the browser back to the API
- Validation and error traps

This chapter is an exciting one! Here's where we tie the front end to the back end for the first time. We'll remove the hard-coded data from the controllers, and end up showing data from the database in the browser instead. On top of this we'll push data back from the browser into the database via the API, creating new subdocuments.

The technology focus for this chapter is on Node and Express. Figure 7.1 shows where this chapter fits into the overall architecture and our grand plan.



**Figure 7.1** This chapter will focus on updating the Express application from chapter 4 to interact with the REST API developed in chapter 6.

In this chapter we'll discuss how to call an API from within Express, and how to deal with the responses. We'll make calls to the API to read from the database and write to the database. Along the way we'll look at handling errors, processing data, and creating reusable code by separating concerns. Toward the end we'll cover the various layers of the architecture to which we can add validation, and why these different layers are useful.

We'll start off by looking at how to call an API from the Express application.

## 7.1 How to call an API from Express

The first part we need to cover is how to call an API from Express. This isn't actually limited to our API; the approach can be used to call any API.

Our Express application needs to be able to call the API URLs that we set up in chapter 6—sending the correct request method, of course—and then be able to interpret the response. To help with doing this we'll use a module called `request`.

### 7.1.1 Adding the request module to our project

The `request` module is just like any of the other packages we've used so far, and can be added to our project using `npm`. To install the latest version and add it to the `package.json` file, head to terminal and type the following command:

```
$ npm install --save request
```

When `npm` has finished doing its thing, we can include `request` into the files that will use it. In `Loc8r` we only have one file that needs to make API calls, and that's the file with all of the controllers for the main server-side application. So right at the top of `locations.js` in `app_server/controllers` add the following line to `require` `request`:

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

Now we're good to go!

### 7.1.2 Setting up default options

Every API call with `request` must have a fully qualified URL, meaning that it must include the full address and not be a relative link. But this URL will be different for development and live environments.

To avoid having to make this check in every controller that makes an API call, we can set a default configuration option once at the top of the controllers file. To use the correct URL depending on the environment we can use our old friend the `NODE_ENV` environment variable.

Putting this into practice, the top of the controllers file should now look something like the following listing.

#### **Listing 7.1 Adding request and default API options to the locations.js controllers file**

```
const request = require('request');
const apiOptions = {
  server : 'http://localhost:3000'
};
if (process.env.NODE_ENV === 'production') {
  apiOptions.server = 'https://pure-temple-67771.herokuapp.com';
}
```

1

1

1

2

2

2

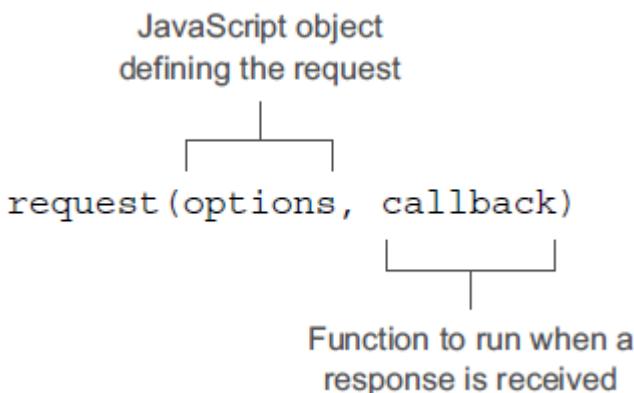
① Set default server URL for local development

② If application running in production mode set different base URL; change to be live address of application

With this in place every call we make to the API can reference `apiOptions.server` and will use the correct base URL.

### 7.1.3 Using the request module

The basic construct for making a request is really simple, being just a single command taking parameters for options and a callback like this:



The options specify everything for the request, including the URL, request method, request body, and query string parameters. These indeed are the options we'll be using in this chapter and they're detailed in table 7.1.

**Tabl 7.1 Four common request options for defining a call to an API**

Option	Description	Required
url	Full URL of the request to be made, including protocol, domain, path, and URL parameters	Yes
method	The method of the request, such as GET, POST, PUT, or DELETE	No—defaults to GET if not specified
json	The body of the request as a JavaScript object; an empty object should be sent if no body data is needed	Yes—ensures that the response body is also parsed as JSON
qs	A JavaScript object representing any query string parameters	No

The following code snippet shows an example of how you might put these together for a GET request. A GET request shouldn't have a body to send, but might have query string parameters.

```

const requestOptions = {
  url : 'http://yourapi.com/api/path',      1
  method : 'GET',                          2
  json : {},                             3
  qs : {                                  4
    offset : 20                           4
  }
};
  
```

- ① Define URL of API call to be made
- ② Set request method

- ③ Define body of request, even if it's an empty JSON object
- ④ Optionally add any query string parameters that might be used by API

There are many more options that you could specify, but these are the common four, and the ones we'll be using in this chapter. For more information on other possible options, take a look at the reference in the GitHub repository: <https://github.com/mikeal/request>.

The callback function runs when a response comes back from the API, and has three parameters: an error object, the full response, and the parsed body of the response. The error object will be `null` unless an error has been caught. Three pieces of data are going to be most useful in our code: the status code of the response, the body of the response, and any error thrown. The following code snippet shows an example of how you might structure a callback for the `request` function:

```
(err, response, body) => {
  if (err) { ①
    console.log(err);
  } else if (response.statusCode === 200) { ②
    console.log(body);
  } else { ③
    console.log(response.statusCode);
  }
}
```

- ① If error has been passed through, do something with it
- ② If response status code is 200 (request was successful), output JSON body of response
- ③ If request returned a different status code, output the code

The full response object contains a huge amount of information, so we won't go into it here. You can always check it out yourself in a `console.log` statement when we start adding the API calls into our application.

Putting the parts together, the skeleton for making API calls looks like the following:

```
const requestOptions = { ①
  url : 'http://yourapi.com/api/path', ①
  method : 'GET', ①
  json : {}, ①
  qs : { ①
    offset : 20 ①
  } ①
}; ①
request(requestOptions, (err, response, body) => { ②
  if (err) { ②
    console.log(err); ②
  } else if (response.statusCode === 200) { ②
    console.log(body); ②
  } else { ②
    console.log(response.statusCode); ②
  }
}); ②
```

- ① Define options for request

**② Make request, sending through options, and supplying a callback function to use responses as needed**

Let's move on and put this theory into practice, and start building the Loc8r controllers to use the API we've already built.

## 7.2 Using lists of data from an API: The Loc8r homepage

By now the controllers file that will be doing the work should already have the `request` module required in, and some default values set. So now comes the fun part—let's update the controllers to call the API and pull the data for the pages from the database.

We've got two main pages that pull data: the homepage showing a list of locations, and a Details page giving more information about a specific location. Let's start at the beginning and get the data for the homepage from the database.

The current homepage controller contains just a `res.render` statement sending hardcoded data to the view. But the way we want it to work is to render the homepage after the API has returned some data. The homepage controller is going to have quite a lot to do anyway, so let's move this rendering into its own function.

### 7.2.1 Separating concerns: Moving the rendering into a named function

There are a couple of reasons for moving the rendering into its own named function. First, we decouple the rendering from the application logic. The process of rendering doesn't care where or how it got the data; if it's given data in the right format it will use it. Using a separate function helps us get closer to the testable ideal that each function should do just one thing. An additional bonus related to this is that it becomes reusable, so we can call it from multiple places.

The second reason for creating a new function for the homepage rendering is that the rendering process occurs inside the callback of the API request. As well as making the code hard to test, it also makes it hard to read. The level of nesting required makes for a rather large, heavily indented controller function. As a point of best practice you should try to avoid these, as they're hard to read and understand when you come back to them.

The first step is to make a new function called `_renderHomepage` in the `locations.js` file in the `app_server/controllers` folder, and move the contents of the `homelist` controller into it. Remember to ensure it accepts the `req` and `res` parameters too. Listing 7.2 shows a very snipped down version of what we're doing here. You can now call this from the `homelist` controller, as also shown in the listing, and things will still work as before.

#### **Listing 7.2 Moving the contents of the homelist controller into an external function**

```
const _renderHomepage = function(req, res){
  res.render('locations-list', {
    title: 'Loc8r - find a place to work with wifi',
    ...
  });
};
```

```
const homelist = function(req, res){          ②
  _renderHomepage(req, res);                ②
};                                         ②
```

- ① Include all code from res.render call here (snipped down for brevity)  
 ② Call new \_renderHomepage function from homelist controller

This is a start, but we're not there yet—we want data!

### 7.2.2 Building the API request

We'll get the data we want by asking the API for it, and to do this we need to build the request. To build the request we need to know the URL, method, JSON body, and query string to send. Looking back at chapter 6, or indeed the API code itself, we can see that we need to supply the information shown in table 7.2.

**Table 7.2 Information needed to make a request to the API for a list of locations**

Parameter	Value
URL	SERVER:PORT/api/locations
Method	GET
JSON body	null
Query string	lng, lat, maxDistance

Mapping this information into a request is quite straightforward. As we saw earlier in the chapter the options for a request are just a JavaScript object. For the time being we'll hard-code values for longitude and latitude into the options, as it's quicker and easier for testing. Later in the book we'll make the application location-aware. For now we'll choose coordinates close to where the test data is stored. The maximum distance is set to be 20 kilometers.

When we make the request we'll pass through a simple callback function to call the `renderHomepage` function so that we don't leave the browser hanging.

Putting this into code, into the `homelist` controller, looks like the following listing.

#### **Listing 7.3 Update the homelist controller to call the API before rendering the page**

```
module.exports.homelist = function(req, res){
  const path = '/api/locations';           ①
  const requestOptions = {                ②
    url : apiOptions.server + path,        ②
    method : 'GET',                      ②
    json : {},                          ②
    qs : {                                ②
      lng : -0.7992599,                  ②
      lat : 51.378091,                  ②
      maxDistance : 20                  ②
    }                                    ②
};
```

```

};

request(
  requestOptions,
  (err, response, body) => {
    _renderHomepage(req, res);
  }
);

```

- 1 Set path for API request (server is already set at top of file)
- 2 Set request options, including URL, method, empty JSON body, and hard-coded query string parameters
- 3 Make request, sending through request options
- 4 Supplying callback to render homepage

If you save this and run the application again, the homepage should display exactly as before. We might now be making a request to the API, but we're ignoring the response.

### 7.2.3 Using the API response data

Seeing as we're going to the effort of calling the API, the least we can do is use the data it's sending back. We can make this more robust later, but we'll start with making it work. In making it work we're going to assume that a response body is returned to the callback, and we can just pass this straight into the `_renderHomepage` function, as highlighted in the following listing.

#### **Listing 7.4 Update the contents of the homelist controller to use the API response**

```

request(
  requestOptions,
  function(err, response, body) {
    _renderHomepage(req, res, body);
  }
);

```

- 1 Pass body returned by request to `_renderHomepage` function

Seeing as we coded the API, we know that the response body returned by the API should be an array of locations. The `_renderHomepage` function needs an array of locations to send to the view, so let's try just passing it straight through, making the changes highlighted in bold in the following listing.

#### **Listing 7.5 Update the `_renderHomepage` function to use the data from the API**

```

const _renderHomepage = function(req, res, responseBody) {           1
  res.render('locations-list', {
    title: 'Loc8r - find a place to work with wifi',
    pageHeader: {
      title: 'Loc8r',
      strapline: 'Find places to work with wifi near you!'
    },
    sidebar: "Looking for wifi and a seat? Loc8r helps you find places to work when"
}

```

```

    out and about. Perhaps with coffee, cake or a pint? Let Loc8r help you find
    the place you're looking for.",  

    locations: responseBody  

});  

};
```

- ① Add additional `responseBody` parameter to function declaration
- ② Remove hard-coded array of locations and pass `responseBody` through instead

Can it really be that easy? Try it out in the browser and see what happens. Hopefully you'll get something like figure 7.2.

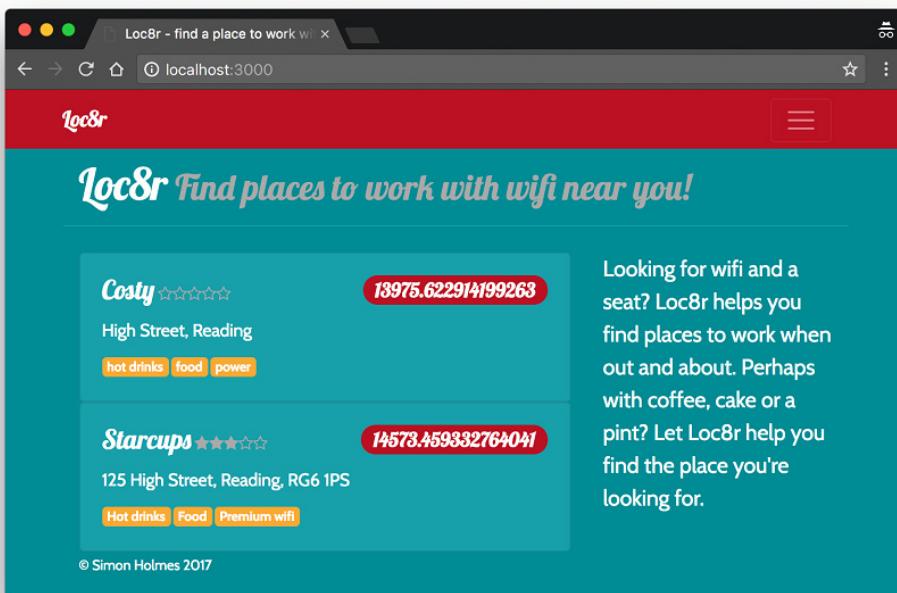


Figure 7.2 The first look at using data from the database in the browser—it's pretty close!

That looks pretty good, right? We need to do something about how the distance is displayed, but other than that all of the data is coming through as we wanted. Plugging in the data was quick and easy because of the work we did upfront designing the views, building controllers based on the views, and developing the model based on the controllers.

We've made it work. Now we need to make it better. There's no error trapping yet, and the distances need some work.

## 7.2.4 Modifying data before displaying it: Fixing the distances

At the moment the distances in the list are displaying 15 decimal places and no unit of measurement, so they're extremely accurate and totally useless! We want to say whether each distance is in meters or kilometers, and round the numbers off a single meter or to one decimal place of a kilometer. This should be done before sending the data to the `_renderHomepage` function, as that function should just be reserved for handling the actual rendering, not sorting out the data.

To do this we need to loop through the array of returned locations, formatting the distance value of each one. Rather than doing this inline we'll create an external function (in the same file) called `_formatDistance` that accepts a distance value and returns it nicely formatted.

Putting this all together looks like the following listing. Note that the framework of the `homelist` controller has been left out in this code snippet to keep things short, and the `request` statement still sits inside the controller.

### **Listing 7.6 Adding and using a function to format the distance returned by the API**

```
request(
  requestOptions,
  (err, response, body) => {
    let data = body;
    for (let i = 0; i < data.length; i++) {
      data[i].distance = _formatDistance(data[i].distance);
    }
    _renderHomepage(req, res, data);
  }
);

const _formatDistance = function (distance) {
  let thisDistance = 0;
  let unit = 'm';
  if (distance > 1000) {
    thisDistance = parseFloat(distance / 1000).toFixed(1);
    unit = 'km';
  } else {
    thisDistance = Math.floor(distance);
  }
  return thisDistance + unit;
};
```

- ① Assign returned body data to a new variable
- ② Loop through array, formatting distance value of location
- ③ Send modified data to be rendered instead of original body
- ④ If supplied distance is over 1000 m, convert to km, round to one decimal place and add km unit
- ⑤ Otherwise round down to nearest meter

If you make these changes and refresh the page you should see that the distances are now tidied up a bit and are actually useful, as shown in figure 7.3.

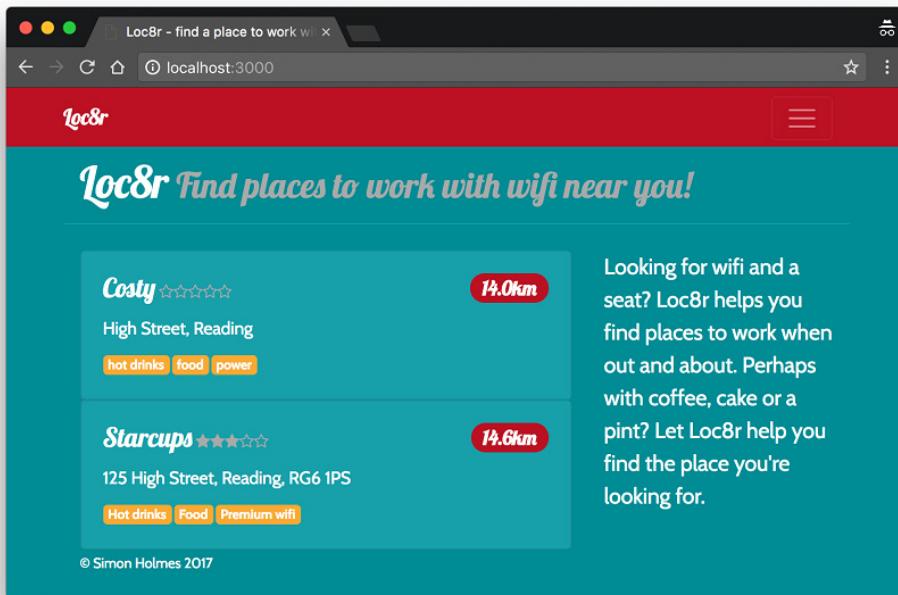


Figure 7.3 The homepage is looking better again after formatting the distances returned by the API.

That's better; the homepage is now looking more like we want it. For extra credit you can add some error trapping to the `_formatDistance` function to make sure that a `distance` parameter has been passed, and that it's a number.

### 7.2.5 Catching errors returned by the API

So far we've assumed that the API is always going to return an array of data along with a `200` success code. But this isn't necessarily the case. We coded the API to return a `200` status even if no locations are found nearby. As things stand, when this happens the homepage will display without any content in the central area. A far better user experience will be to output a message to the user that there are no places nearby.

We also know that our API can give `404` errors, so we'll need to make sure we handle these appropriately. We don't really want to show a `404` to the user in this case, because the error will not be due to the homepage itself being missing. The better option again here is to send a message to the browser in the context of the homepage.

Handling these scenarios shouldn't be too difficult; let's see how to do it, starting with the controller.

## MAKING THE REQUEST CALLBACK MORE ROBUST

One of the main reasons for catching errors is to make sure that they don't cause code to fail. The first point of weakness is going to be in the `request` callback where we're manipulating the response before sending the data off to be rendered. This is fine if the data is always going to be consistent, but we don't have that luxury.

The `request` callback currently runs a `for` loop to format the distances no matter what data is returned by the API. We should really only run this when the API returns a `200` code and some results.

The following listing shows how we can easily achieve this by adding in a simple `if` statement, checking the status code and the length of the returned data.

### **Listing 7.7 Validate that the API has returned some data before trying to use it**

```
request(
  requestOptions,
  (err, response, body) => {
    let data = body;
    if (response.statusCode === 200 && data.length) { ①
      for (let i = 0; i < data.length; i++) {
        data[i].distance = _formatDistance(data[i].distance);
      }
    }
    _renderHomepage(req, res, data);
  );
};
```

① Only run loop to format distances if API returned a 200 status and some data

Updating this piece of code should prevent this callback from falling over and throwing an error if the API responds with a status code other than `200`. The next link in the chain is the `_renderHomepage` function.

## DEFINING OUTPUT MESSAGES BASED ON THE RESPONSE DATA

Just like the `request` callback, our original focus for the `_renderHomepage` function is to make it work when passed an array of locations to display. Now that this might be sent different data types we need to make it handle the possibilities appropriately.

The response body could be one of three things:

- An array of locations
- An empty array, when no locations are found
- A string containing a message when the API returns an error

We already have the code in place to deal with an array of locations, so we need to address the other two possibilities. When catching these errors we also want to set a message that can be sent to the view.

To do this we need to update the `_renderHomepage` function to also do the following:

- Set a variable container for a message.
- Check to see whether the response body is an array; if not, set an appropriate message.
- If the response is an array, set a different message if it's empty (that is, no locations are returned).
- Send the message to the view.

The following listing shows how this looks in code.

#### **Listing 7.8 Outputting messages if the API doesn't return location data**

```
const _renderHomepage = function(req, res, responseBody){
  let message = null;
  if (!(responseBody instanceof Array)) {          1
    message = "API lookup error";                  2
    responseBody = [];
  } else {
    if (!responseBody.length) {                     3
      message = "No places found nearby";         3
    }
  }
  res.render('locations-list', {
    title: 'Loc8r - find a place to work with wifi',
    pageHeader: {
      title: 'Loc8r',
      strapline: 'Find places to work with wifi near you!'
    },
    sidebar: "Looking for wifi and a seat? Loc8r helps you find places to work when
      out and about. Perhaps with coffee, cake or a pint? Let Loc8r help you find
      the place you're looking for.",
    locations: responseBody,
    message: message                            4
  });
};
```

- ① Define a variable to hold a message
- ② If response isn't array, set message, and set responseBody to be empty array
- ③ If response is array with no length, set message
- ④ Add message to variables to send to view

The only surprise in there is when we set the `responseBody` to be an empty array if it was originally passed through as a string. We've done this to prevent the view from throwing an error. The view expects an array to be sent in the `locations` variable; it effectively ignores it if an empty array is sent, but will throw an error if a string is sent.

The last link in this chain is to update the view to display a message when one is sent.

#### **UPDATING THE VIEW TO DISPLAY THE ERROR MESSAGES**

So we're catching the errors from the API, and we're now also working with them to pass something back to the user. The final step is to let the user see the message by adding a placeholder into the view template.

We don't need to do anything fancy here—a simple `div` with a class of `error` to contain any messages will suffice. The following listing shows the `block content` section of the homepage view `locations-list.pug` in `app_server/views`.

#### **Listing 7.9 Update the view to display an error message when needed**

```
block content
  .row.banner
    .col-12
      h1= pageHeader.title
      small &nbsp;#{pageHeader.strapline}
  .row
    .col-12.col-md-8
      .error= message ①

      each location in locations
        .card
          .card-block
            h4
              a(href="/location")= location.name
              small &nbsp;
              +outputRating(location.rating)
              span.badge.badge-pill.badge-default.float-right= location.distance
            p.address= location.address
            .facilities
              each facility in location.facilities
              span.badge.badge-warning= facility

    .col-12.col-md-4
      p.lead= sidebar
```

- ① Add a `div` into main content area and have it display a message if one is sent

That's pretty easy—basic, but easy. It will certainly do for now. All that's left is to test it.

#### **TESTING THE API ERROR TRAPPING**

As with any new code, we now need to make sure that it works. A really easy way to test this is by changing the query string values that we're sending in the `requestOptions`.

To test the “no places found nearby” trap we can either set the `maxDistance` to a very small number (remembering that it's specified in kilometers), or set the `lng` and `lat` to a point where there are no locations. For example

```
requestOptions = {
  url : apiOptions.server + path,
  method : 'GET',
  json : {},
  qs : {
    lng : 1, ①
    lat : 1, ②
    maxDistance : 0.002 ③
  }
};
```

- ① Change query string values sent in request to get no results returned

### Fixing an interesting bug

Did you try testing the API error trapping by setting `lng` or `lat` to 0? You should have been expecting to see the “No places found nearby” message, but instead saw “API lookup error.” This is due to a bug in the error trapping in our API code.

In the `locationsListByDistance` controller, check to see whether the `lng` and `lat` query string parameters have been omitted by using a generic “falsey” JavaScript test. Our code simply has this: `if (!lng || !lat)`.

In falsey tests like this, JavaScript looks for any of the values that it considers to be false, such as an empty string, `undefined`, `null`, and, importantly for us, `0`. This introduces an unexpected bug into our code. If someone happened to be on the equator or on the Prime Meridian (that’s the Greenwich Mean Time line) they’d receive an API error.

This can be fixed by verifying the falsey test to say, “If it’s false but not zero.” In code this looks like this: `if ((!lng && lng !== 0) || (!lat && lat !== 0))`.

Updating your controller in the API will remove this bug.

You can use a similar tactic to test the `404` error. The API expects all of the query string parameters to be sent, and will return a `404` if one of them is missing. So to quickly test the code you can just comment one of them out as shown in the following code snippet:

```
requestOptions = {
  url : apiOptions.server + path,
  method : 'GET',
  json : {},
  qs : {
    // lng : -0.7992599,           ①
    lat : 51.378091,
    maxDistance : 20
  }
};
```

- ① Comment out one query string parameter in request to help test what happens when API returns 404

Do these two things one at a time and refresh the homepage to see the different messages coming through. These are shown in figure 7.4.

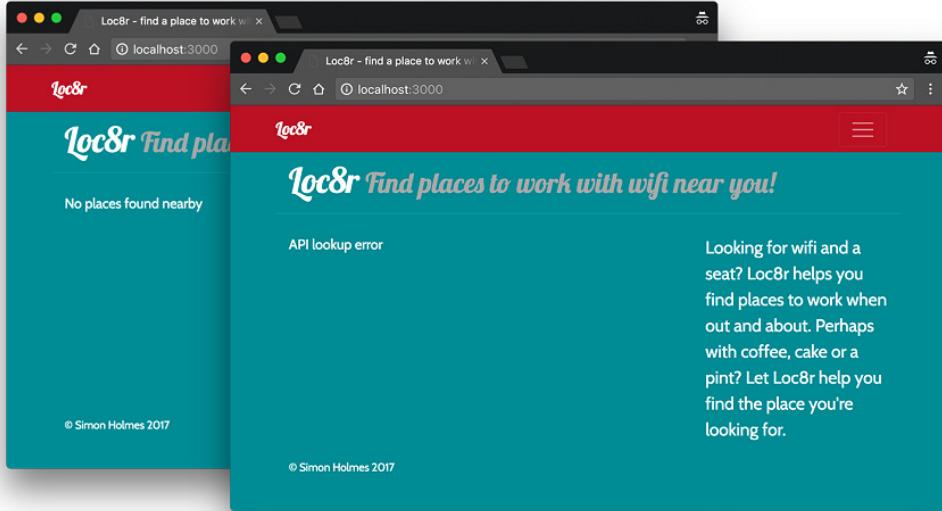


Figure 7.4 Showing the error message in the views after trapping the errors being returned by the API

That shows the homepage set up nicely. Our Express application is querying the API we built, which pulls data from the MongoDB database and passes it back to the application. When the application gets a response from the API, it works out what to do with it and either shows the data or an error message in the browser.

Now let's do the same thing for the Details page, this time working with single instances of data.

### 7.3 Getting single documents from an API: The Loc8r Details page

The Details page should display all of the information we have about a specific location, from the name and address, to ratings, reviews, facilities, and a location map. At the moment this is using data hard-coded into the controller, and looks like figure 7.5.

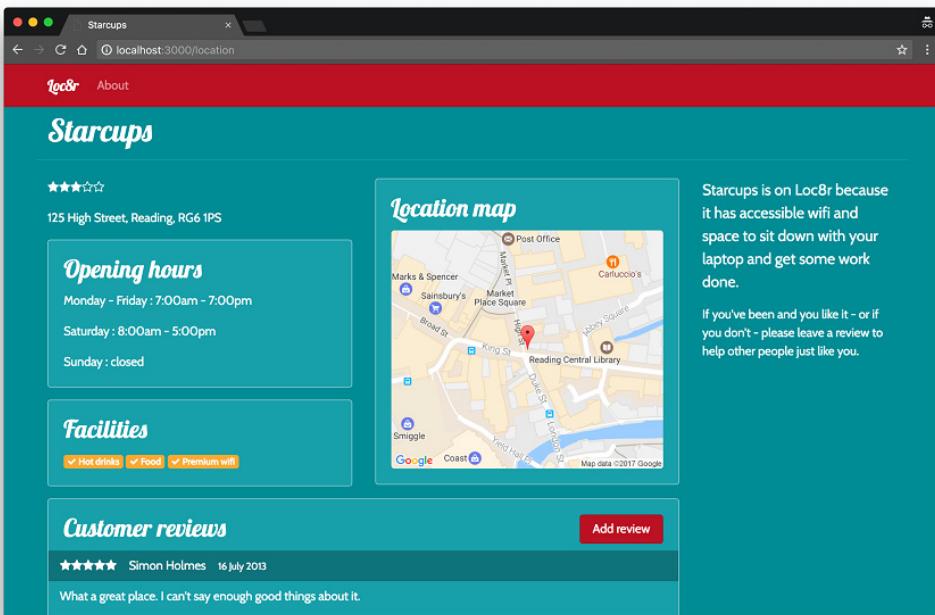


Figure 7.5 The Details page as it is now, using data hard-coded into the controller

In this section we'll update the application to allow us to specify which location we want the details for, get the details from the API, and output them to the browser. We'll also add in some error trapping, of course.

### 7.3.1 Setting URLs and routes to access specific MongoDB documents

The current path we have to the Details page is just `/location`. This doesn't offer a way to specify which location we want to look at. To address this we can borrow the approach from the API routes, where we specify the ID of the location document as a URL parameter.

The API route for a single location is `/api/locations/:locationid`. We can do the same thing for the main Express application and update the route to contain the `locationid` parameter. The main application routes for locations are in `locations.js` in the `/routes` folder. The following code snippet shows the simple change needed to update the location detail route to accept the `locationid` URL parameter:

```
router.get('/', ctrlLocations.homelist);
router.get('/location/:locationid', ctrlLocations.locationInfo); ①
router.get('/location/review/new', ctrlLocations.addReview);
```

① Add `locationid` parameter to route for single location

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

Licensed to Jacob Munkholm Hansen <au518331@uni.au.dk>

Okay, great ... but where do we get the IDs of the locations from? Thinking about the application as a whole, the homepage is the best place to start, as that's where the links for the Details page come from.

When the API for the homepage returns an array of locations, each location object contains its unique ID. This entire object is already passed to the view, so it shouldn't be too difficult to update the homepage view to add this ID as a URL parameter.

It's not difficult at all in fact! The following listing shows the little change that needs to be made in the locations-list.jade file to append the unique ID of each location to the link through to the Details page.

#### **Listing 7.10 Update the list view to add the location ID to the relevant links**

```
block content
  .row.banner
    .col-12
      h1= pageHeader.title
      small &nbsp;#{pageHeader.strapline}
  .row
    .col-12.col-md-8
      .error= message

      each location in locations
        .card
          .card-block
            h4
              a(href=`/location/${location._id}`)= location.name ①
              small &nbsp;
              +outputRating(location.rating)
              span.badge.badge-pill.badge-default.float-right= location.distance
              p.address= location.address
              .facilities
                each facility in location.facilities
                span.badge.badge-warning= facility
```

- ① As each location in array is looped through, pull unique ID from object and append it to href for link to Details page

If only everything in life was that easy. The homepage now contains unique links for each of the locations, and they all click through to the Details page. Now we just need to make them show the correct data.

#### **7.3.2 Separating concerns: Moving the rendering into a named function**

Just like we did for the homepage, we'll move the rendering of the Details page into its own named function. Again, this is to keep the rendering functionality separate from the API call and data processing.

The following listing shows a trimmed-down version of the new `_renderDetailPage` function, and how it's called from the `locationInfo` controller.

**Listing 7.11 Move the contents of the locationInfo controller into an external function**

```
const _renderDetailPage = function (req, res) { ①
  res.render('location-info', {
    title: 'Starcups',
    ...
  });
}

const locationInfo = function(req, res){
  renderDetailPage(req, res); ②
};
```

- ① Create new function called `_renderDetailPage` and move all contents of `locationInfo` controller into it
- ② Call new function from controller, remembering to pass it `req` and `res` parameters

Now we're set up with a nice, clear controller, ready to query the API.

**7.3.3 Querying the API using a unique ID from a URL parameter**

The URL for the API call needs to contain the ID of the location. Our Details page now has this ID as the URL parameter `locationid`, so we can get the value of this using `req.params` and add it to the `path` in the request options. The request is a GET request, and as such the `json` value will be an empty object.

Knowing all of this we can use the pattern we created in the homepage controller to build and make the request to the API. We'll call the `_renderDetailPage` function when the API responds. All of this is shown together in the following listing.

**Listing 7.12 Update the locationInfo controller to call the API**

```
const locationInfo = function(req, res){
  const path = `/api/locations/${req.params.locationid}`; ①
  requestOptions = {
    url : apiOptions.server + path, ②
    method : 'GET', ②
    json : {} ②
  };
  request(
    requestOptions,
    (err, response, body) => { ③
      renderDetailPage(req, res);
    }
  );
};
```

- ① Get `locationid` parameter from URL and append it to API path
- ② Set all request options needed to call API
- ③ Call `_renderDetailPage` function when API has responded

If you run this now you'll see the same static data as before, as we're not yet passing the data returned from the API into the view. You can add some console log statements into the `request` callback if you want to have a quick look at what's being returned.

If you're happy that all is working as it should, it's time for us to pass the data into the view.

### 7.3.4 Passing the data from the API to the view

We're currently assuming that the API is returning the correct data—we'll get around to error trapping soon. This data only needs a small amount of preprocessing: the coordinates are returned from the API as an array, but the view needs them to be named key-value pairs in an object.

The following listing shows how we can do this in the context of the `request` statement, transforming the data from the API before sending it to the `_renderDetailPage` function.

#### **Listing 7.13 Preprocessing data in the controller**

```
request(
  requestOptions,
  (err, response, body) => {
    let data = body;
    data.coords = {
      lng : body.coords[0],
      lat : body.coords[1]
    };
    renderDetailPage(req, res, data);
  }
);
```

- ① Create copy of returned data in new variable
- ② Reset coords property to be an object, setting lng and lat using values pulled from API response
- ③ Send transformed data to be rendered

The next logical step is to update the `_renderDetailPage` function to use this data rather than the hard-coded data. To make this work we need to make sure that the function accepts the data as a parameter, and then update the values passed through to the view as required. The following listing highlights the changes needed in bold.

#### **Listing 7.14 Update `renderDetailPage` to accept and use data from the API**

```
const renderDetailPage = function (req, res, locDetail) { ①
  res.render('location-info', {
    title: locDetail.name, ②
    pageHeader: {
      title: locDetail.name ②
    },
    sidebar: {
      context: 'is on Loc8r because it has accessible wifi and space to sit down with
              your laptop and get some work done.',
      callToAction: 'If you\'ve been and you like it - or if you don\'t - please
                    leave a review to help other people just like you.'
    },
    location: locDetail ③
  });
};
```

- 1 Add new parameter for data in function definition
- 2 Reference specific items of data as needed in function
- 3 Pass full locDetail data object to view, containing all details

We're able to take the approach of sending the full object through like this, because we originally based the data model on what was needed by the view and the controller. If you run the application now you should see that the page loads with the data pulled from the database. A screenshot of this is shown in figure 7.6.

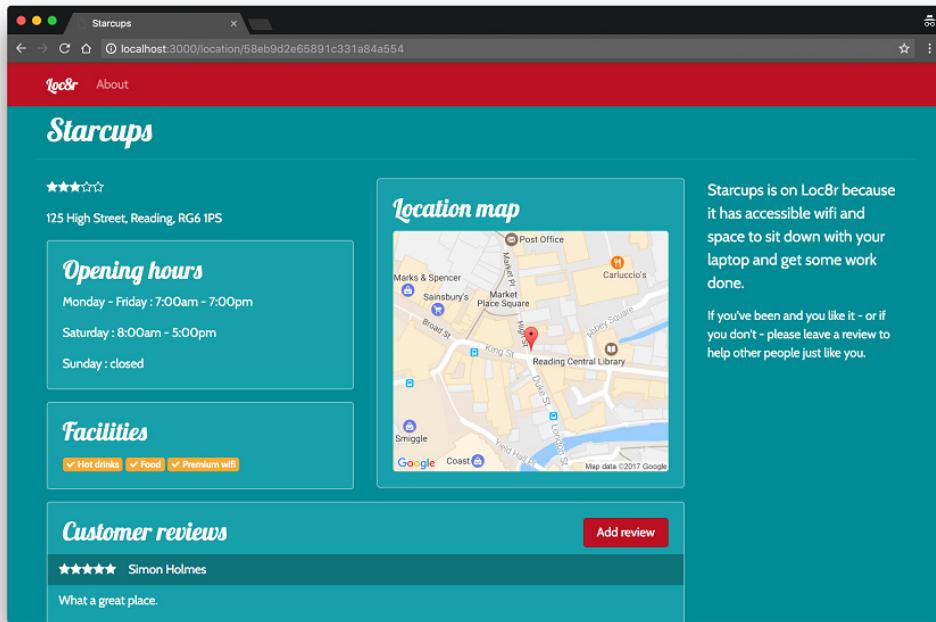


Figure 7.6 Details page pulling in data from MongoDB via the API

The eagle-eyed reader will have noticed a problem with the screenshot in figure 7.6. The review doesn't have a date associated with it.

### 7.3.5 Debugging and fixing the view errors

So, we have a problem with the view. It's not outputting the review date correctly. Perhaps we shouldn't have gotten overconfident about the fact that our data model was based on the view and controller? Let's take a look at what's going on.

Starting with a look at the Pug file location-info.pug in app\_server/views—we can isolate the line that outputs this section:

```
small.reviewTimestamp #{review.timestamp}
```

Now we need to check the schema to see if we changed something when defining the model. The schema for reviews is in locations.js in app\_api/models, and looks like the following code snippet:

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

Ah yes, here we can see that we changed the timestamp to be called `createdOn`, which is a more accurate name for the path.

Updating the Pug file using this value looks like the following:

```
small.reviewTimestamp #{review.createdOn}
```

Making these changes and refreshing the page gives us figure 7.7.



Figure 7.7 Pulling the name and date directly from the returned data; the format of the date isn't very user friendly

Success! Of sorts. The date is now showing, but not quite in the user-readable format that we'd like to see. We should be able to fix this using Pug.

### **FORMATTING DATES USING A PUG MIXIN**

Back when we were setting up the views we used a Pug mixin to output the rating stars based on the rating number provided. In Pug, mixins are like functions—you can send parameters

when you call them, run some JavaScript code if you wish, and have them generate some output.

Formatting dates is something that could be useful in a number of places, so let's create a mixin to do it. Our `outputRating` mixin is in the `sharedHTMLfunctions.pug` file in `app_server/views/_includes`. Let's add a new mixin called `formatDate` to that file.

In this mixin we'll largely use JavaScript to convert the date from the long ISO format into the more readable format of *Day Month Year*, for example *10 May 2017*. The ISO date object actually arrives here as a string, so the first thing to do is convert it into a JavaScript date object. When that's done we'll be able to use various JavaScript date methods to access the various parts of the date.

The following listing shows how this is done in a mixin—remember that lines of JavaScript in a Pug file must be prefixed with a dash.

#### **Listing 7.15 Create a Jade mixin to format the dates**

```
mixin formatDate(dateString)
  -const date = new Date(dateString);          ①
  -const monthNames = ['January', 'February', 'March', 'April', 'May', 'June',
    'July', 'August', 'September', 'October', 'November', 'December']; ②
  -const d = date.getDate();                  ③
  -const m = monthNames[date.getMonth()];      ③
  -const y = date.getFullYear();              ③
  -const output = `${d} ${m} ${y}`;           ④
  =output
```

- ① Convert date provided from string to date object
- ② Set up array of values for names of months
- ③ Use JavaScript data methods to extract and convert required parts of date
- ④ Put parts back together in desired format and render output

That mixin will now take a date and process it to output in the format that we want. As the mixin will render the output, we simply need to call it from the correct place in the code. The following code demonstrates this, again based on the same two isolated lines from the whole template:

```
span.reviewAuthor #{review.author.displayName}
small.reviewTimestamp
  +formatDate(review.createdOn)          ①
```

- ① Call mixin from its own line, passing creation date of review; make sure that new line is correctly indented

The call to the mixin should be placed on a new line, so you'll need to remember to take care with the indentation—the date should be nested inside the `<small>` tag.

Now the Details page is complete and looking like it should, as shown in figure 7.8.

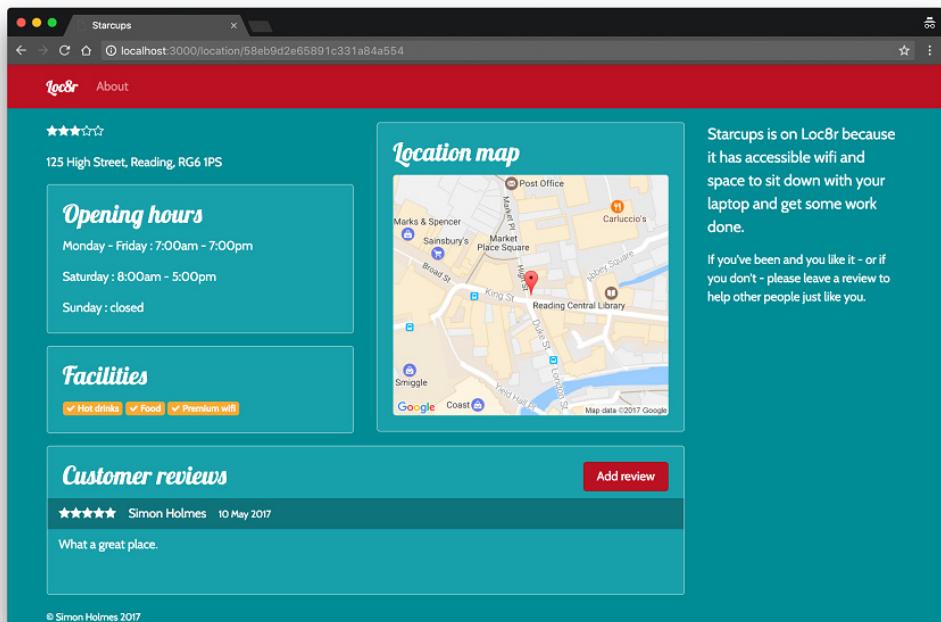


Figure 7.8 The complete Details page. The ID of the location is passed from the URL to the API, and the API retrieves the data and passes it back to the page to be formatted and rendered correctly.

Excellent; that's exactly what we wanted. If the URL contains an ID that's found in the database then the page displays nicely. But what happens if the ID is wrong, or isn't found in the database?

### 7.3.6 Creating status-specific error pages

If the ID from the URL isn't found in the database, the API will return a `404` error. This error originates from the URL in the browser, so the browser should also return a `404`—the data for the ID wasn't found, so in essence the page cannot be found.

Using techniques we've already seen in this chapter we can quite easily catch when the API returns a `404` status, using `response.statusCode` in the `request` callback. We don't really want to deal with it inside the callback, so we'll just pass the flow into a new function that we can call, `_showError`.

## CATCHING ALL ERROR CODES

Even better than just trapping for a `404` response, we can flip it over and look for any response from the API that isn't a `200` success response. We can pass the status code to the `_showError` function and let it figure out what to do. To enable the `_showError` function to keep control we'll also pass through the `req` and `res` objects.

The following listing shows how to update the `request` callback to render the Details page for successful API calls, and route all other errors to the catch-all function `_showError`.

### **Listing 7.16 Trap any errors caused by the API not returning a 200 status**

```
request(
  requestOptions,
  (err, response, body) => {
    let data = body;
    if (response.statusCode === 200) {           1
      data.coords = {
        lng : body.coords[0],                  2
        lat : body.coords[1]                  2
      };
      _renderDetailPage(req, res, data);       2
    } else {                                    3
      _showError(req, res, response.statusCode); 3
    }
  }
);
```

- 1 Check for successful response from API
- 2 Continue with rendering page if check successful
- 3 If check wasn't successful, pass error through to `_showError` function

Great, so now we'll only try to render the Details page if we have something from the API to display. So what shall we do with the errors? Well, for now we just want to send a message to the users letting them know that there's a problem.

## DISPLAYING ERROR MESSAGES

We don't want to do anything fancy here, just let the user know that something is going on and give them some indication of what it is. We have a generic Pug template already that's suitable for this; in fact, it's called `generic-text.pug` and expects just a title and some content. That will do us.

If you wanted to you could create a unique page and layout for each type of error, but for now we're good with just catching it and letting the user know. As well as letting the user know, we should also let the browser know by returning the appropriate status code when the page is displayed.

Listing 7.17 shows what the `_showError` function looks like, accepting a `status` parameter that, as well as being passed through as the response status code, is also used to define the

title and content of the page. Here we have a specific message for a 404 page and a generic message for any other errors that are passed.

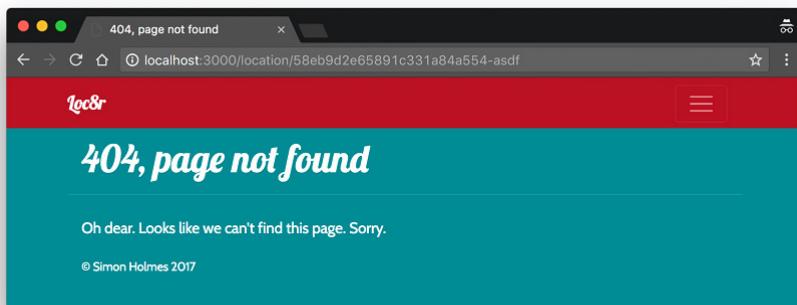
#### **Listing 7.17 Create an error-handling function for API status codes that aren't 200**

```
const _showError = function (req, res, status) {
  let title = '';
  let content = '';
  if (status === 404) {
    title = '404, page not found';
    content = 'Oh dear. Looks like we can\'t find this page. Sorry.';
  } else {
    title = `${status}, something's gone wrong`;
    content = 'Something, somewhere, has gone just a little bit wrong.';
  }
  res.status(status);
  res.render('generic-text', {
    title: title,
    content: content
  });
};
```

- ① If status passed through is 404, set title and content for page
- ② Otherwise set a generic catch-all message
- ③ Use status parameter to set response status
- ④ Send data to view to be compiled and sent to browser

This function can be reused from any of the controllers where we might find it useful. It's also built in such a way that we can easily add new, specific error messages for particular codes if we want to.

You can test the 404 error page by just slightly changing the location ID in the URL, and you should see something like figure 7.9.



**Figure 7.9** The 404 error page displayed when the location ID in the URL isn't found in the database by the API

That brings us to the end of the Details page. We can successfully display all of the information from the database for a given location, and also display a 404 message to the visitor if the location can't be found.

Following through the user journey, our next and final task is to add the ability to add reviews.

## 7.4 Adding data to the database via the API: Add Loc8r reviews

In this section we'll see how to take form data submitted by a user, process it, and post it to the API. Reviews are added to Loc8r by clicking the Add Review button on a location's Details page, filling in a form, and submitting it. At least that's the plan anyway. We currently have the screens to do this, but not the underlying functionality to make it happen. We're going to change that right now.

Here's a quick list of the things we're going to need to do:

1. Make the review form aware of which location the review will be for.
2. Create a route for the form to POST to.
3. Send the new review data to the API.
4. Show the new review in place on the Details page.

Note that at this stage in the development we don't have an authentication method in place, so we have no concept of user accounts.

### 7.4.1 Setting up the routing and views

The first item on our list is really about getting the ID of the location to the Add Review page in a way that we can use it when the form is submitted. After all, this is the unique identifier that the API will need to add a review.

The best approach for getting the ID to the page will be to contain it in the URL, like we did for the Details page itself.

#### **DEFINING THE TWO REVIEW ROUTES**

Getting the location ID into the URL will mean changing the route of the Add Review page to add a `locationid` parameter. While we're at it, we can deal with the second item on the list and create a route for the form to POST to. Ideally, this should have the same path as the review form, and be associated with a different request method and different controller. To do this, we'll update to use the `router.route` syntax, making it clear that we're using a single route with two different methods.

The following code snippet shows how we can update the routes in `index.js` in the `app_server/routes` folder:

```
router.get('/', ctrlLocations.homelist);
router.get('/location/:locationid', ctrlLocations.locationInfo);
router
  .route('/location/:locationid/review/new')
```

1

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

Licensed to Jacob Munkholm Hansen <au518331@uni.au.dk>

```
.get(ctrlLocations.addReview)
.post(ctrlLocations.doAddReview);
```

②

- ① Update to router.route syntax and insert locationid parameter into the review form route
- ② Create new route on same URL but using POST method and referencing different controller

Those are all of the routes we'll need for this section, but restarting the application will fail because the POST route references a controller that doesn't exist. We can fix this by adding a placeholder function into the controller file. Add the following code snippet into locations.js in app\_server/controllers and add doAddReview to the exports list at the bottom, then the application will fire up successfully once again:

```
const doAddReview = function(req, res){
};
```

Now the application will start again, but if you click through to the Add Review page you'll get an error. Oh yes, we need to update the link to the Add Review page from the Details page.

### **FIXING THE LOCATION DETAIL VIEW**

We need to add the location ID to the `href` specified in the Add Review button on the Details page. The controller for this page passes through the full data object as returned from the API, which, along with the rest of the data, will contain the `_id` field. This data object is called `location` when passed to the view.

The following code snippet shows a single line from the location-info.pug template in the app\_server/views folder. This shows how to add the location ID to the link for the Add Review button; note that we now use a JavaScript template string for the `href` value.

```
a.btn.btn-primary.float-right(href={`/location/${location._id}/review/new`}) Add review
```

Note that this is a single line of code; the linebreak is just the formatting of the book!

With that updated and saved, we can now click through to a review form for each individual location. There are just a couple of issues here: the form still doesn't post anywhere, and the name of the location is currently hard-coded into the controller.

### **UPDATING THE REVIEW FORM VIEW**

Next we want to make sure that the form posts to the correct URL. When the form is submitted now, it just makes a GET request to the /location URL as shown in the following code snippet:

```
form.form-horizontal(action="/location", method="get", role="form")
```

This line is taken from the location-review-form.pug file in app\_server/views. The /location path is no longer valid in our application, and we also want to use a POST request instead of a GET request. The URL we want to post the form to is actually the same as the URL for the Add Review: /location/:locationid/reviews/new.

A really easy way to achieve this is to set the action of the form to be an empty string, and set the method to be `post`, as shown in the following code snippet:

```
form.form-horizontal(action="", method="post", role="form")
```

Now when the form is submitted it will make a POST request to the URL of the current page.

### **CREATING A NAMED FUNCTION FOR RENDERING THE ADD REVIEW PAGE**

As with the other pages, we'll move the rendering of the page into a separate named function. This allows us the separation of concerns we're looking for when coding, and prepares us for the next steps.

The following listing shows how this should look in the code. Make your changes in `locations.js` in `app_server/controllers`.

#### **Listing 7.18 Create an external function to hold the contents of the addReview controller**

```
const _renderReviewForm = function (req, res) {           ①
  res.render('location-review-form', {
    title: 'Review Starcups on Loc8r',
    pageHeader: { title: 'Review Starcups' }
  });
};

/* GET 'Add review' page */
const addReview = function(req, res){
  _renderReviewForm(req, res);                           ②
};
```

- ① Create new function `_renderReviewForm` and move contents of `addReview` controller into it
- ② Call new function from within `addReview` controller, passing through same parameters

This might look a little odd, creating a named function and then having the call to that function be the only thing in the controller, but it will be very useful in just a moment.

### **GETTING THE LOCATION DETAIL**

On the Add Review page we want to display the name of the location in order to retain a sense of context for the user. This means we want to hit the API again, give it the ID of the location, and get the information back to the controller and into the view. We've just done this for the Details page, albeit with a different controller. If we approach this right we shouldn't have to write much new code.

Rather than duplicating the code and having to maintain two pieces, we'll go for a DRY (don't repeat yourself) approach. The Details page and the Add Review page both want to call the API to get the location information and then do something with it. So why not create a new function that does just this? We've already got most of the code in the `locationInfo` controller, we just need to change how it calls the final function. Instead of calling the `_renderDetailPage` explicitly, we'll make it a callback.

So we'll have a new function called `_getLocationInfo` that will make the API request. Following a successful request, this should then invoke whatever callback function was passed. The `locationInfo` controller will now call this function, passing a callback function that simply calls the `_renderDetailPage` function. Similarly, the `addReview` controller can also call this new function, passing it the `_renderReviewForm` function in the callback.

This gives us one function making the API calls that will have different outcomes depending on the callback function sent through. The following listing shows this all in place.

### **Listing 7.19 Create a new reusable function to get location information**

```
const _getLocationInfo = function (req, res, callback) { ①
  const path = `/api/locations/${req.params.locationid}`;
  const requestOptions = {
    url : apiOptions.server + path,
    method : 'GET',
    json : {}
  };
  request(
    requestOptions,
    (err, response, body) => {
      let data = body;
      if (response.statusCode === 200) {
        data.coords = {
          lng : body.coords[0],
          lat : body.coords[1]
        };
        callback(req, res, data); ②
      } else {
        _showError(req, res, response.statusCode);
      }
    }
  );
};

const locationInfo = function(req, res){
  _getLocationInfo(req, res, (req, res, responseData) => { ③
    _renderDetailPage(req, res, responseData);
  });
};

const addReview = function(req, res){
  _getLocationInfo(req, res, (req, res, responseData) => { ④
    _renderReviewForm(req, res, responseData);
  });
};
```

- ① New function `_getLocationInfo` accepts callback as third parameter and contains all code that used to be in `locationInfo` controller
- ② Following successful API response, invoke callback instead of named function
- ③ In `locationInfo` controller call `_getLocationInfo` function, passing a callback function that will call `_renderDetailPage` function upon completion
- ④ Also call `_getLocationInfo` from `addReview` controller, but this time pass `_renderReviewForm` in callback

And there we have a nice DRY approach to the problem. It would have been very easy to just copy and paste the API code from one controller to another, which, if we're being honest, is

absolutely fine if you're figuring out your code and what you need to make it work. But when you see two pieces of code doing pretty much exactly the same thing, always ask yourself how you can make it DRY—it makes your code cleaner and easier to maintain.

### DISPLAYING THE LOCATION DETAIL

We're forgetting one thing here. The function for rendering the form still contains hard-coded data instead of using the data from the API. A quick tweak to the function will change that, as is illustrated in the following listing.

#### **Listing 7.20 Removing hard-coded data from the `_renderReviewForm` function**

```
const _renderReviewForm = function (req, res, locDetail) {    ①
  res.render('location-review-form', {
    title: `Review ${locDetail.name} on Loc8r`,
    pageHeader: { title: `Review ${locDetail.name}` }          ②
  });
};
```

- ① Update `_renderReviewForm` function to accept new parameter containing data
- ② Swap out hard-coded data for data references using template strings

And with that the Add Review page is looking good once again, displaying the correct name based on the ID found in the URL, as shown in figure 7.10.

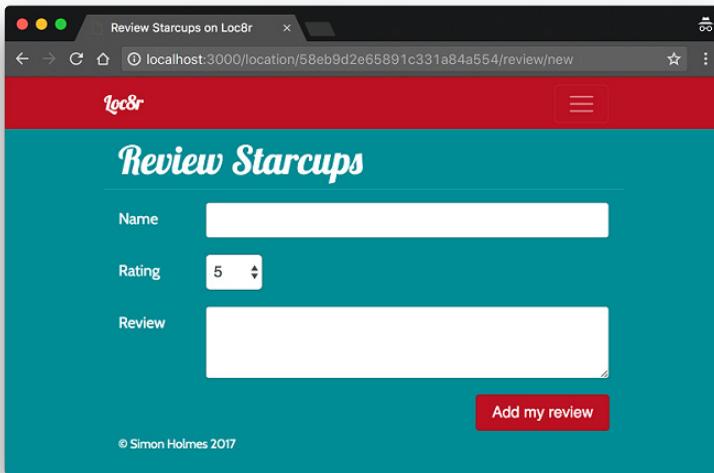


Figure 7.10 Add Review page pulling in the location name via the API, based on the ID contained in the URL

### 7.4.2 POSTing the review data to the API

By now we have the Add Review page set up and ready to go, including the posting destination. We've even got the route and controller for the POST action in place. The controller, `doAddReview`, is just an empty placeholder, though.

The plan for this controller is as follows:

1. Get the location ID from the URL to construct the API request URL.
2. Get the data posted in the form and package it up for the API.
3. Make the API call.
4. Show the new review in place if successful.
5. Display an error page if not successful.

The only part of this that we haven't seen yet is passing the data to the API; so far we've just passed an empty JSON object to ensure that the response is formatted as JSON. Now we're going to take the form data and pass it to the API in the format it expects. We have three fields on the form, and three references that the API expects. All we need to do is map one to the other. The form fields and model paths are shown in table 7.3.

**Table 7.3 Mapping the names of the form fields to the model paths expected by the API**

Form field	API references
name	author
rating	rating
review	reviewText

Turning this mapping into a JavaScript object is pretty straightforward. We just need to create a new object containing the variable names that the API expects, and use `req.body` to get the values from the posted form. The following code snippet shows this in isolation, and we'll put it into the controller in just a moment:

```
const postdata = {
  author: req.body.name,
  rating: parseInt(req.body.rating, 10),
  reviewText: req.body.review
};
```

Now that we've seen how that works, we can add it into the standard pattern we've been using for these API controllers and build out the `doAddReview` controller. Remember that the status code the API returns for a successful POST operation is `201`, not the `200` we've been using so far with the GET requests. The following listing shows the `doAddReview` controller using everything we've learned so far.

**Listing 7.21 doAddReview controller used to post review data to the API**

```
const doAddReview = function(req, res){
  const locationid = req.params.locationid; ①
  const path = `/api/locations/${locationid}/reviews`; ①
  const postdata = {
    author: req.body.name, ②
    rating: parseInt(req.body.rating, 10), ②
    reviewText: req.body.review ②
  };
  const requestOptions = {
    url : apiOptions.server + path, ③
    method : 'POST', ③
    json : postdata ③
  };
  request( ④
    requestOptions,
    (err, response, body) => {
      if (response.statusCode === 201) { ⑤
        res.redirect(`/location/${locationid}`); ⑤
      } else {
        _showError(req, res, response.statusCode); ⑤
      }
    }
  );
};
```

- ① Get location ID from URL to construct API URL
- ② Create data object to send to API using submitted form data
- ③ Set request options, including path, setting POST method and passing submitted form data into json parameter
- ④ Make the request
- ⑤ Redirect to Details page if review was added successfully or show an error page if API returned an error

Now we can create a review and submit it, and then see it on the Details page, as shown in figure 7.11.

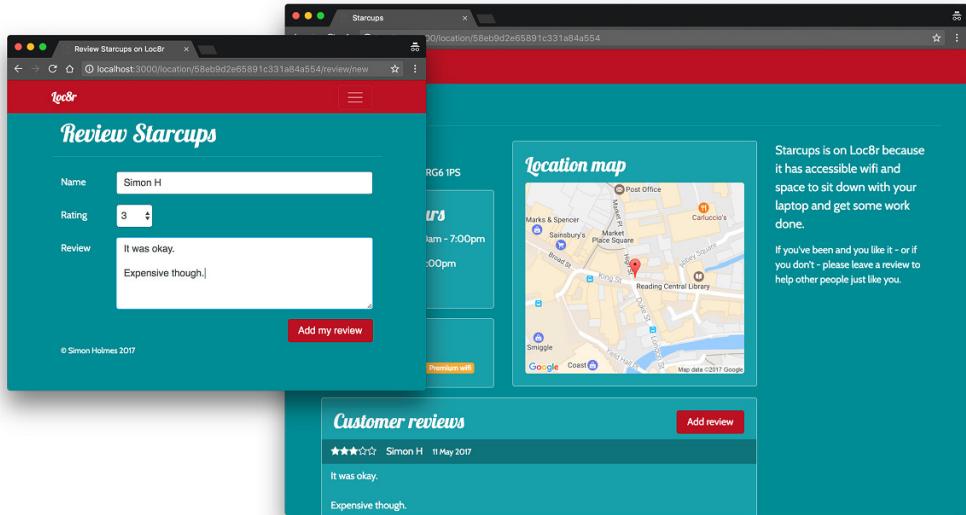


Figure 7.11 After filling in and submitting the review form, the review is shown in situ on the Details page.

Now that it all works, let's take a quick look at adding form validation.

## 7.5 Protecting data integrity with data validation

Whenever an application accepts external input and adds it to a database you need to make sure that the data is complete and accurate—as much as you can, or as much as it makes sense to. For example, if someone is adding an email address you should check that it's a valid email format, but you can't programmatically validate that it's a *real* email address.

In this section we're going to look at the ways we can add validation to our application, to prevent people from submitting empty reviews. There are three places that we can add validation:

- At the schema level, using Mongoose, before the data is saved
- At the application level, before the data is posted to the API
- At the client side, before the form is submitted

We'll look at each of these in turn, and add some validation at every step.

### 7.5.1 Validating at the schema level with Mongoose

Validating the data before saving it is arguably the most important stage. This is the final step, the one last chance to make sure that everything is as correct as it can be. This stage is particularly important when the data is exposed through an API; if we don't have control over

all of the applications using the API we can't guarantee the quality of the data that we're going to get. So it's important to ensure that the data is valid before saving it.

### UPDATING THE SCHEMA

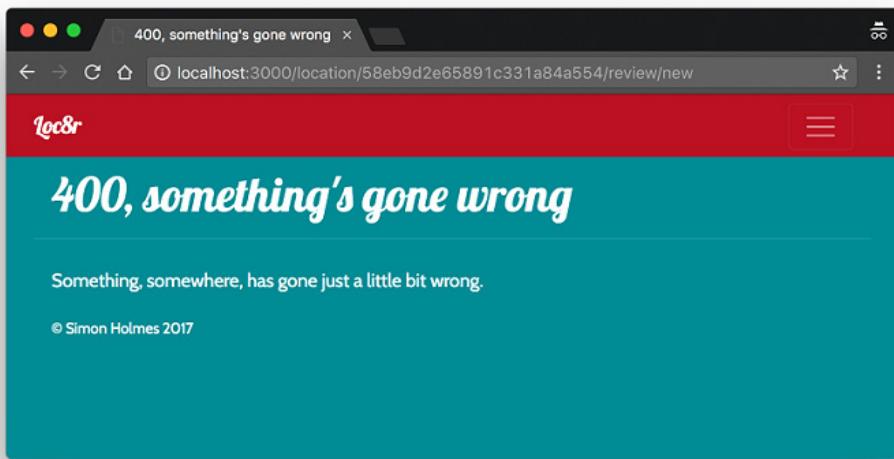
When we first set up the schema in chapter 5, we looked at adding some validation in Mongoose. We set the `rating` path to be required, but we also want the `author` `displayName` and `reviewText` to be required. If any of these fields are missing, a review won't make sense. Adding this to the schema is simple enough, and looks like the following listing (the schema is in `locations.js` in the `app_api/model` folder).

#### **Listing 7.22 Adding validation to reviews at the schema level**

```
const reviewSchema = new mongoose.Schema({
  author: {
    type: String,
    required: true
  },
  rating: {
    type: Number,
    required: true,
    min: 0,
    max: 5
  },
  reviewText: {
    type: String,
    required: true
  },
  createdOn: {
    type: Date,
    'default': Date.now
  }
});
var reviewSchema = new mongoose.Schema({
  author: {type: String, required: true},
  rating: {type: Number, required: true, min: 0, max: 5}, ①
  reviewText: {type: String, required: true}, ①
  createdOn: {type: Date, "default": Date.now} ②
});
```

- ① Make each of these paths a required field because if any of them are missing, a review won't make sense
- ② `createdOn` doesn't need to be required because Mongoose automatically populates it when a new review is created

Once this is saved we can no longer save a review without any review text. We can try, but we'll see the error page shown in figure 7.12.



**Figure 7.12** Error message shown when trying to save a review without any review text, now that the schema says it's required

On the one hand it's good that we're protecting the database, but it's not a great user experience. We should try to catch that error and let the visitor try again.

#### CATCHING MONGOOSE VALIDATION ERRORS

If you try to save a document with one or more required paths missing or empty, Mongoose will return an error. It does this without having to make a call to the database, as it's Mongoose itself that holds the schema and knows what is and isn't required. The following code snippet shows an example of such an error message:

```
{
  message: 'Validation failed',
  name: 'ValidationError',
  errors: {
    'reviews.1.reviewText': {
      message: 'Path `reviewText` is required.',
      name: 'ValidatorError',
      path: 'reviewText',
      type: 'required',
      value: ''
    }
  }
}
```

In the flow of the application this happens inside the callback from the `save` function. If we take a look at the `save` command inside the `_doAddReview` function (in `locations.js` in `app_api/controllers`) we can see where the error bubbles up and where we set the `400` status. The following code snippet shows this, including a temporary console log statement to show the output of the error to terminal:

```
location.save((err, location) => {
  if (err) {
    console.log(err); ①
    res
      .status(400)
      .json(err);
  } else {
    _updateAverageRating(location._id);
    let thisReview = location.reviews[location.reviews.length - 1];
    res
      .status(201)
      .json(thisReview);
  }
});
```

- ① Mongoose validation errors are returned through error object following attempted save action

Our API returns this message as the response body, alongside the `400` status. So we can look for this information in our application by looking at the response body when the API returns a `400` status.

The place to do this is in the `app_server`, in the `doAddReview` function in `controllers/locations.js`, to be precise. When we've caught a validation error we want to let the user try again by redirecting to the Add Review page. So that the page knows that an attempt has been made, we can pass a flag in the query string.

The following listing shows this code in place, inside the `request` statement callback for the `doAddReview` function.

#### **Listing 7.23 Trapping validation errors returned by the API**

```
request(
  requestOptions,
  (err, response, body) => {
    if (response.statusCode === 201) {
      res.redirect('/location/${locationid}`);
    } else if (response.statusCode === 400 && body.name && body.name ===
      'ValidationError') {
      res.redirect('/location/${locationid}/review/new?err=val'); ②
    } else {
      console.log(body);
      _showError(req, res, response.statusCode);
    }
  });
});
```

- ① Add in check to see if status is 400, if body has a name, and if that name is ValidationError

**② If true redirect to review form, passing an error flag in query string**

So now when the API returns a validation error we can catch it and send the user back to the form to try again. Passing a value in the query string means that we can look for this in the controller that displays the review form, and send a message to the view to alert the user to the problem.

#### **DISPLAY AN ERROR MESSAGE IN THE BROWSER**

To display an error message in the view, we need to send a variable to the view if we see the `err` parameter passed in the query string. The `_renderReviewForm` function is responsible for passing the variables into the view. When it's called it's also passed the `req` object, which contains the `query` object, making it quite easy to pass the `err` parameter when it exists. The following listing highlights the simple change required to make this happen.

**Listing 7.24 Update the controller to pass a query string error string to the view**

```
const renderReviewForm = function (req, res, locDetail) {
  res.render('location-review-form', {
    title: `Review ${locDetail.name} on Loc8r`,
    pageHeader: { title: `Review ${locDetail.name}` },
    error: req.query.err ①
  });
};
```

**① Send new error variable to view, passing it query parameter when it exists**

The `query` object is always part of the `req` object, regardless of whether it has any content. This is why we don't need to error trap this and check that it exists—if the `err` parameter isn't found it will just return `undefined`.

All that remains is to do something with this information in the view, letting the user know what the problem is. We'll show a message to the user at the top of the form, if a validation error was bubbled up. To give this some style and presence on the page we'll use a Bootstrap alert component; this is just a `div` with some relevant classes and attributes. The following code snippet shows the two lines needed added in place in the `location-review-form` view:

```
form(action="", method="post", role="form")
  - if (error == "val")
    .alert.alert-danger(role="alert") All fields required, please try again
```

So now when the API returns a validation error we catch this and display a message to the user. Figure 7.13 shows how this looks.

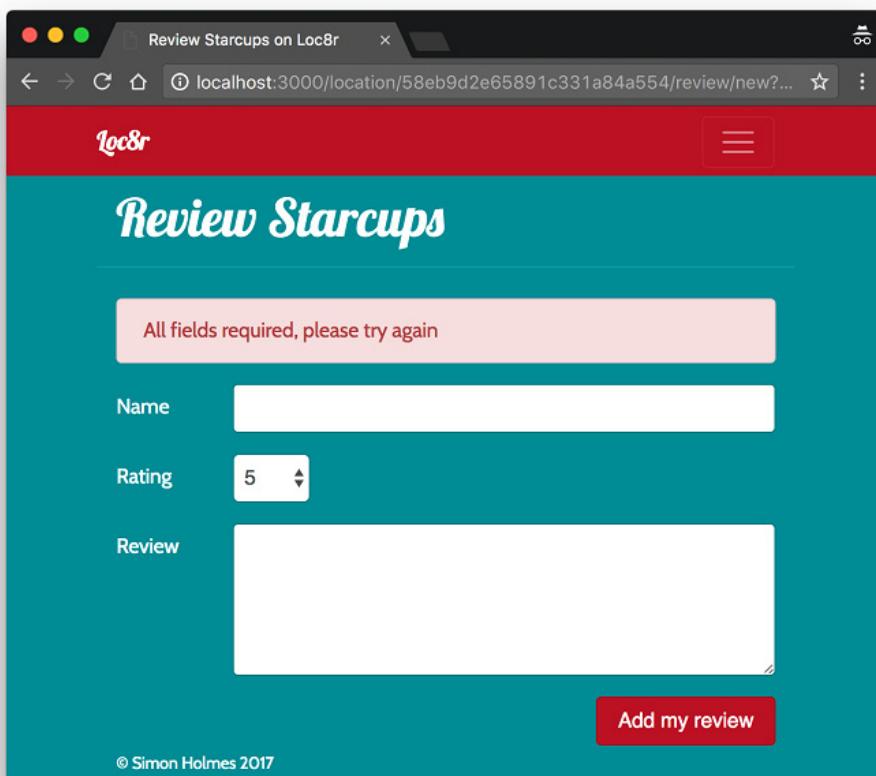


Figure 7.13 The validation error message showing in the browser, the end result of a process kicked off by Mongoose catching the error and returning it

This type of validation at the API level is important, and is generally a great place to start because it always protects a database against inconsistent or incomplete data, no matter the origin. But the experience for end users isn't always the best—they have to submit the form, and it makes a roundtrip to the API before the page reloads with an error. There's clearly room for improvement here, and the first step is to perform some validation at the application level before the data is passed to the API.

### 7.5.2 Validating at the application level with Node and Express

Validation at the schema level is the backstop, the final line of defense in front of a database. An application shouldn't rely solely on this, and you should try to prevent unnecessary calls to the API, reducing overhead and speeding things up for the user. One way to do this is to add validation at the application level, checking the submitted data before sending it to the API.

In our application, the validation required for a review is pretty simple; we can add some simple checks to ensure that each of the fields has a value. If this fails then we redirect the user back to the form, adding the same query string error flag as before. If the validation checks are successful then we allow the controller to continue into the request method. The listing on the next page shows the additions needed in the `doAddReview` controller in `locations.js` in the `app_server/controllers` folder.

#### **Listing 7.25 Adding some simple validation to an Express controller**

```
const doAddReview = function(req, res){
  const locationid = req.params.locationid;
  const path = `/api/locations/${locationid}/reviews`;
  const postdata = {
    author: req.body.name,
    rating: parseInt(req.body.rating, 10),
    reviewText: req.body.review
  };
  const requestOptions = {
    url : apiOptions.server + path,
    method : 'POST',
    json : postdata
  };
  if (!postdata.author || !postdata.rating || !postdata.reviewText) { ①
    res.redirect('/location/' + locationid + '/review/new?err=val'); ①
  } else { ②
    request(
      requestOptions,
      (err, response, body) => {
        if (response.statusCode === 201) {
          res.redirect(`/location/${locationid}`);
        } else if (response.statusCode === 400 && body.name && body.name ===
        'ValidationError') {
          res.redirect(`/location/${locationid}/review/new?err=val`);
        } else {
          _showError(req, res, response.statusCode);
        }
      }
    );
  }
};
```

- ① If any of three required data fields are falsey, then redirect to Add Review page, appending query string used to display error message
- ② Otherwise continue as before

The outcome for this will be the same as before—if the review text is missing then the user gets shown the error message on the Add Review page. The user doesn’t know that we’re no longer posting data to the API, but it’s one less roundtrip and so it should be a faster experience. But we can make it even faster with the third tier of validation: browser-based validation.

### 7.5.3 Validating in the browser with jQuery

Just like application-level validation speeds things up by not requiring a call to the API, client-side validation in the browser can speed things up by catching an error before the form is submitted to the application, by removing yet another call. Catching an error at this point will keep the user on the same page.

To get JavaScript running in the browser, we need to place it in the public folder in the application. Express treats the contents of this folder as static files to be downloaded to the browser instead of being run on the server. If you don’t have a folder called javascripts in your public folder already, create one now. Inside this new folder create a new file called validation.js.

#### WRITING THE JQUERY VALIDATION

Inside this new validation.js file we’ll put a jQuery function that will do the following:

- Listen for the submit event of the review form.
- Check to see that all of the required fields have a value.
- If one is empty, show an error message like we’ve used in the other types of validation, and prevent the form from submitting.

We won’t dive into the semantics of jQuery here, assuming you have some familiarity with it or a similar library. The following listing shows the code to do this.

#### Listing 7.26 Creating a jQuery form validation function

```
$('#addReview').submit(function (e) {  
    $('.alert.alert-danger').hide();  
    if (!($('input#name').val() || !($('select#rating').val() ||  
        !$('textarea#review').val())) {  
        if ($('.alert.alert-danger').length) {  
            $('.alert.alert-danger').show();  
        } else {  
            $(this).prepend('<div role="alert" class="alert alert-danger">All fields  
                required, please try again</div>');  
        }  
        return false;  
    }  
});
```

- ① Listen for submit event of review form
- ② Check for any missing values
- ③ Show or inject error message onto page if value is missing

#### ④ Prevent form from submitting if value is missing

For this to work we need to ensure that the form has an ID of `addReview` set so that the jQuery can listen for the correct event. We also need to add this script to the page so that the browser can run it.

#### **ADDING THE JQUERY TO THE PAGE**

We'll include this jQuery file at the end of the body, along with the other client-side JavaScript files. These are set in the `layout.pug` view in `app_server/views`, right at the very bottom. Add a new line below the others pointing to the new file, as shown in the following code snippet:

```
script(src='/bootstrap/js/bootstrap.min.js')
script(src='/javascripts/validation.js')
```

That's all there's to it. The form will now validate in the browser without the data being submitted anywhere, removing a page reload and any associated calls to the server.

**TIP** Client-side validation can seem like it's all that you need, but the other types are vital to the robustness of an application. JavaScript can be turned off in the browser, removing the ability to run this validation, or the validation could be bypassed and have data posted directly to either the form action URL or the API endpoint.

## 7.6 Summary

In this chapter we've covered

- Using the `request` module to make API calls from Express
- Making POST and GET requests to API endpoints
- Separating concerns by keeping rendering functions away from the API request logic
- Applying a simple pattern to the API logic in each controller
- Using the status code of the API response to check for success or failure
- Applying data validation in three places in the architecture, and when and why to use each

Coming up next in chapter 8 we're going to introduce Angular into the mix, and start playing with some interactive front-end components on top of the Express application.

# 8

## *Creating an Angular application with TypeScript*

### This chapter covers

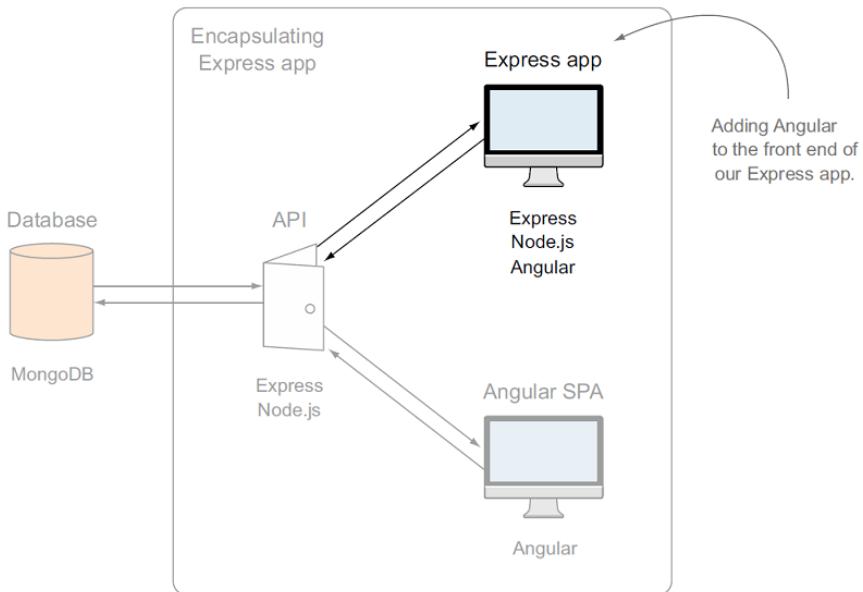
- Using the Angular CLI
- Creating an Angular application
- The basics of TypeScript
- Creating and using Angular components
- Binding data to HTML templates
- Getting data from an API
- Building an Angular application for production

Here it comes. It's now time to take a look at the final part in the MEAN stack: Angular! When getting started with Angular and TypeScript, it can feel like a different language at times, but TypeScript is really just a superset of JavaScript—so it's JavaScript with some additional bits and pieces. TypeScript is the preferred language for creating Angular applications. We'll cover what we need to as we go, and we'll be fairly comfortable with it by the end of this chapter.

To get into it all, we'll rebuild the list of locations shown in the homepage as an Angular application. We'll embed this little application into the Express-driven homepage - replacing the list delivered by Express - to serve two purposes:

1. We'll experience some of the building blocks of Angular without getting overwhelmed.
2. We'll see how Angular can be used to create a single component within an existing page or application.

Figure 8.1 shows where we are in the overall plan, adding Angular into the front end of the existing Express application.



**Figure 8.1** This chapter focuses on adding Angular to the front end of the existing Express application

The approach taken in this chapter is what you'd do if you wanted to enhance a page, project or application with a bit of Angular. Building a full application entirely in Angular is coming up in chapters 9 and 10 and builds upon what we learn in this chapter.

## 8.1 Getting up and running with Angular

In this section, we're going to create a skeleton Angular application, look at how it's put together, and explore some of the tools that come with it to help development. If you haven't done so yet, you'll need to have installed the Angular command-line interface (CLI) as described in appendix A to follow along.

We'll start by using the CLI to create a new application.

### 8.1.1 Using the command line to create a boilerplate Angular app

The easiest way to create a new Angular application is to use the Angular CLI: it will create a fully functional small application and generate a good folder structure.

The base command is simple: `ng new your-app-name`

Before we run the command to create our Angular app for Loc8r—which would create a new application called `your-app-name` with default settings in the current folder—we want to look at some options.

There are many options you can apply to this command, and you can see them by running `ng help` in the command line. The options we're interested in are the following:

- `--sg` to skip the default Git initialization and first commit; by default `ng new` will initialize the folder as a new Git repository, but we don't need to do that because we are going to create it inside an existing Git repo.
- `--st` to skip installation of some testing files; we're not covering unit tests in this book so we don't need these extra files.
- `--dir` to specify the folder where we want the application to be generated.

Putting this all together, we're going to use a command to create a boilerplate Angular application inside a new folder called `app_public`. This installs quite a lot of stuff, so will take a little while to run and you'll need to be online for it to work. Make sure that in terminal you are in the root folder of your Loc8r application before running the following command:

```
$ ng new loc8r-public --sg --st --dir app_public
```

**IMPROVEMENT** For those familiar with AngularJS (Angular 1.x), this is quite a change from the days of being able to download a single library file to start coding! The good news is that this new approach encourages better application architecture right out of the box.

When it's all installed, the contents of your `app_public` folder should look like Figure 8.1.

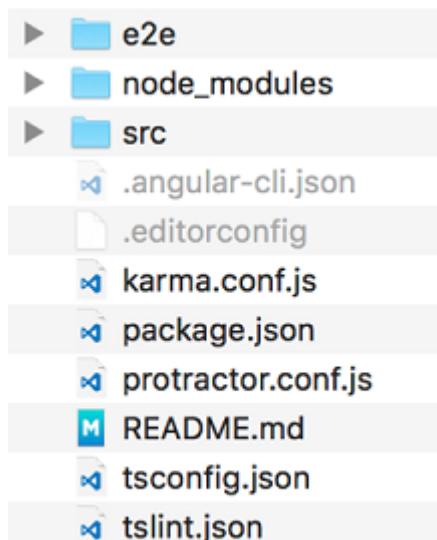


Figure 8.2 Default contents of a freshly generated Angular project

You may notice that this project has its own package.json file and node\_modules folder, so it looks very much like a Node application! The `src` folder is where we'll do most of our work.

### 8.1.2 Running the Angular app

What we've got here is a fully functional Angular app, albeit a rather minimal one. So let's run it, see what we've got, and then take a look under the hood.

To run the app, head to your `app_public` folder in terminal and run the following command:

```
$ ng serve
```

When you run this command, you'll see some notifications in terminal as Angular builds the application, finally ending with `webpack: Compiled successfully.` When you see this message, your app is ready to view on port 4200; to check it out, open your browser and go to `http://localhost:4200`. There's not much going on here, admittedly, but if you view the source or "inspect element," you should see something like figure 8.2.

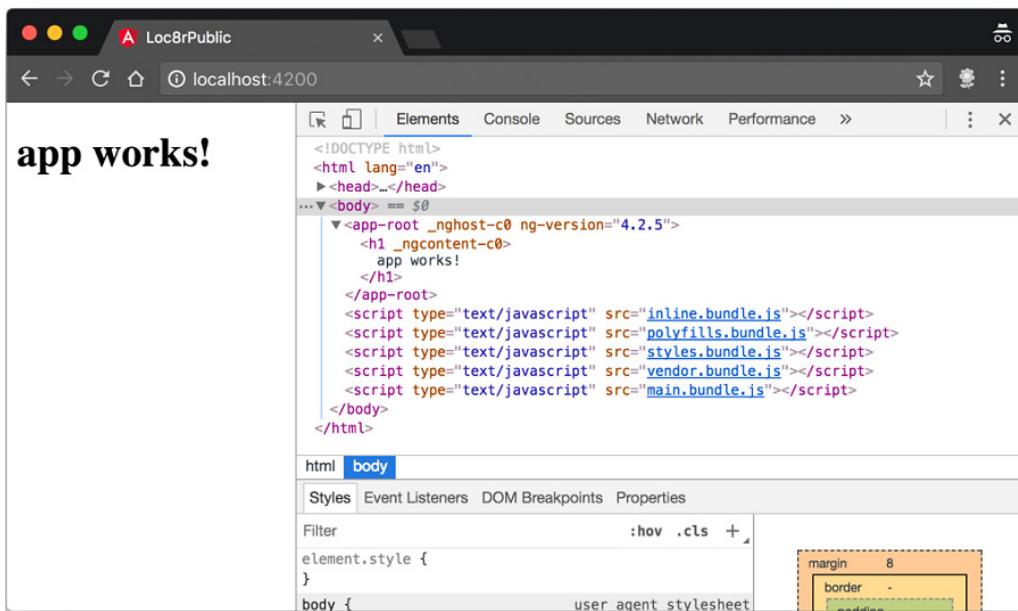


Figure 8.3 The auto-generated Angular app working in the browser, alongside the generated HTML.

As we can see, there's just an `<h1>` tag and a bunch of JavaScript files being referenced. However, take note of the `app-root` HTML tag - that's unusual and important. Remember this tag, because we'll come back to that in a moment, when we look at the source files.

### 8.1.3 The source code behind the application

Angular applications are built using **components**, which are compiled together into **modules**. *Component* and *module* are terms that are often used very loosely to label the building blocks of an application, but in Angular they have very specific meanings. A component handles a specific piece of functionality, and a module contains one or more components working together. This default example is a simple module with just one component in it.

Open the `src` folder in your editor and you'll see quite a few different files and folders. We'll start at the beginning and look at the `index.html` file in the `src` folder; it should look something like code listing 8.1.

#### **Listing 8.1 The default contents of the src/index.html file**

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Loc8rPublic</title> ①
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root>Loading...</app-root> ②
</body>
</html>
```

- ① The title has been created from the application name
- ② The only tag in the body is the app-root

There's not a huge amount in here aside from some very basic HTML scaffolding. We can see that Angular has populated the `title` tag for us (A), taking the application name we specified in the terminal command (`loc8r-public`) and turning it into camel-case. Second, we can see that `app-root` tag (B) that we noticed in the source of the running application. However, this time there is no `<h1>` tag inside it.

Let's dig a bit deeper and look inside the `app` folder (inside the `src` folder).

#### **THE MAIN MODULE**

Remember that we said *Angular applications are built using components, which are compiled together into modules?* A good place to start would be the module definition to see what's going on in there.

In `src/app` you'll find a file called `app.module.ts`. This is the central point of our Angular module, and it's where all of the components are brought together. At the moment, this looks like listing 8.2.

We won't go deep into the semantics of each part right now, just a high-level view of what each section is doing. In essence, this file does the following things:

- Imports various pieces of Angular functionality that the app will use
- Imports the components that the app will use
- Describes the module using a **decorator**
- Exports the module

### Decorators and dependency injection

A decorator is a way with ES6 and TypeScript to provide metadata and annotations to functions, modules, and classes. A common use case in Angular is to handle **dependency injection**, which is a way of saying “this module or class depends on this piece of functionality to run.”

You can see in listing 8.2 that we’re importing the modules `BrowserModule`, `FormsModule` and `HttpModule` into our module.

In this instance, the decorator is also declaring the components it contains, and which component should be used as the start point (`bootstrap`).

In this file, follow the journey of `AppComponent` - highlighted in bold in the listing. First, it is imported from the file system (you may recognize the `./` syntax from require and Node.js), before being both declared and bootstrapped inside the module *decorator*. For more information on decorators, check out the “Decorators and dependency injection” sidebar.

#### Listing 8.2 The default contents of the `src/app/app.module.ts` file

```
import { BrowserModule } from '@angular/platform-browser';          1
import { NgModule } from '@angular/core';                            1
import { FormsModule } from '@angular/forms';                        1
import { HttpModule } from '@angular/http';                          1

import { AppComponent } from './app.component';                      2

@NgModule({
  declarations: [
    AppComponent                                              3
  ],
  imports: [
    BrowserModule,                                                 3
    FormsModule,                                                   3
    HttpModule                                                    3
  ],
  providers: [],                                                     3
  bootstrap: [AppComponent]                                     4
})
export class AppModule {}                                         5
```

- ① Import various Angular modules that the application will use
- ② Import a component from the file system
- ③ Describe the module using a “decorator”
- ④ ... including the entry point into the application
- ⑤ Export the module

So then, this is the main module, and we can see from the `bootstrap` line in the decorator that the entry point into the application itself is the `AppComponent` component. We can also see from the `import` statement where this component lives in the file system - in this case it's in the same folder as this module definition.

Let's check it out.

### **THE DEFAULT BOOTSTRAPPED COMPONENT**

We're still looking in the `app_public/src/app` folder here, and alongside the module file we can see three `app.component` files:

- `app.component.css`
- `app.component.html`
- `app.component.ts`

These are the typical files for any component. The CSS and HTML files define the styles and markup for the component, and the TS file defines the behavior in TypeScript.

The CSS file is empty, but the HTML contains the following code:

```
<h1>
  {{title}}
</h1>
```

This makes some sense, as we think back to when we inspected the elements within the browser we saw an `<h1>` tag with some content. In Angular, double curly brackets are used to denote a binding between the data and the view. So here the variable `title` is being bound, as the contents of the `<h1>` tag. To see where this `title` variable is being defined, we need to look inside the component definition file - `app.component.ts` - which is shown in full in listing 8.3.

This component file will do three main things:

1. Import what it needs from Angular
2. Decorate the component - giving it the information that the app needs to run it
3. Export the component as a class

#### **Listing 8.3 The default contents of `app.component.ts`**

```
import { Component } from '@angular/core';          ①

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {                         ②
  title = 'app works!';                          ③
}
```

- ① Import “Component” from the Angular core

- 2 Decorate the component
- 3 Export the component as a class

So this is a pretty simple file, but the syntax is still a bit alien if you're used to plain JavaScript. If we look inside it though, we can see some interesting information and can see the pieces coming together.

Starting with the decorator, we can see the HTML and CSS files being referenced, but we can also see `selector: 'app-root'`. Ah-ha! That's the name of the tag we found in the `index.html` file! And when we inspected the elements we saw that tag, with an `<h1>` and some content inside, which matches our `app.component.html` file. Okay, it's coming together.

Next, we see the class `AppComponent` being exported, which we've already seen imported and bootstrapped in the module definition. Finally, there's the definition of `title` - we saw the binding in the HTML file for the component - and the value of `app` works! which we saw when running it in the browser. Note that there is no `var`, `const` or `let` associated with `title`; this is because inside a class definition you define **class members** as opposed to variables.

### **TYING IT ALL TOGETHER**

Okay, we've seen a lot here, so let's just quickly recap how it all ties together.

1. The component `AppComponent` comprises three files: TypeScript, HTML and CSS.
2. The TypeScript file is the key part of the component, defining the functionality referencing the other files and declaring which selector (HTML tag) it will bind to.
3. The component TypeScript file exports the `AppComponent` class.
4. The module file imports the `AppComponent` class from the component TypeScript file, and declares it as the entry point into the application.
5. The module file also imports various pieces of native Angular functionality.

This is all illustrated in figure 8.3.

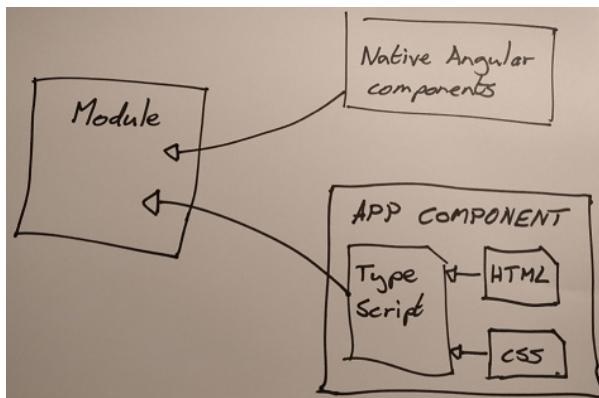


Figure 8.4 How the pieces of the simple Angular app fit together

This gives us a good understanding of how this simple app is constructed. But when we viewed the source in the browser earlier, none of the files we've just looked at were referenced, and we just saw a few JavaScript files. So what's going on? How did the TypeScript files become JavaScript in the browser?

### THE ANGULAR BUILD PROCESS

Currently browsers don't support TypeScript—only JavaScript—and not even ES6 fully yet. But writing in TypeScript gives you much more robust code. And although this sample application is very small, you can look into the future a little bit and see that if you have an application with several components, then that's a lot of separate files to deal with. You don't want to have to specify all of these in your HTML source.

Angular deals with these issues by using a **build** process to take all of the separate TypeScript files, convert them into vanilla JavaScript, and put them all into one file. This file is called `main.bundle.js`; if you take a look at the sources in the browser, you'll be able to find `title = 'app works!'` in there, as shown in figure 8.4.

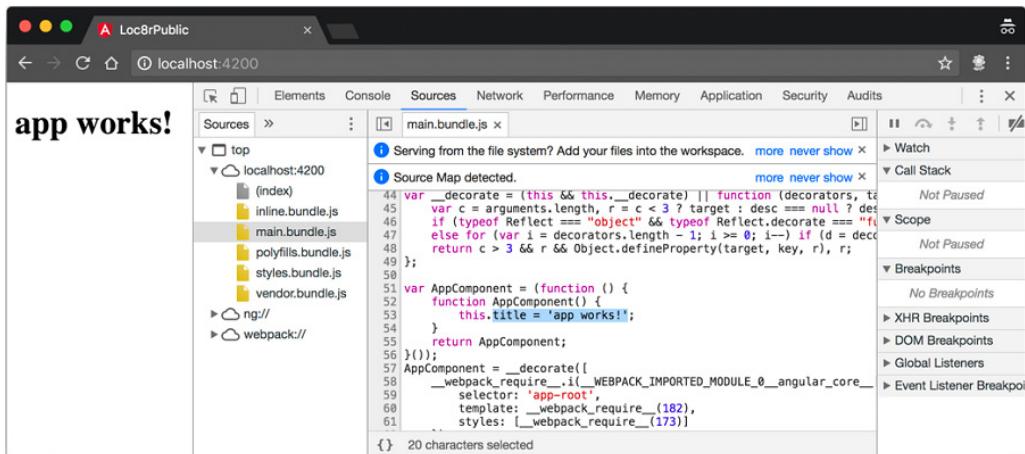
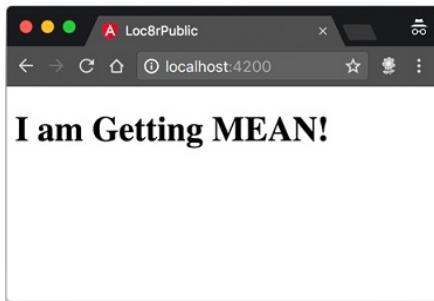


Figure 8.5 Finding the component definition inside the built JavaScript code

At the moment, we are using the `ng serve` command to compile, build, and deliver the Angular application to the browser on port 4200. This runs in memory: you won't find these built files inside the application code anywhere. When it comes to building a final version, we'll use a different command, `ng build`, but more on that later when we need it.

For development, `ng serve` is perfect, because not only does it give us this browser environment, it also watches the source code for changes and rebuilds and refreshes the application when it has changed. You can see this in action by changing 'app works' in

src/app/app.component.ts to 'I am Getting MEAN!' Head back to the application in the browser and you'll see that the content has changed, as in Figure 8.5.



**Figure 8.6** ng serve rebuilds and reloads the application when the source code changes

So, `ng serve` really helps us in the development process by removing the need to manually build and refresh with every change. Now that we know enough about Angular to be dangerous, we'll make the move into building something for Loc8r. We'll uncover more about Angular and TypeScript as we go, and it will all start to become more familiar.

## 8.2 Working with Angular components

We're going to start with building the listing section of the homepage, which we'll embed into the Express application. It's an example of how you can add some Angular functionality to an existing site, which is a very common requirement in large enterprise sites where you're not likely to have complete control over everything. In the following chapters, we'll build on this foundational knowledge and see how to build a standalone single-page application (SPA) in Angular.

We'll begin by creating a new component.

### 8.2.1 Create a new home-list component

You can create all of the files manually, or you can use the Angular CLI to help you out. We'll take advantage of this to create a skeleton component. In terminal, from within the `app_public` folder, run the following command:

```
$ ng generate component home-list
```

This will create a new folder called `home-list` within the `src` folder. Create the TypeScript, HTML, and CSS files inside it, and also update the `app.module.ts` file to tell the module about the new component. You'll also see a `spec.ts` file in the new component folder; this is a template for unit testing, but we're not covering that here so can ignore it for now. Angular CLI will output into terminal confirmations of all these actions.

## MAKE IT THE DEFAULT COMPONENT

This new `home-list` component is going to be the basis for this Angular module, so we need to make it the default component. We do this inside the `app.module.ts` file, by changing the `bootstrap` value inside the module decorator from `AppComponent` to `HomeListComponent`.

The `AppComponent` component is no longer needed, so we can remove the import statement, remove it from the declarations, and even delete the files. The changes to `app.module.ts` can be seen in listing 8.4.

### **Listing 8.4 Changes made to `app.module.ts` changing to the new component**

```
import { HomeListComponent } from './home-list/home-list.component'; ①

@NgModule({
  declarations: [
    HomeListComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [HomeListComponent] ③
}) ②
```

- ① This line was added by the Angular CLI; delete the `AppComponent` import as it is no longer needed
- ② Delete `AppComponent` from the declarations array
- ③ Change `AppComponent` to `HomeListComponent` for the bootstrap value

If you run `ng serve`, or still have it running, you'll see in the browser that now we just see "Loading..." displayed in the window and that there's an error in the JavaScript console. There are actually several errors and a lot of red text that can seem quite intimidating, but the very first line is very helpful because it says, "The selector "app-home-list" did not match any elements."

If we think back to the original component, we remember that `selector` defines the tag on the page that the component will bind to. We've changed the component but not the tag on the page!

## SET THE HTML TAG FOR THE COMPONENT

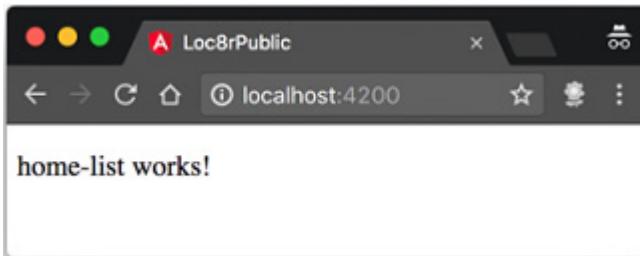
Just to ensure that we use the right tag, open up the `home-list.component.ts` file and check out the component decorator, which should look something like this:

```
@Component({
  selector: 'app-home-list',
  templateUrl: './home-list.component.html',
  styleUrls: ['./home-list.component.css']
})
```

Here we can see that the selector is `app-home-list` so that's what we need to use. We could change it if we wanted to have a different naming convention but this will work for us. Open up the `index.html` file in the `src` folder and change the `app-root` tag to `app-home-list` so that it looks like this:

```
<body>
  <app-home-list>Loading...</app-home-list>
</body>
```

Now check the browser—from now on I'll assume you have `ng serve` running whenever we check out the browser—and see that the page has changed to now say, "home-list works!" as shown in figure 8.6.



**Figure 8.7 Confirmation that the new home-list component is working as the default in the application**

Now that our component is there, let's start work on making it look like it should.

### 8.2.2 Create the HTML template

Following a similar approach to how we built the Express application, we'll start off by creating some static HTML with hard-coded data. This way we make sure that everything is working properly before we try to get the data from the API.

Fortunately for this component, we've already created the markup and the styles, so we just need to transfer them to Angular.

#### **GET THE HTML MARKUP**

We can't copy and paste the HTML directly from the Express source code because it's in Pug format, and it's also templated to use data bindings. For now we want the full HTML including data.

The easiest way for us to get the HTML is to run the Express app and go to the home page in a browser. Different browsers will have slightly different ways of getting the HTML, but will be similar to the following procedure in Chrome:

1. Right click in the HTML area and select *inspect element*
2. Highlight the `<div class="card">` element

### 3. Select *Copy* and then *Copy outer html*

Paste this into home-list.component.html, replacing the existing contents, and you should see something like listing 8.5.

#### **Listing 8.5 Some static HTML for home-list.component.html to get started**

```
<div class="card">
  <div class="card-block">
    <h4>
      <a href="/location/590d8dc7a7cb5b8e3f1bfc48">Costy</a>
      <small>&nbsp;
        <i class="fa fa-star-o"></i>
        <i class="fa fa-star-o"></i>
        <i class="fa fa-star-o"></i>
        <i class="fa fa-star-o"></i>
        <i class="fa fa-star-o"></i>
      </small>
      <span class="badge badge-pill badge-default float-right">14.0km</span>
    </h4>
    <p class="address">High Street, Reading</p>
    <div class="facilities">
      <span class="badge badge-warning">hot drinks</span>
      <span class="badge badge-warning">food</span>
      <span class="badge badge-warning">power</span>
    </div>
  </div>
</div>
```

If you take a look in the browser when this is saved, you'll be able to see the contents, but it won't look very nice at all. Let's add the styles.

#### **BRING IN THE STYLES**

Like the HTML, the CSS styles already exist in the Express application we just need to access them. We could update the index.html file to directly access them from localhost:3000, but certain browsers will give you a *CORS* warning if you try this because the Angular development app and the Express app are running on different ports. See the sidebar on CORS if this is new to you.

#### **What is CORS?**

Browsers are not allowed to access or request certain resources from a different domain, including requesting font files and making AJAX calls. This is known as the **same-origin policy**.

**CORS - cross-origin resource sharing** - is a mechanism that allows this to happen, but can be set only from the server that hosts the resources. If this denies you, there's nothing you can do from the browser side to change it.

To allow access to the resources, the server must be set to respond with a new HTTP header called `Access-Control-Allow-Origin` with a value that matches the requesting domain.

Not all browsers give a CORS warning for just a different port, but to avoid the problem all together, we'll just grab all the styles and fonts and drop them into the Angular app. So simply copy the `fonts` and `stylesheets` folders from `/public` folder and paste them into the `assets` folder in `app_public`.

Next, reference these CSS files in the `index.html` file (in `app_public`), as shown in listing 8.6.

#### **Listing 8.6 Adding the CSS files to index.html for the Angular app**

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Loc8rPublic</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">

  <link rel="stylesheet" href="assets/stylesheets/bootstrap.min.css">
  <link rel="stylesheet" href="assets/stylesheets/font-awesome.min.css">
  <link rel="stylesheet" href="assets/stylesheets/style.css">

</head>
<body>
  <app-home-list>Loading...</app-home-list>
</body>
</html>
```

With the styles in place, we can now take a look at the browser and see something like figure 8.7.

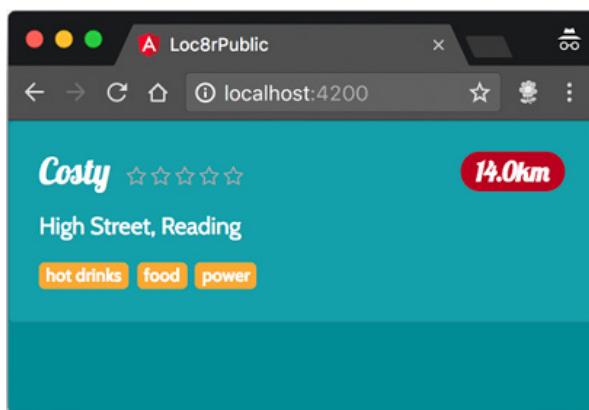


Figure 8.8 The Angular app displaying static content and using the styles and fonts

**NOTE** When building an application to sit inside another page - like we are here - the application will use the CSS of that containing page. The copies of the stylesheets we have here are for development use only, so that our module looks right as we build it. When building an SPA however, the final application will use the stylesheets inside the Angular app.

Now that we've got our homepage component looking about right, let's move on to making the HTML smarter by moving the hardcoded data out.

### 8.2.3 Moving data out of the template into the code

As we saw earlier in this chapter, with Angular you can define a class member inside the component code and bind it to the HTML using curly braces. So for example we could add this to home-list.component.ts to define the name of a location:

```
export class HomeListComponent implements OnInit {
  constructor() { }

  name = 'Costy';

  ngOnInit() { }
}
```

And then we could have this display in the HTML by replacing the location name with the binding as shown in bold here:

```
<a href="/location/590d8dc7a7cb5b8e3f1bfc48">{{name}}</a>
```

The end result of this would be that the browser displays the same as before, but now a part of the data is coming from the code and being bound to the template; it is no longer hardcoded HTML.

This is good, and shows us the way forward, but we need a lot more data for a location, and a better way to manage it. For this we need to use a **class**.

#### DEFINING A CLASS TO GIVE STRUCTURE TO DATA

In Angular a class is used to define the structure of a data object - in terms of what we've already learned you could think of it as similar to a simple Mongoose schema. So essentially a list of the pieces of data we expect an object to hold, and the *type* of value.

The type here is important. One thing that JavaScript doesn't have is that ability to state what type of value can be assigned to a given variable; it's really easy to change the value from a string to a number to a boolean - JavaScript doesn't care! But TypeScript does care, and it can help your code be more robust by making sure that you're always using the correct type of data for each variable. It's called *TypeScript* for a reason! See the *Types in TypeScript* sidebar for a list of the types available.

## Types in TypeScript

The different data types that TypeScript accepts are as follows:

- **string** - text values
- **number** - any numerical value, integer and decimal are treated the same
- **boolean** - true or false
- **array** - an array of a given type of data
- **enum** - a way of giving friendly names to a set of numeric values
- **any** - can be anything, like how JavaScript is by default
- **void** - the absence of a type, typically used for function that don't return anything

Defining a class is a simple task, and we'll do it at the top of the home-list.component.ts file, after the initial `import` statement but before the component decorator. To define a class and make it accessible, simply export it, give it a name, and then list out the names of the data items along with their expected datatypes as shown in listing 8.7.

### **Listing 8.7 Defining the Location class in home-list.component.ts**

```
import { Component, OnInit } from '@angular/core';

export class Location {
  _id: string;          1
  name: string;         2
  distance: number;    2
  address: string;     2
  rating: number;       2
  facilities: [string]; 3
}
```

- ① Create and export a class called "Location"
- ② Define the class members and their types ...
- ③ ... including an array of strings

With this done, we have defined the data we expect to see in our location objects. In fact - and this is important - each object defined with the class `Location` *must* have a value for each item specified. There is no concept of optional values in class definitions.

Now we've defined a class, let's see how to use it.

### **CREATE AN INSTANCE OF THE LOCATION CLASS**

When declaring variables and class members in TypeScript you should state the type of data as well as the name, just like we did when defining the properties of the Location class. This is done in the format `variableName: variableType = variableValue`.

For example, when we added `name = 'Costy'` to the home list component to try it out, we should really have added `name: string = 'Costy'` instead. This would have told TypeScript that name should only ever be a string value.

We do the same when creating a variable or class member that is an instance of a class, but in this case stating that the “type” is the name of the class. Listing 8.8 shows how to add a `location` class member with the type `Location` to the home list component, giving it all of the values it needs. The common way to describe this is to say `location` is an instance of type `Location`.

#### **Listing 8.8 Defining a location with the Location class in home-list.component.ts**

```
export class HomeListComponent implements OnInit {

    constructor() { }

    location: Location = {
        _id: '590d8dc7a7cb5b8e3f1bfc48',
        name: 'Costy',
        distance: 14.0,
        address: 'High Street, Reading',
        rating: 3,
        facilities: ['hot drinks', 'food', 'power']
    };

    ngOnInit() {
    }

}
```

A little later on we'll look at `constructor` and `ngOnInit`, why they are there and what they can be used for. But for now we can ignore them and focus on the new class member we've created. That's got all of the data that we need for one of the homepage listings, so next we'll use this data in the HTML.

#### **8.2.4 Using class member data in the HTML template**

As a quick recap, we've already seen how to bind data exposed from the component class in the HTML template using curly braces, for example `{{title}}`. Now our data is a little more complex and we need to access the properties of the class member, which we can do using the standard JavaScript dot syntax. For example, `location.name` will give the value of the `name` property.

Listing 8.9 highlights some of the quick and easy changes to make to the HTML template to bring the data in.

#### **Listing 8.9 Binding the first pieces of data in home-list.component.html**

```
<div class="card">
    <div class="card-block">
        <h4>
            <a href="/location/{{location._id}}">{{location.name}}</a>
            <small>&ampnbsp
                <i class="fa fa-star-o"></i>
                <i class="fa fa-star-o"></i>
            </small>
        </h4>
    </div>
</div>
```

```

<i class="fa fa-star-o"></i>
<i class="fa fa-star-o"></i>
<i class="fa fa-star-o"></i>
</small>
<span class="badge badge-pill badge-default float-right">{{location.distance}}km</span>
</h4>
<p class="address">{{location.address}}</p>
<div class="facilities">
  <span class="badge badge-warning">hot drinks</span>
  <span class="badge badge-warning">food</span>
  <span class="badge badge-warning">power</span>
</div>
</div>
</div>

```

Here we have four single pieces of data being bound into the HTML template. The facilities and the star rating are going to take a bit more work. Let's start with the facilities and looping through an array of data.

#### **FACILITIES: LOOPING THROUGH AN ARRAY OF ITEMS IN A HTML TEMPLATE**

In the TypeScript file we defined facilities as an array of strings like this: `['hot drinks', 'food', 'power']`. Now we're going to see how Angular can help us loop through these and create a `span` tag for each facility in the array.

The secret here is to use an Angular directive called `*ngFor`. When applied to a HTML tag and given an array of data, it will loop through the array creating an element for each entry. To access the value or properties of each item, you need to define a variable that Angular can use as it goes through the loop.

Listing 8.10 shows how to use the `*ngFor` directive to loop through the `location.facilities` array, assigning and use the variable `facility` to access the value.

#### **Listing 8.10 Using `*ngFor` to loop through an array in `home-list.component.html`**

```

<div class="facilities">
  <span *ngFor="let facility of location.facilities" class="badge badge-warning">{{facility}}</span>
</div>

```

The `*` is important, without it Angular won't perform the loop. With the `*` it will repeat the `<span>` and everything in it. Given the data facilities: `['hot drinks', 'food', 'power']` this will output:

```

<span class="badge badge-warning">hot drinks</span>
<span class="badge badge-warning">food</span>
<span class="badge badge-warning">power</span>

```

Note that Angular will create some additional comments and tag attributes which you can see in figure 8.8, along with the output in the browser.

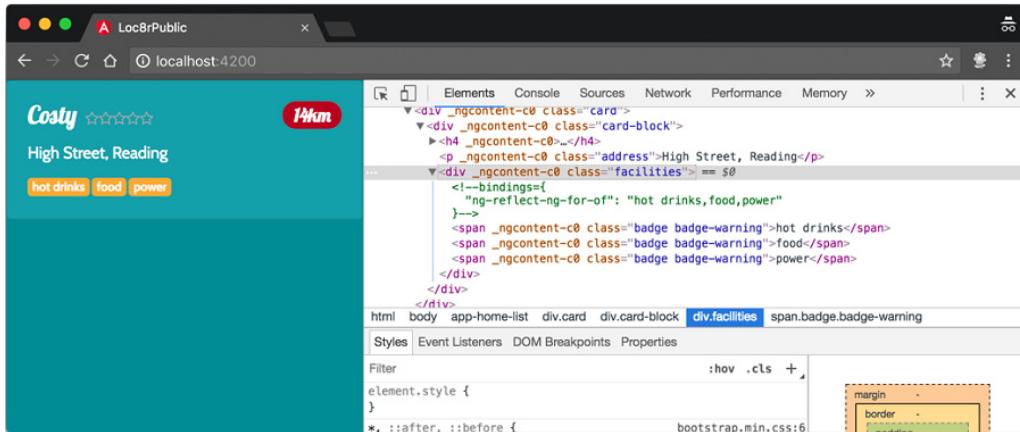


Figure 8.9 The output of Angular looping through the array of facilities

Okay, so that's the facilities done let's move on to the rating stars.

### RATING STARS: USING ANGULAR EXPRESSIONS TO SET CSS CLASSES

So far, the data bindings we've used have been simple - just one variable name or property within the double curly braces. With Angular we can also use simple expressions inside a binding; for example you could join two strings together `{} 'Getting' + 'MEAN' {}` or a simple math operation `{} Math.floor(14.65) {}`.

For the rating stars, each star is defined with a Font Awesome class: `fa-star` for a solid star and `fa-star-o` for an outline. What we want to do is set the classes using Angular, making sure that we have the correct number of solid and hollow stars to convey the rating.

To achieve this, we'll use a JavaScript ternary operator, which is a shorthand for a simple `if / else` expression. Using the first star as an example we want to say "if the rating is less than 1 make the star hollow, otherwise make it solid". As an if statement that looks like this:

```
if (location.rating < 1) {
  return 'fa-star-o';
} else {
  return 'fa-star';
}
```

Translated into a ternary operator the same expression look like this:

```
{} location.rating < 1 ? 'fa-star-o' : 'fa-star' {}
```

Flowing this logic through into the `<i>` tags that make up the rating stars - and putting the expressions into Angular bindings - looks like listing 8.11. Note that each expression has a different number to show the correct stars, and that we're always outputting `fa-star` so have taken it out of the expression.

### Listing 8.11 Binding the ternary expressions to generate rating star classes

```
<small>&nbsp;
  <i class="fa fa-star{{ location.rating < 1 ? '-o' : '' }}"></i>
  <i class="fa fa-star{{ location.rating < 2 ? '-o' : '' }}"></i>
  <i class="fa fa-star{{ location.rating < 3 ? '-o' : '' }}"></i>
  <i class="fa fa-star{{ location.rating < 4 ? '-o' : '' }}"></i>
  <i class="fa fa-star{{ location.rating < 5 ? '-o' : '' }}"></i>
</small>
```

You can validate that this is working correctly in the browser and you'll see something like figure 8.9.

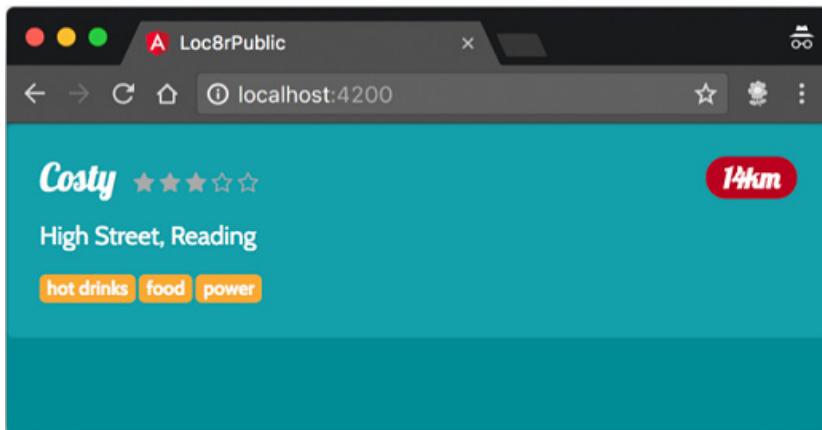


Figure 8.10 Showing the rating stars correctly, using Angular expression bindings to generate the correct class

Looking good! Just one more piece of data to deal with: the distance.

#### FORMATTING DATA USING PIPES

Angular gives us a way to format data within the binding, using what are known as **pipes**. If you're familiar with AngularJS, these used to be called filters. There are several built-in pipes in Angular, including date and currency formatting, and uppercase, lowercase, and title case string transformations.

Pipes are applied inside a binding by adding the pipe character (|) after the variable or expression to be bound, followed by the name of the pipe. If we wanted to display the address of a location in uppercase for example, we could simply add the uppercase binding like this:

```
<p class="address">{{location.address | uppercase}}</p>
```

We don't want to do that, but we could do if we wanted to!

A pipe that can be useful for debugging is the JSON pipe; this turns a JSON object into a string so that it can be displayed in the browser. If we weren't sure what data is coming

through in the `location` object, we could temporarily bind to it somewhere in the HTML and add the JSON pipe.

Some pipes can also take options to define how they will work. Let's take the currency pipe for example. You can apply the currency pipe without any options like this:

```
 {{ 12.3485 | currency }}
```

This will assume a default currency of US Dollars, and round up the digit to the near cent. In this example, the output would be `USD12.35`.

You can apply options to this pipe to change the currency and display the symbol instead of the currency code. Pipe options are specified directly after the pipe name, separated by colons. The order of the options is important. The first option for the currency pipe is the currency code itself, to change the currency; the second option is a Boolean to state whether or not to display the symbol.

If you wanted to display the currency as Euros, for example, and show the symbol instead of the code, you could use the pipe like this:

```
 {{ 12.3485 | currency:'EUR':true }}
```

This would now output `€12.35`. So that's how pipes work, and we'll work with some other default pipes as we build out the Loc8r application. Our need now is to format the distance into meters or kilometers, and for that we will need to create a custom pipe.

### DISTANCES: CREATING A CUSTOM PIPE

Before we create a new pipe to format the distance, we first need to make sure that the data we're passing it reflects what we'll get from the API. In our current mocked up data, we've got `14.0` so that it displays nicely. But the API returns the distance in meters, so update the distance in `home-list.component.ts` to reflect this—say `14000.1234` for example.

To create the boilerplate files for a custom pipe, we can use the Angular CLI. In terminal, from the `app_public` folder, run the following command:

```
ng generate pipe distance
```

This will generate two new files, `distance.pipe.ts` and `distance.pipe.spec.ts`, in the `src/app` folder. It will also tell the application about it by updating the `app.module.ts` folder. If you want to move your pipe files somewhere else, into a sub-folder for instance, you'd have to update `app.module.ts` to say where it has moved to. We'll leave it where it is for now.

The boilerplate pipe file, `distance.pipe.ts`, looks like this:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'distance'
})
export class DistancePipe implements PipeTransform {

  transform(value: any, args?: any): any {
```

```

        return null;
    }
}

```

This structure should be starting to look familiar. We've got the imports at the top, followed by the decorator, with the export class at the end. It's the contents of the class that we're really interested in here, in particular that `transform` function.

At first glance, this looks a bit odd, a bit complicated with all of the colons and `any` all over the place. But this is TypeScript doing what it does - defining the types for variables. The contents of braces `value: any, args?: any` is saying that the function *accepts* a parameter `value` of any type, and other arguments of any type. The third `: any` - after the braces - is defining the type of the *return* value of the function.

We want to change these, as our `distance` function will accept a number and return a string. To do so, update the `transform` function like this:

```

transform(distance: number): string {
    return null;
}

```

Note that we've changed the name of the parameter to `distance`. We have actually already written the code to format the `distance` in Node, so we can literally copy it from `/app_server/controllers/locations.js` and paste it in here. We want the `_isNumeric` helper along with the contents of the `_formatDistance` function. When that's done, the `transform` function looks like listing 8.12.

#### **Listing 8.12 Creating the distance format pipe in `distance.pipe.ts`**

```

transform(distance: number): string {
    const _isNumeric = function (n) {
        return !isNaN(parseFloat(n)) && isFinite(n);
    };

    if (distance && _isNumeric(distance)) {
        let thisDistance = '0';
        let unit = 'm';
        if (distance > 1000) {
            thisDistance = (distance / 1000).toFixed(1);
            unit = 'km';
        } else {
            thisDistance = Math.floor(distance).toString();
        }
        return thisDistance + unit;
    } else {
        return '?';
    }
}

```

**MEAP NOTE:** If you've followed along from early code, remove the `parseFloat` from the following line, because it prevents TypeScript from compiling; it's already a floating point number:

```
thisDistance = parseFloat(distance / 1000).toFixed(1);
```

Note that all of the code - including the helper function - is inside the `transform` function. All that's left now is to update the binding to use our new pipe and also remove the km from the template. The following snippet shows the updated binding from `home-list.component.html`.

```
<span class="badge badge-pill badge-default float-right">{{location.distance | distance}}</span>
```

We can also check this out in the browser, which is shown in figure 8.11.

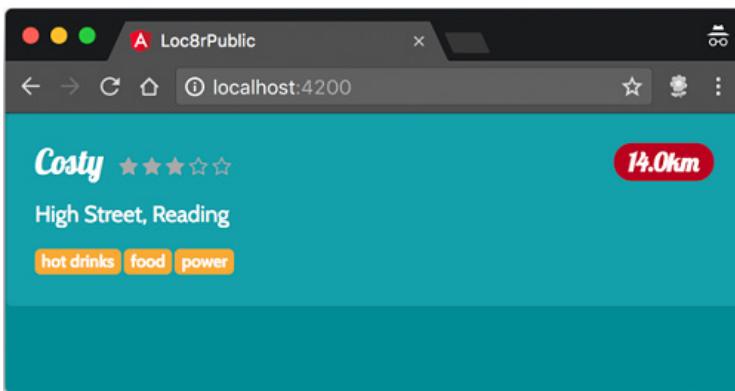


Figure 8.11 Using the Angular pipe to format the distance supplied in meters

Play around with the data in the component definition and test that it displays as we think it should. This looks good, and we've now got all of the data bindings set with all of the data being supplied by the component definition. However, this is just a single item and our API will return an array of multiple items - it is a list after all! Let's update it to work as a list.

### **WORKING WITH MULTIPLE INSTANCES OF A CLASS**

The data for our single location is defined as `location` of type `Location`. Don't read that out loud! The construct - without data - looks like this:

```
location: Location = {};
```

When we get the data from the API however, this will be an array, so we need to define an array of objects of type `Location`. The way to do this is to add square brackets after the class name, so that is looks like this construct:

```
locations: Location[] = [{} , {}];
```

If we take this approach - note that we change the member name to be the plural `locations` as we're dealing with an array - and update our `home-list` component to contain two locations it will look like listing 8.13.

**Figure 8.13 Changing the locations instantiation to be an array in `home-list.component.ts`**

```
locations: Location[] = [{  
    _id: '590d8dc7a7cb5b8e3f1bfc48',  
    name: 'Costy',  
    distance: 14000.1234,  
    address: 'High Street, Reading',  
    rating: 3,  
    facilities: ['hot drinks', 'food', 'power']  
}, {  
    _id: '590d8dc7a7cb5b8e3f1bfc48',  
    name: 'Starcups',  
    distance: 120.542,  
    address: 'High Street, Reading',  

```

Of course, having renamed `location` to `locations` and changed the type to an array we'll need to update the HTML template. We've already seen how to loop through an array using `*ngFor` and this is no different. In fact, all we need to do is add an `*ngFor` attribute to the outermost div of a single location - the one with the class of `card`. It will look like this:

```
<div class="card" *ngFor="let location of locations">
```

By defining the instance name of `location`, we don't need to change any of the data bindings inside the template, because that's what we were already using. And with that, we have multiple items showing in our list, as shown in figure 8.12.

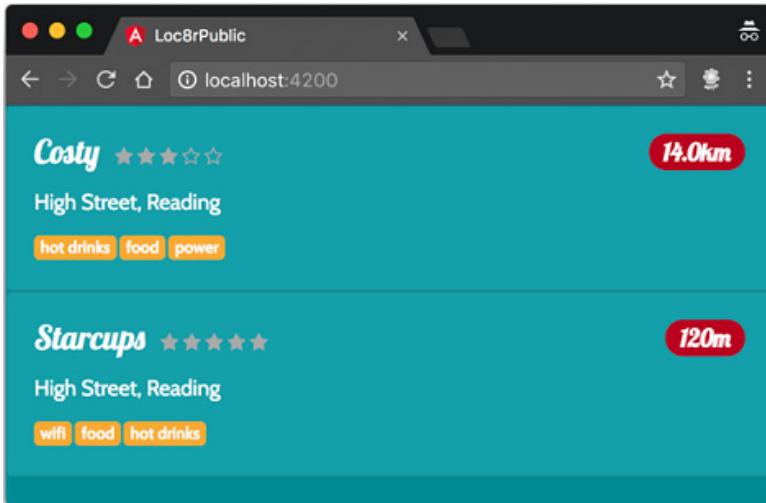


Figure 8.12 Updating the component to display multiple locations in a list

This is looking good and working well. Next step is to remove the hard-coded data entirely and call the API instead.

## 8.3 Getting data from an API

In this section, we're going to see how to call an API from an Angular application to get data. When we've got the data, we'll display it instead of the hard-coded data we currently have.

To interact with an API, we need to use another building block of Angular applications, a **service**. A service works in the background and isn't directly connected to the user interface like everything we've seen so far.

### 8.3.1 Creating a data service

Creating a service is done in the same way that we've created components and pipes so far - by using the Angular CLI. It's using the same `ng generate` command as before, this time followed by the options of `service` and `service name`. So, in the `app_public` folder run the following command in terminal:

```
$ ng generate service loc8r-data
```

This will generate the files for a new service called `loc8r-data` in the `app/src` folder. Terminal will confirm the creation of the files, but also note the warning: *WARNING Service is generated but not provided, it must be provided to be used.*

Unlike components and pipes, services are not automatically added into `app.module.ts` when generated, and with good reason. A service is not necessarily designed to be included application-wide; instead, you include it where you need it.

Before we worry about including it, let's look at the boilerplate code and build it out. The code layout should be looking familiar by now: imports followed by a decorator followed by the exported class.

```
import { Injectable } from '@angular/core';

@Injectable()
export class Loc8rDataService {

  constructor() { }

}
```

This boilerplate is very sparse, which isn't surprising because services can be used for many different things, not just requesting data from APIs. So we need to get started by giving the service some of the things it needs.

### **ENABLING HTTP REQUESTS AND PROMISE HANDLING IN A SERVICE**

In Angular, HTTP requests run asynchronously and return **observables**, but we want to wait until the data is complete before working with it so will convert to **promises**. For a quick explanation, see the "observables and promises" sidebar.

---

#### **Observables and promises**

Observables and promises are both great ways of handling asynchronous requests. Observables return chunks of data in a stream, whereas promises return a complete set of data. Angular includes the RxJS library for working with observables, including converting them into promises.

There's much more to RxJS and observables than we can cover here - a whole book in fact! Check out the Manning book *RxJS in Action* by Luis Atencio and Paul P. Daniels to learn more (<https://www.manning.com/books/rxjs-in-action>).

---

To set up the service to be able to make HTTP requests and return promises, we need to import some services from Angular and the included RxJS library. We'll also need to inject the HTTP service into our service.

Importing the HTTP service and RxJS promise support is done by updating the top of the `loc8r-data.service.ts` file like this:

```
import { Injectable } from '@angular/core';
import { Headers, Http } from '@angular/http';
import 'rxjs/add/operator/toPromise';
```

The second step is to inject the HTTP service into our service, so that we can use it and call the HTTP service methods. This is done using the **constructor** part of the boilerplate code. A class constructor defines the parameters that will be provided when the class is instantiated. Angular uses this to manage dependency injection, telling the class which other services or components it needs to run.

Injecting the service is very simple: it's just a case of defining the parameter name and its type. You can also state whether it's public or private—whether it will then be accessible from outside the class or kept within it. Private is the most common option.

So we'll inject `http` of type `Http` and keep it private by updating the constructor in `loc8r-data.service.ts` to look like this:

```
constructor(private http: Http) { }
```

With those two small updates, our data service will now be able to make HTTP requests and return promises.

### **CREATING THE METHOD TO GET DATA**

Our service will need a public method exposed so that the component can call it. At this point, the method doesn't need to accept any parameters but will return a promise containing an array of locations.

So inside the `Loc8rDataService` class, we want to define a method like this:

```
public getLocations(): Promise<Location[]> {
  // Our code will go here
}
```

This is all good, except our service doesn't know what `Location` is. We defined and exported the `Location` class in our `home-list` component, so we can import that into the service by adding this line along with the other imports.

```
import { Location } from './home-list/home-list.component';
```

And now we're ready to code the meat of our service.

### **MAKING HTTP REQUESTS**

Making the HTTP request to the API is actually quite straightforward, there are just a few steps to go through. In brief, the steps are:

1. Build the URL to call
2. Tell the HTTP service to make a request for the URL
3. Convert the observable response to a promise
4. Convert the response to JSON
5. Return the response
6. Catch, handle and return errors

Putting these steps into code looks like listing 8.14: this is all inside the Loc8rDataService class in loc8r-data.service.ts.

#### **Listing 8.14 Making and returning the HTTP request to our API in loc8r-data.service.ts**

```
private apiBaseUrl = 'http://localhost:3000/api/';  
  
public getLocations(): Promise<Location[]> {  
    const lng: number = -0.7992599;  
    const lat: number = 51.378091;  
    const maxDistance: number = 20;  
    const url: string =  
        `${this.apiBaseUrl}/locations?lng=${lng}&lat=${lat}&maxDistance=${maxDistance}  
    `;  
    return this.http  
        .get(url)  
        .toPromise()  
        .then(response => response.json() as Location[])  
        .catch(this.handleError);  
}  
  
private handleError(error: any): Promise<any> {  
    console.error('Something has gone wrong', error);  
    return Promise.reject(error.message || error);  
}
```

- ① Build the URL to the API, using parameters for future enhancements
- ② Return the promise
- ③ Make the HTTP GET call to the URL we built
- ④ Convert the observable response to a promise
- ⑤ Convert the response to a JSON object of type Location
- ⑥ Handle and return any errors

Note that only the method we need to call from somewhere else, `getLocations`, is public; everything else is defined as private, so that they can't be accessed externally.

That's not a lot of code, but it's doing quite a lot. As you'll see is quite common with Angular, once you get your head around the setting up of components, classes, and services, a lot of the actual code can be quite simple because many of the common tasks have had the complexities abstracted away.

Now that our data service is created, it's time to use it from our home-list component.

### **8.3.2 Using a data service**

We're now at a point where we have an Angular component that can display an array of locations (which are currently hard-coded), an API that can return an array of locations, and also a service to call that API and expose the response. The missing link is between the component and the service.

### **IMPORT THE SERVICE INTO THE COMPONENT**

There are three steps for including the service into the component, all of which take place inside the home-list.component.ts file. We will need to import the service, inject the service, and then provide the service.

First then we need to import the service from the TypeScript file, which we do at the top of the component file directly under the existing import line like this:

```
import { Component, OnInit } from '@angular/core';
import { Loc8rDataService } from '../loc8r-data.service';
```

Note how we define a relative path to the service file with the `../` meaning “go up a level in the folder structure.” This means that if we move the service files to a different place, we need to remember to update the references in code.

The second step is to inject the service into the component, using the constructor like we’ve just seen inside the data service itself. This time though, we’ll update the constructor in home-list.component.ts by injecting `Loc8rDataService` of type `Loc8rDataService` and keeping it private, like this:

```
constructor(private loc8rDataService: Loc8rDataService) { }
```

Finally we need to deal with the “not provided” part of the warning we saw when we created the component. We need to provide it to the application by adding the class to the providers part of the component decorator, like this:

```
@Component({
  selector: 'app-home-list',
  templateUrl: './home-list.component.html',
  styleUrls: ['./home-list.component.css'],
  providers: [Loc8rDataService]
})
```

By the end of all of this, the top of the home-list.component.ts file should look like listing 8.14.

#### **Listing 8.14 Making our service available to the component in home-list.component.ts**

```
import { Component, OnInit } from '@angular/core';
import { Loc8rDataService } from '../loc8r-data.service'; ①

export class Location {
  _id: string;
  name: string;
  distance: number;
  address: string;
  rating: number;
  facilities: [string];
}

@Component({
  selector: 'app-home-list',
```

```

        templateUrl: './home-list.component.html',
        styleUrls: ['./home-list.component.css'],
        providers: [Loc8rDataService] ②
    })
export class HomeListComponent implements OnInit {
    constructor(private loc8rDataService: Loc8rDataService) {} ③
}

① Import the service from the source code file
② Provide the service to the component
③ Inject the service into the component using the constructor

```

Now that the service is created and brought into the component we can use it.

### USING THE SERVICE TO GET THE DATA

Inside the class, we'll create a private method to call our data service method and handle the promise response. When it has the promise response, this method can set the value of the locations array, which will automatically update in the HTML.

To show that this is working, we'll remove all of the hard-coded data from the component, and just declare `locations` to be of type `Location` with no value assigned. Pop the code from listing 8.15 into the `HomeListComponent` class definition in `home-list.component.ts`.

#### **Listing 8.15 Creating a function to call the data service from `home-list.component.ts`**

```

locations: Location[]; ①

private getLocations(): void {
    this.loc8rDataService
        .getLocations() ②
        .then(foundLocations => { ③
            this.locations = foundLocations;
        });
} ④

```

- ① Change `locations` declaration to have no default value
- ② Define a `getLocations` method that accepts no parameters and returns nothing
- ③ Call our data service method
- ④ Update the `locations` array with the contents of the response

Great stuff. This still won't work though, because we're not calling the private `getLocations` method in the component. That will be the next and final step, but we need to make sure it's done at the right time.

As we've started to see already, an Angular application is composed of many files. But we have no control over the order in which the files are put together, and therefore no direct control of the execution order. So we need to make sure that the service is only called after it is available. This is where that little empty `ngOnInit()` block comes into play.

`ngOnInit` is one of several Angular **lifecycle hooks**. When an Angular application is starting up and running, things happen in a very specific order to make sure that the

application maintains integrity and always does things the same way. The lifecycle hooks allow you listen to the process and take action at certain times.

The `ngOnInit` hook allows you to hook into when the component is initialized and ready. This is a good time to make that data call, because we know it is now safe to do so and the component is ready to run. So we simply need to just make a call to the local `getLocations` method in `home-list.component.ts`, like so:

```
ngOnInit() {
  this.getLocations();
}
```

Now the application will compile properly, run, and make the call to the API. Great! Except if you try it on certain browsers, no data comes through. If you open up the browser developer tools or JavaScript console you'll see a CORS warning because the Angular app and Express API are running on different ports.

### **ALLOWING CORS REQUESTS IN EXPRESS**

The CORS issue can't be fixed from the browser side, it has to be done on the server side. We need to change gears for a moment and drop back into Express.

Allowing cross-origin requests is fortunately quite simple. For every request made to the API, we need to add two HTTP headers `Access-Control-Allow-Origin` and `Access-Control-Allow-Headers`. The first of these headers can contain a specific URL that you will allow requests from, or a `*` as a wildcard to accept requests from any domain. We'll limit it to our Angular development application by specifying the URL and port.

Head right back up to `app.js` in the root of the application and add the following bold-font lines before the routes are used.

```
app.use('/api', function(req, res, next) {
  res.header('Access-Control-Allow-Origin', 'http://localhost:4200');
  res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type,
  Accept');
  next();
});
app.use('/', index);
app.use('/api', apiRoutes);
```

This will add the two headers and their values to the responses for all requests made to the API routes. So now, if you've still got your Express application running on port 3000 and your Angular application running on port 4200, you should see your data coming through into the browser, like in figure 8.12.

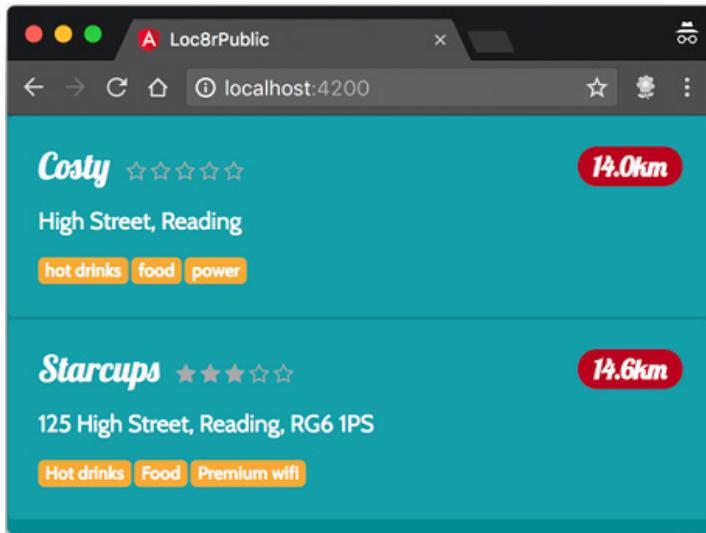


Figure 8.12 Our Angular component is now displaying data brought in from the API

This is great! We've built a nice little self-contained Angular application without too much trouble. Not a bad start, especially considering we've also been getting to grips with TypeScript throughout this chapter too. Let's just finish this off though, and embed it into our Express application.

## 8.4 Putting an Angular application into production

So far, we've been working with Angular in development mode while building our little application. But as soon as we stop `ng serve` from running, all we're left with is a bunch of source files and nothing we could include in a website. What we need to do now is build our application for production and add it to our homepage.

### 8.4.1 Building an Angular application for production

We've been using the `ng serve` command throughout this chapter to automatically rebuild our application and serve the compiled files from memory. Now we are going to use the `ng build` command to compile the files once and save them to disk.

The `ng build` command will generate all of the application files and put them in a folder called `dist`. This will be at the same level as the `src` folder and would be great, but if you run `ng serve` again afterward, it deletes the `dist` folder! This isn't very helpful, as you can imagine. But you can change this destination folder by using the option `--output-path` (or `-o` for short) when running the command. If you do this, your destination folder won't unexpectedly be deleted the next time you decide to run `ng serve`.

There are far too many build options for us to go through here—but you can check them out by running `ng help` in terminal—and the only other one we need to know right now is the one to specify that we want a production build (as opposed to a development build). This is specified by adding the flag `-prod` to the command.

So, to create a production build of our application in the folder `app_public/build`, run the following command in terminal from the `app_public` folder.

```
$ ng build -prod -op build
```

This will kick off the build process. If you get an error about not being able to find where `AppComponent` goes, it will probably be because the references were taken out of `app.module.ts` but the files weren't deleted. The fix here is to delete the old `app.component` files, because we're not using them anymore.

And that's it: the application is built for production! Now we need to include it in the Express application.

### 8.4.2 Using the Angular application from the Express site

To use the Angular application in our homepage, we just need to do a few small things in Express. First, we'll make it so that the `app_public` folder is set to be a static path, meaning that we can easily reference the files in the build folder from the browser. To do the second part, we'll update the Pug templates to include the JavaScript files in the build folder.

Easy right? Let's do it!

#### **DEFINING A STATIC PATH FOR THE ANGULAR APPLICATION**

We've already seen how Express defines folders to use for static resources, because the generator automatically defined the `public` folder to be static. We can do the same for the `app_public` folder by duplicating the line in `app.js` in the root of the application and setting the name to be `app_public`.

```
app.use(express.static(path.join(__dirname, 'public')));
app.use(express.static(path.join(__dirname, 'app_public')));
```

Now Express will serve static resources from either the `public` or `app_public` folders. Why define the whole `app_public` folder and not just the `build` subfolder to be a static resource? Well, the `build` folder also contains an `index.html` file. If this is included as a static resource, it will show up as the homepage as the static resources are checked before the other Express routes. This will be useful in the following chapters when we create the full Angular application, but it's not what we want right now. Right now, we want to use the Angular application *inside* our existing site, because we're just replacing a part of the homepage.

## REFERENCE THE COMPILED ANGULAR JAVASCRIPT FILES FROM THE HTML

We want to reference the Angular files on only the homepage, not on the other pages. The problem at the moment is that we can only include script files on the layout.pug template, because all of the other templates extend a small nested HTML part of this. There's nowhere to put new script tags.

A simple way to address this, is to create a new `block` in the layout.pug template in the best place to add new scripts. Then any other page that extends this layout will have an option for including page-specific scripts.

In layout.pug, include this line right at the very bottom, to define a new `block` called `scripts`:

```
block scripts
```

Make sure that the indentation matches that of the final `script` tag in the file; the desired outcome is that any page-specific scripts will be added at the bottom of the HTML `body`.

Next we need to use this new `block` from within locations-list.pug, and reference all four JavaScript files from the app\_public/build folder. It should look a bit like this, but you'll have different file names.

```
block scripts
  script(src='/build/inline.f0178fcd0cc34a5688b1.bundle.js')
  script(src='/build/polyfills.682313b6b06f69a5089e.bundle.js')
  script(src='/build/vendor.b5827d44ee5b8956cc5e.bundle.js')
  script(src='/build/main.ad6de91d9e2170cae9d4.bundle.js')
```

Almost there! We just need to tag in the HTML for the application to bind to.

## ADD THE HTML TAG TO BIND THE ANGULAR APP

If we cast our minds back to earlier in the chapter - or check in the source code - we'll remember that our app was bootstrapped into an HTML tag called `app-home-list`. So, all we want to do now is replace the list part of the homepage with our new holding tag.

In locations-list.pug, find the `each location` in `locations` section and either delete it or comment it out for reference. In its place, add `app-home-list Loading...`, ensuring the indentation is correct of course. This part of the template should now look something like this:

```
.row
.col-12.col-md-8
.error= message

app-home-list Loading...
```

Now we're done! Let's head to the browser, back on localhost:3000 and check out the homepage, now including our Angular application, which is getting data from our API.

If we've done everything properly, the page should look the same as before. To prove that the homepage is really using the Angular application, "inspect the element" of the list and see the `app-home-list` tag in there and all of Angular stuff inside. Figure 8.13 shows this in action.

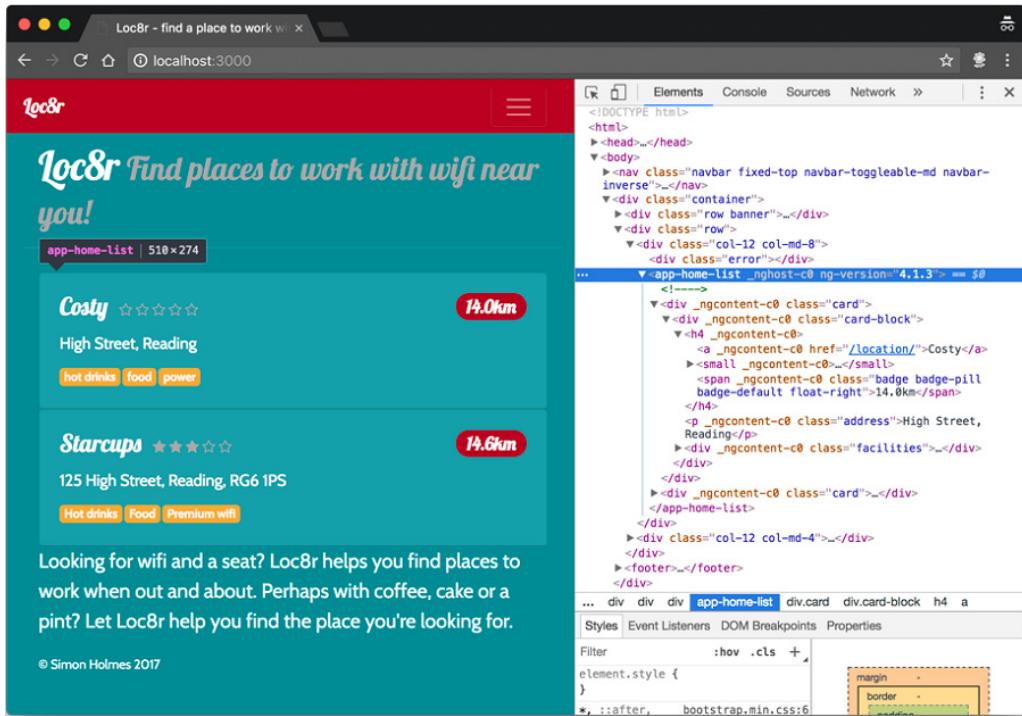


Figure 8.14 Validating that the homepage list is now using the Angular module

I love this stuff! It's great how all of the pieces fit together and work together. Now we really are Getting MEAN.

## 8.5 Summary

In this chapter, we've covered the following:

- Using the Angular CLI to generate boilerplate application, components and more
- Running a development version of the application using `ng serve`
- Working with TypeScript classes, importing and exporting, and using them to define types for variables
- Binding and manipulating data in templates
- Defining classes using TypeScript features like decorators and constructors
- Controlling the code execution flow using Angular lifecycle hooks
- How to create and use some of the Angular building blocks to put an application together, covering modules, components, pipes and services
- Using the Angular CLI to build an Angular application for production
- Including an Angular app in an existing Express site

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>

Coming up in the next chapter, we're going to start work on building Loc8r as a full Angular SPA.

# 9

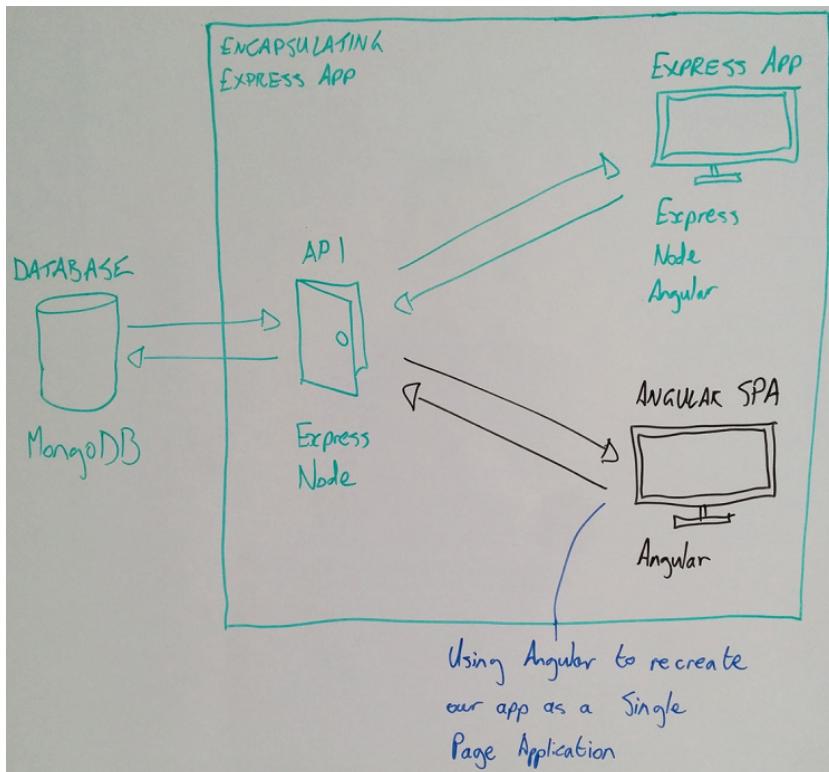
## *Building a single page application with Angular*

### This chapter covers

- Navigating between pages in Angular
- Working with the Angular router
- Architectural best practices for an SPA
- Building up views through multiple components
- Injecting HTML into bindings
- Working with browsers' native geolocation capabilities

We saw in chapter 8 how to use Angular to add functionality to an existing page. Over the next two chapters we're going to take Angular to the next level and use it to create a single-page application (SPA). This means that instead of running the entire application logic on the server using Express, we'll be running it all in the browser using Angular. For a reminder of some of the merits and considerations for using an SPA instead of a traditional approach flick through chapter 2. By the end of this chapter we'll have the framework for an SPA in place, and have the first part up and running by using Angular to route to the homepage and display the content.

Figure 9.1 shows where we're at in the overall plan, recreating the main application as an Angular SPA.



**Figure 9.1** This chapter will recreate the Loc8r application as an Angular SPA, moving the application logic from the back end to the front end.

In a normal development process you probably wouldn't create an entire application on the server and then recreate it as an SPA. Ideally, your early planning phases will have defined whether or not you want an SPA, enabling you to start in the appropriate technology. For the learning process we're going through now it's a good approach; we're already familiar with the functionality of the site and the layouts have already been created. This will let us focus on the more exciting prospect of seeing how to build a full Angular application.

In this chapter we'll start off by adding the Angular router to navigate between pages, before creating the homepage, the about page and adding geolocation functionality. As we add more components and functionality we'll explore various best practices, such as making reusable components and building up a modular application.

## 9.1 Adding navigation in an Angular SPA

In this section we are going to add the outline of the about page, and enable navigation between this new page and the homepage. The main focus of this section is the navigation, we'll complete the About page towards the end of this chapter in section 9.4.

You may remember that when we configured the Express application, we defined URL paths (routes) and used the Express router to map the routes to specific pieces of functionality. In Angular we do the same, but use the Angular router instead.

One big difference when using the Angular router is that the full application is already loaded in the browser, so when we navigate between pages the browser doesn't fully download all of the HTML, CSS and JavaScript each time. This makes navigating around a much quicker experience for the end user; the only things we normally have to wait for are data from API calls and any new images.

The first step is to import the Angular router into the application.

### 9.1.1 Import Angular router and define first route

The Angular router needs to be imported into `app.module.ts`, which is also where we define the routes. The router is imported from `@angular/router as RouterModule`, which should be placed with the other Angular imports as the top of `app.module.ts` as shown in listing 9.1.

#### **Listing 9.1 Add the RouterModule to the list of imports in app.module.ts**

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';
import { RouterModule } from '@angular/router';
```

In the same file, in the `@NgModule` decorator, all of these modules are listed in the imports section. We need to do the same with `RouterModule`, but in this case we also need to pass it the routing configuration we want.

#### **ROUTING CONFIGURATION**

The routing configuration is an array of objects, each object specifying one route. The properties for each route are:

- `path` - the URL path to match
- `component` - the name of the Angular component to use

The `path` property shouldn't contain any leading or trailing slashes - so instead of `/about/` you'd have `about` for example. It can also be an empty string to denote the homepage. Remember that the `base href` is set in the `index.html` file? We set ours to be `"/"` as we want everything running at the top level, but even if we had set it to have a value it wouldn't make

any difference to the routing configuration. In your routing configuration you should leave out anything in set in the `base href`.

We'll start by adding the configuration for the homepage, so `path` will be an empty string and `component` will be the name of our existing component: `HomeListComponent`. The configuration is passed to a `forRoot` method on the `RouterModule`, as shown in listing 9.2.

**Listing 9.2 Adding the routing configuration for the homepage to the decorator in `app.module.ts`**

```
@NgModule({
  declarations: [
    HomeListComponent,
    RatingStarsComponent,
    DistancePipe
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot([
      {
        path: '',
        component: HomeListComponent
      }
    ]),
    providers: [],
    bootstrap: [HomeListComponent]
})
```

- ① Add the `RouterModule` to the imports, calling the `forRoot` method
- ② Define the homepage route as an empty string
- ③ Specify the `HomeListComponent` as the one to use for this route

We've now imported Angular Router into our application, and told it which component to use for the homepage. However, we can't really test this out as we're also specifying the same component as the default component - note the line `bootstrap: [HomeListComponent]` in listing 9.2.

What we need to do is create a new default component, which we'll use to hold the navigation.

### 9.1.2 Create a component for the framework and navigation

To hold the navigation we need to create a new component, and make that the default component for the application. We'll also use this component to hold all of the framework HTML, much like we did in `layout.pug` in Express. In reality this is just three things: navigation, content container and footer.

First up, create a new component called `framework` by running the following in terminal, from the `app_public` directory.

```
$ ng generate component framework
```

This will of course create a new framework folder for us inside app\_public/src, and also generate all of the files we need. Find the framework.component.html file and add in all of the HTML seen in listing 9.3. This is pretty much what the HTML content of layout.pug would look like when converted to HTML

### **Listing 9.3 Add the HTML for the framework in framework.component.html**

```
<nav class="navbar fixed-top navbar-toggleable-md navbar-inverse"> ①
  <div class="container">
    <button type="button" data-toggle="collapse" data-target="#navbarMain"
      class="navbar-toggler navbar-toggler-right">
      <span class="navbar-toggler-icon"></span>
    </button>
    <a href="/" class="navbar-brand">Loc8r</a>
    <div id="navbarMain" class="navbar-collapse collapse">
      <ul class="navbar-nav mr-auto">
        <li class="nav-item"><a href="/about/" class="nav-link">About </a></li>
      </ul>
    </div>
  </div>
</nav>
<div class="container"> ②
  <footer>
    <div class="row">
      <div class="col-12"><small>&copy; Simon Holmes 2017</small></div>
    </div>
  </footer>
</div> ③
```

- ① Set up the navigation section
- ② Create the main container
- ③ Nest the footer inside the main container

Now that we've got the component set up we need to tell the application to use it as the default component, and where to put it in the HTML.

To set the new framework component as the default component update the bootstrap value in app.module.ts like so, replacing HomeListComponent with FrameworkComponent:

```
bootstrap: [FrameworkComponent]
```

Finally of course we need to update index.html to have the correct tag for this component rather than home list. Open up framework.component.ts and find the selector in the decorator as shown here. This gives us the name of the HTML tag we should use.

```
@Component({
  selector: 'app-framework',
  templateUrl: './framework.component.html',
  styleUrls: ['./framework.component.css']
})
```

So `app-framework` is the tag we need to have in `index.html` so that Angular knows where to put the framework component. Update `index.html` to look like listing 9.4.

#### **Listing 9.4 Updated index.html file to use the new framework component**

```
<body>
  <app-framework>Loading...</app-framework>
</body>
```

Now that our framework component is created and linked to the HTML, we can check it out in the browser as shown in figure 9.2. If you're not already, remember to run `nodemon` from the root folder of the application to get the API running, and also `ng serve` from the `app_public` folder to get the development version of the Angular app running.

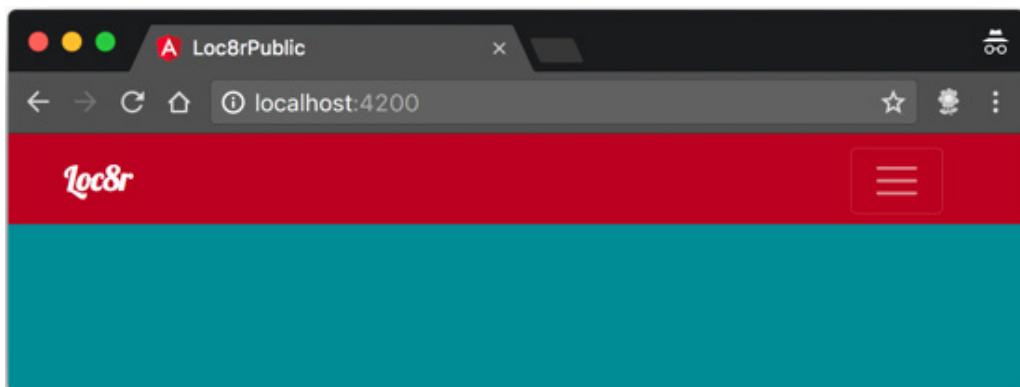


Figure 9.2 Showing the framework component by default instead of the listing

We can see the page header displaying, so we have success of sorts. Our new component works! But ... we don't see any content even though we're on the homepage route. If we open up the JavaScript console in the browser we'll also see an error: `Cannot find primary outlet to load 'HomeListComponent'`.

We've told the application to load `HomeListComponent` for the homepage route but haven't specified where it should be positioned in the HTML.

#### **9.1.3 Define where to display the content using router-outlet**

Specifying the destination for routed components is as simple as adding an empty tag pair in the HTML where you want it to go. This special tag is `<router-outlet>`. Angular will add the routed component *after* this tag, not inside it which you might expect if you're familiar with AngularJS.

Adding this empty tag pair to the correct place in the framework HTML - where we had block content in `layout.pug` - looks like listing 9.5.

### Listing 9.5 Adding router-outlet to framework.component.html

```
<div class="container">
  <router-outlet></router-outlet>
  <footer>
    <div class="row">
      <div class="col-12"><small>© Simon Holmes 2017</small></div>
    </div>
  </footer>
</div>
```

If we check out the browser we can now see the listing information as well as the framework. Inspecting the elements, as shown in figure 9.3, demonstrates that `<router-outlet>` remains empty, and that `<app-home-list>` has been injected afterwards.

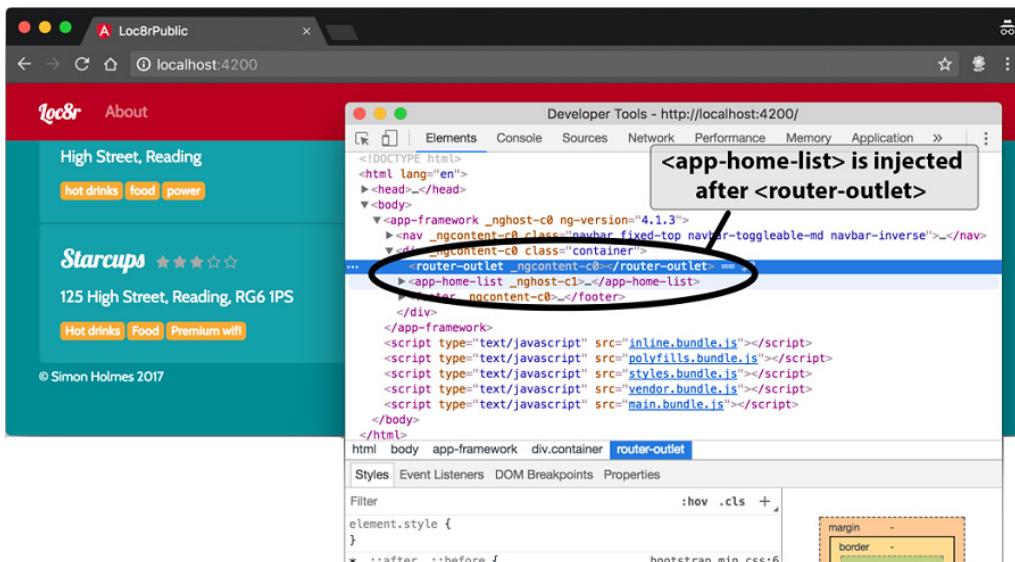


Figure 9.3 The routed component - the listing information - is now being displayed on the homepage route, with the HTML being injected after the `<router-outlet>` outlet tag.

We can now see the framework and the listing for the homepage, but it's not the homepage we know and love. It's missing a header and sidebar. We'll come back to this in section 9.2. First, let's see how navigation works.

#### 9.1.4 Navigating between pages

To see the navigation in action we'll update the Angular application so we can flip between the homepage and the about page. If we click the links right now and they won't work. To get the

navigation working we need to create an `about` component, define the `about` route and change the links in the navigation to something Angular can use.

Creating the `about` component using Angular CLI should be familiar by now. In terminal - in the `app_public` folder - run the following generate command:

```
$ ng generate component about
```

This will create the new component inside `app_public/src/app/about`. We'll leave it as it is for now so that we can focus on the navigation. In section 9.4 we'll return to the `about` page and build it out fully.

### DEFINING A NEW ROUTE

As with the homepage route, we need to configure the route for the `about` page in `app.module.ts`. We will need to specify the path for the route, as well as the name of the component. The path will be '`about`' - remember we don't need any leading or trailing slashes.

To make sure that we get the name of the component correct we can open up `about.component.ts` to find it in the export line: `export class AboutComponent implements OnInit`.

Know the path and component name we can now add the new route in `app.module.ts` as shown in listing 9.6.

#### **Listing 9.6 Defining the new about route in app.module.ts**

```
RouterModule.forRoot([
  {
    path: '',
    component: HomeListComponent
  },
  {
    path: 'about',
    component: AboutComponent
  }
])
```

If we open the browser directly onto `localhost:4200/about` we will get the `about` page, but the navigation links don't work properly yet. Let's fix them.

### SETTING ANGULAR NAVIGATION LINKS

When using links defined in the router, Angular doesn't want to see `href` attributes in the `<a>` tags, instead it looks for a directive called `routerLink`. Angular will take the value you give to `routerLink`, and create the `href` property.

The rules that applied to defining a path in the router also apply when setting the value for a `routerLink`. You don't need to include leading or trailing slashes, and bear in mind that you don't need to duplicate anything set in the `base href`.

Following these rules, updating the navigation links in `framework.component.html` looks like listing 9.7, replacing `href` attributes with `routerLink` directives, ensuring the values match what we have in the router definition in `app.module.ts`.

#### **Listing 9.7 Defining the navigation router links in framework.component.html**

```
<a routerLink="" class="navbar-brand">Loc8r</a>
<div id="navbarMain" class="navbar-collapse collapse">
  <ul class="navbar-nav mr-auto">
    <li class="nav-item">
      <a routerLink="about" class="nav-link">About </a>
    </li>
  </ul>
</div>
```

With this in place and saved we can now click between the two links as shown in figure 9.4.

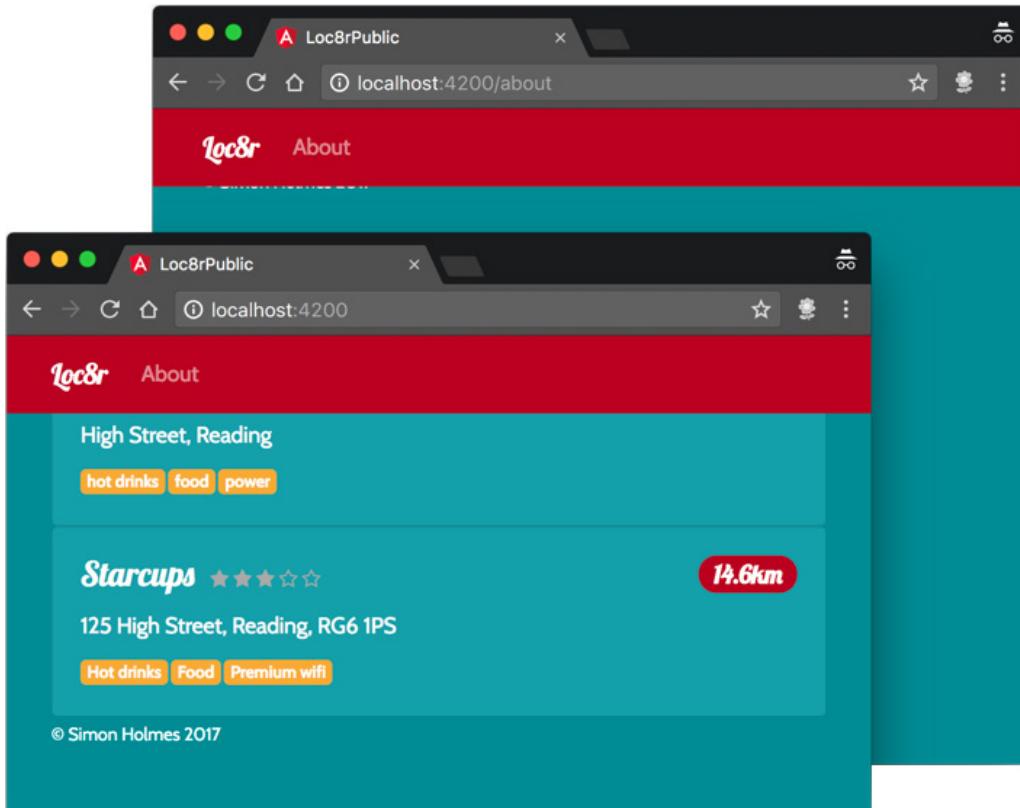


Figure 9.4 Using the navigation buttons to switch between the homepage and the about page - an Angular SPA!

Notice that the URL in the browser changes as normal, but the page doesn't reload or flicker when moving between the pages. If you check the network traffic when switching between these two pages, you'll only see calls to the API being made. You can also use the back and forward buttons on your browser and it will work just like a traditional website. Congratulations, you've just built a single page application!

Before we move on, let's quickly improve the navigation by adding "active" styles.

### 9.1.5 Adding active navigation styles

It is standard practice in web design to have an "active" class on navigation items, so that the link for the current page looks a bit different. It's a simple visual cue to help tell the user where they are. We've only got one link in our navigation, but it's still a worthwhile process.

Twitter Bootstrap has helper classes defined to create an active navigation state; we simply need to set the class `active` on the active link. As it's such a common requirement Angular also has a helper for this, a directive called `routerLinkActive`.

On an `<a>` tag containing a router link you can add the `routerLinkActive` directive and specify the name of the class you want to use for active links. In our case shown here, we will use the class `active`, in `framework.component.html`.

```
<a routerLink="about" routerLinkActive="active" class="nav-link">About </a>
```

Now when we visit the about page, the `<a>` tag has an extra class of `active` added to it, which Bootstrap displays as a stronger white color. This we can see in figure 9.5.

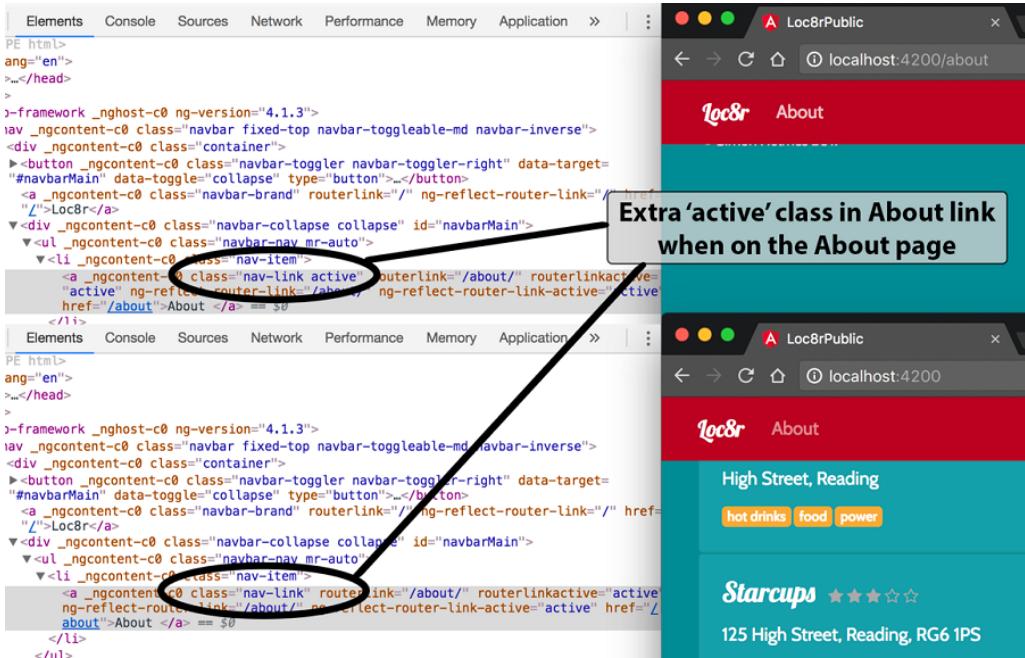


Figure 9.5 Seeing the active class in action; Angular adds and removes it from the link as navigation changes are made

And with that we've now covered the basics of the Angular router, creating a working navigation for our SPA. We can see that the views clearly need some work, so that's what we'll focus on for the next two sections.

## 9.2 Building a modular app using multiple nested components

In this section we're going to focus on building out the familiar homepage in Angular. To set ourselves up for success - and to follow Angular architectural best practices - we'll do this by creating several new components and nesting them as we need to. This will give us a very modular application, so that we can reuse pieces in different places of the application.

If we think about the homepage there are three main sections:

1. Page header
2. List of locations
3. Sidebar

We already have the list of location built as a component - that's our `home-list` component. What we'll need to do is create the header and the sidebar as two new components.

We'll also need to wrap all three of these components inside a main homepage component, so that we can ensure everything works together, has the correct layout and can be navigated to via the Angular router. Figure 9.6 shows an overlay of how these components fit together on top of the homepage design. See how we have the framework component on the outside, holding everything. Nested inside this is the homepage component to control the content area, with the page header, listing and sidebar components nested inside of it.

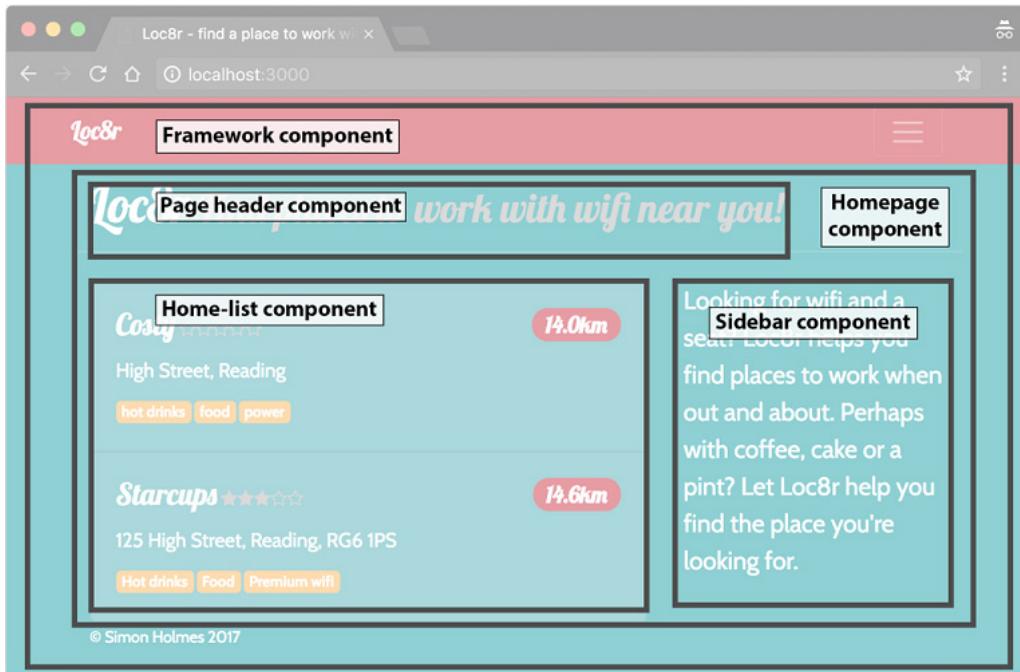


Figure 9.6 Breaking up the homepage layout into different components, using two levels of nesting

This is what we're going to build. We'll start with the homepage component.

### 9.2.1 Creating the main homepage component

The homepage component will contain all of the HTML and information for the homepage, so everything between the header and the footer. It is this component that we'll reference in the router for Angular to use whenever anybody requests the homepage.

Start by using the Angular CLI to generate the component in the now familiar way, in terminal from the app\_public folder.

```
$ ng generate component homepage
```

Next tell the router to use this component for the default home route by updating app.module.ts like so:

```
RouterModule.forRoot([
  {
    path: '',
    component: HomepageComponent
  },
  {
    path: 'about',
    component: AboutComponent
  }
])
```

In homepage.component.html just put the selector for the home-list component for a moment before checking it in the browser.

```
<app-home-list>Loading...</app-home-list>
```

If you look at the application in the browser it looks just like it did before, with the navigation bar, footer and listing section in between.

But we want to see all of the content for the homepage now; that's the page header, main content and sidebar. Taking the framework from the Pug templates and turning them into HTML looks like listing 9.8. Note that we're putting in the app-home-list component here, to display the listing section.

#### **Listing 9.8 Putting the HTML for the homepage content in homepage.component.html**

```
<div class="row banner">
  <div class="col-12">
    <h1>Loc8r
      <small>Find places to work with wifi near you!</small>
    </h1>
  </div>
</div>
<div class="row">
  <div class="col-12 col-md-8">
    <div class="error"></div>
    <app-home-list>Loading...</app-home-list>
  </div>
  <div class="col-12 col-md-4">
    <p class="lead">Looking for wifi and a seat? Loc8r helps you find places to work
      when out and about. Perhaps with coffee, cake or a pint? Let Loc8r help you
      find the place you're looking for.</p>
  </div>
</div>
```

① The page header

② Container for the homepage listing component

③ The sidebar

Now when we view it in the browser we get something like figure 9.7 - our good old familiar homepage!

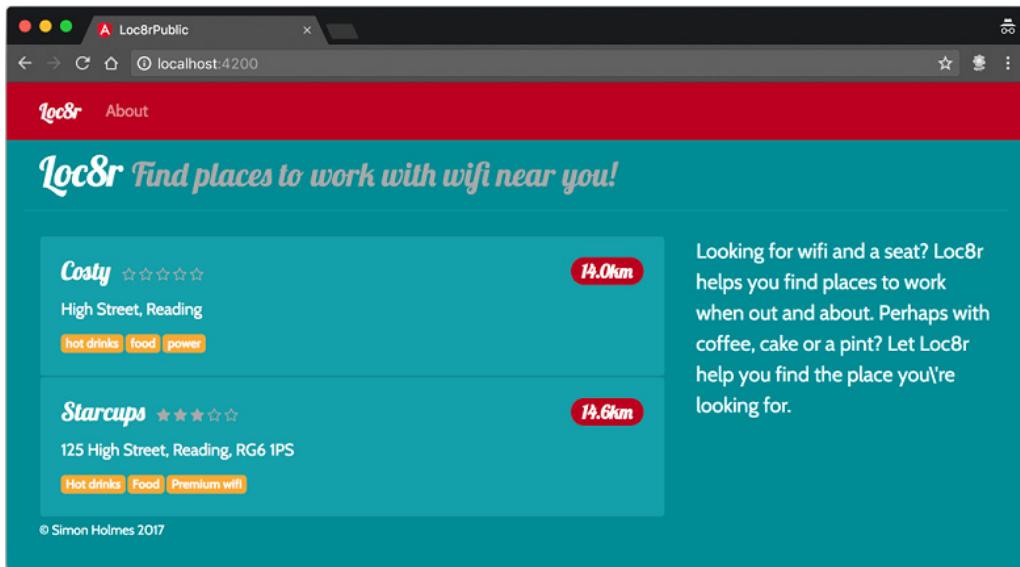


Figure 9.7 The homepage in Angular with the page header and sidebar hard coded in the homepage component

So everything is there and working correctly, including the home-list component nested inside the homepage component. But we can do better. The page header and sidebar are both repeated on other pages, albeit with different text content. We can follow some architectural best practice here and try to avoid repeating code by creating reusable components.

### 9.2.2 Create and use reusable sub-components

We are going to create the page header and sidebar as new components so that we don't end up copying the HTML into multiple different views. If the site grows to have dozens or hundreds of pages we wouldn't want to have to repeat the same HTML in each layout. This gets even worse if you need to make an update to the HTML in the future - it is much easier to make it in just one place, and is also much less prone to errors or missing something.

We'll make the components "smart" so that we can pass them different content to display. So really, in our case here, the reusable components are all about the HTML rather than the content. Let's start with the page header.

#### **CREATING THE PAGE HEADER COMPONENT**

The first step is the familiar component generation command in terminal:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

**Licensed to Jacob Munkholm Hansen <au518331@uni.au.dk>**

```
$ ng generate component page-header
```

Following that we'll take the header content from the homepage HTML and paste it into page-header.component.html.

```
<div class="row banner">
  <div class="col-12">
    <h1>Loc8r
      <small>Find places to work with wifi near you!</small>
    </h1>
  </div>
</div>
```

Then of course we need to reference this in the homepage.component.html instead of the full HTML currently there. To do this we need the correct tag, which we can find by looking for the selector in the page-header.component.ts file. In this case it is app-page-header, so that's what we'll use in the homepage component HTML as shown in listing 9.9.

### **Listing 9.9 Replacing the page header HTML with the component in homepage.component.html**

```
<app-page-header></app-page-header>
<div class="row">
  <div class="col-12 col-md-8">
    <div class="error"></div>
    <app-home-list>Loading...</app-home-list>
  </div>
  <div class="col-12 col-md-4">
    <p class="lead">Looking for wifi and a seat? Loc8r helps you find places to work
      when out and about. Perhaps with coffee, cake or a pint? Let Loc8r help you
      find the place you're looking for.</p>
  </div>
</div>
```

Good start, we've created the new page header component, but it still has hard coded content. Next we'll pass data to the page header from the homepage component.

#### **DEFINE THE DATA FOR THE PAGE HEADER COMPONENT ON THE HOMEPAGE**

We want to set the data for the homepage instance of the page header component from within the homepage component, so that we can pass it through.

Defining the data is pretty simple. In the homepage component class definition we create a new member to hold the data. We'll create a member called `pageContent` and nest the header inside as shown in listing 9.10. The class member is a simple JavaScript object with text data - note that the `strapline` content is shortened in this snippet to save trees.

### **Listing 9.10 Defining the homepage page header content in homepage.component.ts**

```
export class HomepageComponent implements OnInit {
  constructor() { }
```

```

ngOnInit() {
}

pageContent = {
  header : {
    title : 'Loc8r',
    strapline : 'Find places to work with wifi near you!'
  }
};

}

```

1

**① Create a new class member to hold the page header content**

The `header` is nested inside `pageContent` like this because very soon we'll add the sidebar content too, and having them both within the same member will keep the code neater. Now to pass this data to the page header component.

### PASS DATA INTO THE PAGE HEADER COMPONENT

The homepage class member `pageContent` is now available to the homepage HTML, but rather than use the data directly we want to pass it through to the page header component. Data is passed through to the nested component by a special binding in the HTML. The name of the binding is a property you define in the nested component, so it can be anything you want.

We'll bind the page header content to a property called `content` - this property doesn't exist yet, we'll define it in the next step. In `homepage.component.html` update the `<app-page-header>` to include the binding like this:

```
<app-page-header [content]="pageContent.header"></app-page-header>
```

Note that while the square brackets may not be valid HTML, that's okay here because Angular removes them before serving the HTML to the browser. The actual HTML the browser will receive is something like this: `<app-page-header _ngcontent-c6="" _nghost-c2="">`, which is valid HTML.

We're now passing data from the homepage component into the nested page header component; we just need to update the page header to accept and use this data.

### ACCEPTING AND DISPLAYING INCOMING DATA IN A COMPONENT

We now need to tell the `pageHeader` component that `content` should exist as a property and to get the value from the "outside". Technically, `content` is an *input* to the component.

Any property of a class needs to be defined, and this one is no different. Where it is different from what we've seen before is that it needs to be defined as an input property. To do that we need to import `Input` into the component from the Angular core, and use it as a decorator when we define the `content` member, as shown in listing 9.11.

**Listing 9.11 Telling page-header.component.ts to accept content as an Input**

```
import { Component, OnInit, Input } from '@angular/core'; ①

@Component({
  selector: 'app-page-header',
  templateUrl: './page-header.component.html',
  styleUrls: ['./page-header.component.css']
})
export class PageHeaderComponent implements OnInit {

  @Input() content: any; ②

  constructor() { }

  ngOnInit() {
  }

}
```

- ① Import Input from the Angular core  
② Define content as a class member that accepts an input of any type

When that is done the component will understand the data being sent to it from the homepage component and we'll be able to display it. To do that, simply replace the hardcoded text in page-header.component.html with the relevant Angular data bindings as shown in figure 9.12.

**Listing 9.12 Putting the data bindings in page-header.component.html**

```
<div class="row banner">
  <div class="col-12">
    <h1>{{ content.title }}</h1>
    <small>{{ content.strapline }}</small>
  </div>
</div>
```

And now we have a fully re-usable component for the page header, which can display the data sent to it from a parent component. This is an important building block of Angular application architecture that we've just completed. We'll cement the process by doing the same for the sidebar so that we can complete the homepage, and we'll run into a little hiccup along the way.

**CREATING THE SIDEBAR COMPONENT**

We won't dwell too long on the steps for setting up the sidebar component, as we've just done them for the page header.

First, generate the component:

```
$ ng generate component sidebar
```

Second, grab the sidebar HTML from `homepage.component.html` and paste it into `sidebar.component.html`. When you do, replace the text content with a binding to `content`.

```
<div class="col-12 col-md-4">
  <p class="lead">{{ content }}</p>
</div>
```

Third, allow the sidebar component to receive data by importing `Input` from Angular core, and defining the `content` property - of type `string` - with the `@Input` decorator.

```
import { Component, OnInit, Input } from '@angular/core';

@Component({
  selector: 'app-sidebar',
  templateUrl: './sidebar.component.html',
  styleUrls: ['./sidebar.component.css']
})
export class SidebarComponent implements OnInit {

  @Input() content: string;

  constructor() { }

  ngOnInit() {
  }

}
```

Fourth, update the `pageContent` member in `homepage.component.ts` to contain the sidebar data:

```
pageContent = {
  header : {
    title : 'Loc8r',
    strapline : 'Find places to work with wifi near you!'
  },
  sidebar : 'Looking for wifi and a seat? Loc8r helps you find places to work when
            out and about. Perhaps with coffee, cake or a pint? Let Loc8r help you find
            the place you\'re looking for.'
};
```

Fifth, update the `homepage.component.html` to use the new sidebar component and pass the data through as `content`.

```
<app-page-header [content]="pageContent.header"></app-page-header>
<div class="row">
  <div class="col-12 col-md-8">
    <div class="error"></div>
    <app-home-list>Loading...</app-home-list>
  </div>
  <app-sidebar [content]="pageContent.sidebar"></app-sidebar>
</div>
```

All done! But is it? If you view this in the browser you'll notice that no matter how wide you make your browser window - as shown in figure 9.8 - the sidebar is always below the content.

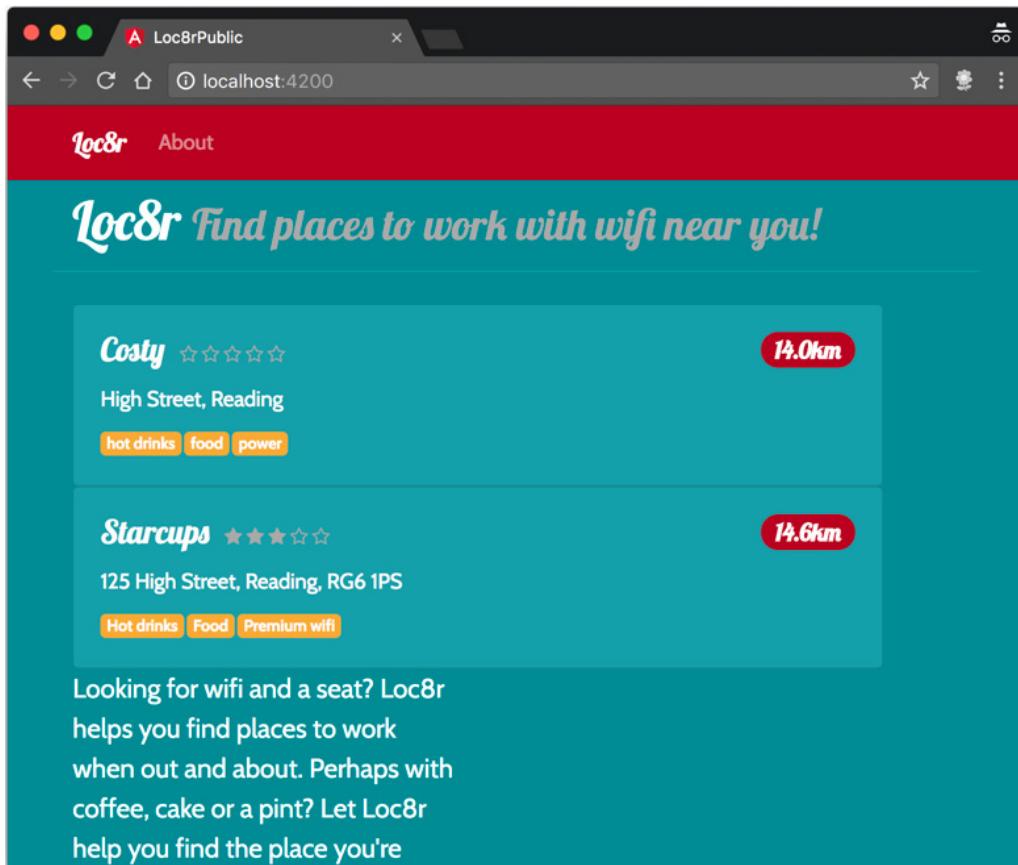


Figure 9.8 The new sidebar component is in and working, but is always below the main content instead of where it should be

This is because the position of the sidebar is defined by the classes on the `<div class="col-12 col-md-4">` element. But by putting this content inside a component we've wrapped it in a new tag `<app-sidebar>`, so Bootstrap is throwing the sidebar below as a new row.

This is something to look out for, especially when nesting components. But it's quite easy to fix.

#### **WORKING WITH ANGULAR ELEMENTS AND BOOTSTRAP LAYOUT CLASSES**

The problem we have is that the browser now sees this following HTML markup generated:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

**Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>**

```
<div class="col-12 col-md-8">
  <app-home-list>Loading...</app-home-list>
</div>
<app-sidebar [content]="pageContent.sidebar">
  <div class="col-12 col-md-4">
    <p class="lead">{{ content }}</p>
  </div>
</app-sidebar>
```

The Bootstrap `col` classes for the sidebar are in the wrong level in the hierarchy, so `<app-sidebar>` is being treated as a full-width column regardless of browser size. All we need to do, is move the classes from the `<div>` in `sidebar.component.html` to `<app-sidebar>` in `homepage.component.html`, so that `homepage.component.html` looks like listing 9.13.

### **Listing 9.13 Moving the sidebar classes into homepage.component.html**

```
<app-page-header [content]="pageContent.header"></app-page-header>
<div class="row">
  <div class="col-12 col-md-8">
    <app-home-list>Loading...</app-home-list>
  </div>
  <app-sidebar class="col-12 col-md-4" [content]="pageContent.sidebar"></app-sidebar>
</div>
```

With that done, we no longer need the `<div>` in `sidebar` markup, we can keep just the `<p>` and the content, so that `sidebar.component.html` looks like this:

```
<p class="lead">{{ content }}</p>
```

And with that, everything should now look right with the homepage, as shown in figure 9.9.

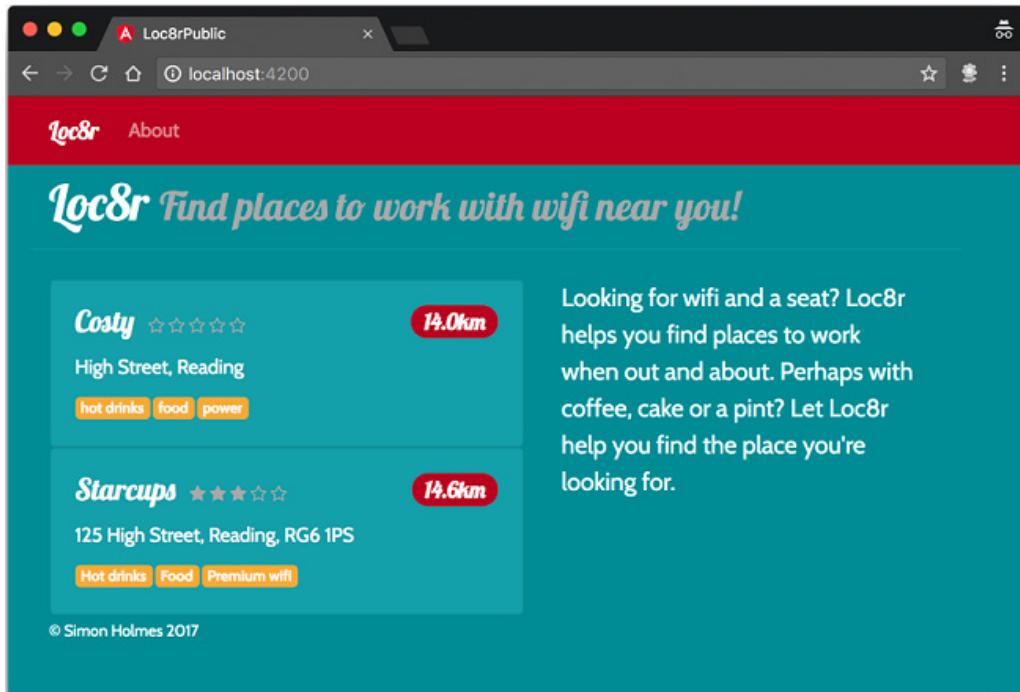


Figure 9.9 The completed homepage rendering correctly, constructed of multiple nested components

The homepage is looking good! There's something we've been missing so far though - wouldn't it be great if it could tell where you are, and find places nearby? Let's do it, and add geolocation to the homepage.

### 9.3 Adding geolocation to find places near you

The main premise of Loc8r is that it will be location-aware, and able to find places that are near you. So far we've been faking it by hard-coding geographic coordinates into the API requests. We're going to change that right now by adding in HTML5 geolocation.

To get this working we'll need to do the following:

- Add a call to the HTML5 location API into our Angular application
- Only look for places when we have the location
- Pass the coordinates to our Angular data service, removing the hard-coded location
- Output messages along the way so the user knows what's going on

Starting at the top, we'll add the geolocation JavaScript function by creating a new service.

### 9.3.1 Creating an Angular geolocation service

Being able to find the location of the user feels like something that would be reusable, in this project and other projects. So to snap it off as a piece of standalone functionality we'll create another service to hold this. As a rule, any code that's interacting with APIs, running logic, or performing operations should be externalized into services. **Leave the component to control the services, rather than perform the functions.**

To create the skeleton for geolocation service run the following in terminal from app\_public.

```
$ ng generate service geolocation
```

We won't get distracted by diving into the details of how the HTML5/JavaScript geolocation API works right now. We'll just say that modern browsers have a method on the `navigator` object that you can call to find the coordinates of the user. The user has to give permission for this to happen. The method accepts two parameters, a success callback and an error callback, and looks like the following:

```
navigator.geolocation.getCurrentPosition(cbSuccess, cbError);
```

We'll need to expose the standard geolocation script in a public method so that we can use it as a service. While we're here, we'll also error trap against the possibility that the current browser doesn't support this. The following listing shows the full code for `geolocation.service.ts`, providing a public `getPosition` method that other components can call.

#### **Listing 9.14 Create a geolocation service returning a method to get the current position**

```
import { Injectable } from '@angular/core';

@Injectable()
export class GeolocationService {

    constructor() { }

    public getPosition(cbSuccess, cbError, cbNoGeo): void {
        if (navigator.geolocation) {
            navigator.geolocation.getCurrentPosition(cbSuccess, cbError);
        } else {
            cbNoGeo();
        }
    }
}
```

1  
2  
2  
3  
3

- ➊ Define public member called `getPosition` that accepts three callback functions for success, error, and not supported
- ➋ If geolocation supported, call native method, passing through success and error callbacks
- ➌ If geolocation isn't supported, invoke not supported callback

That code gives us a geolocation service, with a public method `getPosition` that we can pass three callback functions to. This service will check to see whether the browser supports geolocation and then attempt to get the coordinates. The service will then call one of the three different callbacks depending on whether geolocation is supported and whether it was able to successfully obtain the coordinates.

The next step is to add it to the application.

### 9.3.2 Adding the geolocation service to the application

To use our new geolocation service we need to import it into the `home-list` component, just like we did for our data service. We will need to:

- Import the service into the component
- Add the service to the providers in the decorator
- Add the service to the class constructor

The following code listing highlights in bold the additions we need to make to the `home-list` component definition to import and register the geolocation service

#### **Listing 9.15 Updating `home-list.component.ts` to bring in the geolocation service**

```
import { Component, OnInit } from '@angular/core';
import { Loc8rDataService } from '../loc8r-data.service';
import { GeolocationService } from '../geolocation.service'; ①

export class Location {
  _id: string;
  name: string;
  distance: number;
  address: string;
  rating: number;
  facilities: [string];
}

@Component({
  selector: 'app-home-list',
  templateUrl: './home-list.component.html',
  styleUrls: ['./home-list.component.css'],
  providers: [Loc8rDataService, GeolocationService] ②
})
export class HomeListComponent implements OnInit {

  constructor(private loc8rDataService: Loc8rDataService, private geolocationService: GeolocationService) {} ③
}
```

- ① Import the geolocation service
- ② Add the service to the providers
- ③ Pass the service into the class constructor

Once this has been done, we'll be able to use the geolocation service from with our `home-list` component.

### 9.3.3 Using the geolocation service from the home-list component

The home-list component now has access to the geolocation service, so let's use it! Thinking back, our `getPosition` method in the service accepts three callback functions, so we'll need to create those before we can call the method.

As the geolocation process can take a few seconds before we even start searching the database for locations, we'll also want to provide some useful messages to the user so that they know what's going on.

We actually already have an element for messages in our HTML, but it's currently in `homepage.component.html` - we need it in `home-list.component.html`. Find the `<div class="error"></div>` in the homepage HTML and remove it. Then paste it into the top of `home-list.component.html` adding in a binding so that we can display messages like so:

```
<div class="error">{{message}}</div>
<div class="card" *ngFor="let location of locations">
```

With this we'll be able to use the message binding to keep the user up-to-date with what's happening. Now we're ready to create the callback functions.

#### CREATING THE GEOLOCATION CALLBACK FUNCTIONS

Inside the component we'll create three new private members, one for each of the possible geolocation outcomes:

- Successful geolocation attempt
- Unsuccessful geolocation attempt
- Geolocation not supported

We'll also update the messages being displayed to the user, letting them know that the system is doing something. This is particularly important now because geolocation can take a second or two.

The success callback will be the existing `getLocations` method, with some additional message-setting thrown in, and the other two will simply set error messages as shown in listing 9.16. As we'll be using the message binding from within these new functions, we'll also need to define it as a property of the class with type `string`.

#### **Listing 9.16 Setting up the geolocation callback functions in `home-list.component.ts`**

```
export class HomeListComponent implements OnInit {
    constructor(private loc8rDataService: Loc8rDataService, private geolocationService: GeolocationService) { }

    locations: Location[];
    message: string;

    private getLocations(position: any): void {
        this.message = 'Searching for nearby places';
```

①

②

```

    this.loc8rDataService
      .getLocations()
      .then(foundLocations => {
        this.message = foundLocations.length > 0 ? '' : 'No locations found'; ②
        this.locations = foundLocations;
      });
}

private showError(error: any): void { ③
  this.message = error.message;
};

private noGeo(): void { ④
  this.message = 'Geolocation not supported by this browser.';
};

ngOnInit() {
  this.getLocations();
}

}

```

- ① Define the message property of type string
- ② Set some messages inside the existing getLocations member
- ③ Function to run if geolocation is supported but not successful
- ④ Function to run if geolocation isn't supported by browser

We've got our three callback functions there, for success, failure and error. Now we need to use our geolocation service rather than calling `getLocations` on init of the component.

### **CALLING THE GEOLOCATION SERVICE**

In order to call the `getPosition` method of our geolocation service we'll need to create a new member in the `home-list` component and call it on init, instead of directly calling the `getLocations` method.

Our geolocation service accepts three callback parameters - `success`, `error` and `unsupported` - so we can add a new member to `home-list.component.ts` called `getPosition` that calls our service, passing through our callback functions. That should look like this:

```

private getPosition(): void {
  this.message = 'Getting your location...';
  this.geolocationService.getPosition(
    this.getLocations,
    this.showError,
    this.noGeo);
}

```

Then of course we need to call this when the component is initialized instead of the `getLocations` method, so replace the call in `ngOnInit` to be this new member:

```

ngOnInit() {
  this.getPosition();
}

```

Save this and head to the browser. You should see something like figure 9.10, where the browser now asks us for permission to access our location.

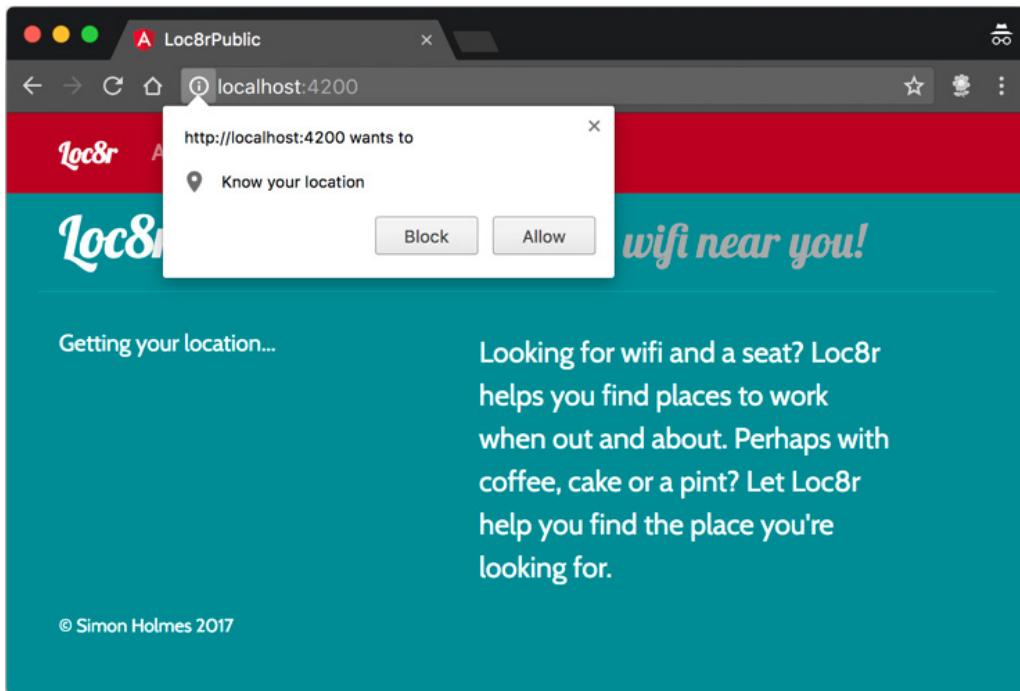


Figure 9.10 A successful call to our geolocation service is marked by a browser request to know our location

Great news! Until you hit Allow and the screen hangs on the “Getting your location...” message, quietly throwing a JavaScript error in the background. The error we’re getting says “Cannot set property ‘message’ of null” and looks like figure 9.11.

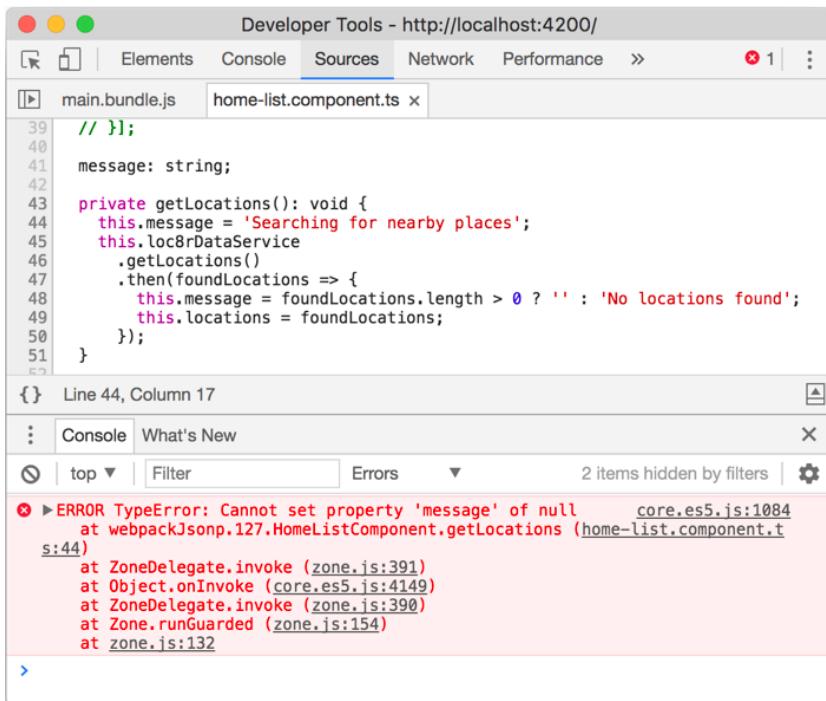


Figure 9.11 Error message shown when trying to set messages in the geolocation callback

This tells us what the problem is and where it occurs, which will help us fix it.

### **WORKING WITH THIS IN CALLBACKS ACROSS COMPONENTS AND SERVICES**

We can see from the error in figure 9.11 that it can't set `this.message` inside the `getLocations` callback because `this` is null. When passing the class member through as a callback we have lost the context of `this`, which was the instance of the class itself.

Luckily the fix is easy. We can send the context through by binding `this` to each callback as we send it. Where each callback function is passed `.bind(this)` to the end, as shown in the following snippet.

#### **Listing 9.17 Bind this to the geolocation callback functions in home-list.component.ts**

```
private getPosition(): void {
  this.message = 'Getting your location...';
  this.geolocationService.getPosition(
    this.getLocations.bind(this),
    this.showError.bind(this),
    this.noGeo.bind(this)
  );
}
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

**Licensed to Jacob Munkholm Hansen <au518331@uni.au.dk>**

Now we are binding the context of this to the callback function so that it exists when we need it. When we visit the browser again we have success! After displaying a few messages and getting our location, the browser displays our home-list again.

But we're not actually using the location yet. We're getting it, but doing nothing with it. Let's change that.

### **USING THE GEOLOCATION COORDINATES TO QUERY THE API**

In `home-list.component.ts` the `getPosition` method calls our geolocation service to get the coordinates. When it is successful it calls the `getLocations` method - again in `home-list.component.ts` - as a callback, passing the position as a parameter. We will need to update this callback to receive the position. This callback then calls our data service to search for locations. We'll need to pass the coordinates to the service, and then update the service to use these values when calling the API.

So we have just two things to update. Starting with `getLocations` in `home-list.component.ts`, we need to update it to accept a position parameter, extract the coordinates from it and pass them through to the data service. This is all highlighted in the following listing.

#### **Listing 9.19 Update `home-list.component.ts` to use the geolocation position**

```
private getLocations(position: any): void {  
    this.message = 'Searching for nearby places';  
    const lat: number = position.coords.latitude;  
    const lng: number = position.coords.longitude;  
    this.loc8rDataService  
        .getLocations(lat, lng)  
        .then(foundLocations => {  
            this.message = foundLocations.length > 0 ? '' : 'No locations found';  
            this.locations = foundLocations;  
        });  
}
```

- ① Accept the position as a parameter
- ② Extract the latitude and longitude coordinates from the position
- ③ Pass the coordinates to the data service call

We are now getting the position from the geolocation service, extracting the latitude and longitude coordinates and passing them to the data service. To get the last piece into place we need to update the data service to accept the coordinate parameters and use them instead of the hardcoded values, as shown in the following listing.

#### **Listing 9.20 Update `loc8r-data.service.ts` to use the geolocation coordinates**

```
public getLocations(lat: number, lng: number): Promise<Location[]> {  
    const maxDistance: number = 20;  
    const url: string =  
        `${this.apiBaseUrl}/locations?lng=${lng}&lat=${lat}&maxDistance=${maxDistance}`  
        ;
```

```

    return this.http
      .get(url)
      .toPromise()
      .then(response => response.json() as Location[])
      .catch(this.handleError);
}

```

- ① Accept lat and lng parameters of type number
- ② Delete the hardcoded values we had for lat and lng before

Now the coordinates are finding their way from the geolocation service to the API call, so we are now actually using Loc8r to find places near us! If you check it out in the browser - if you have added some places within 20km of where you are - you should now see them listed! It should look like figure 9.12, and you'll probably notice a slight change in the distance coordinates, depending on how accurate your test data was.

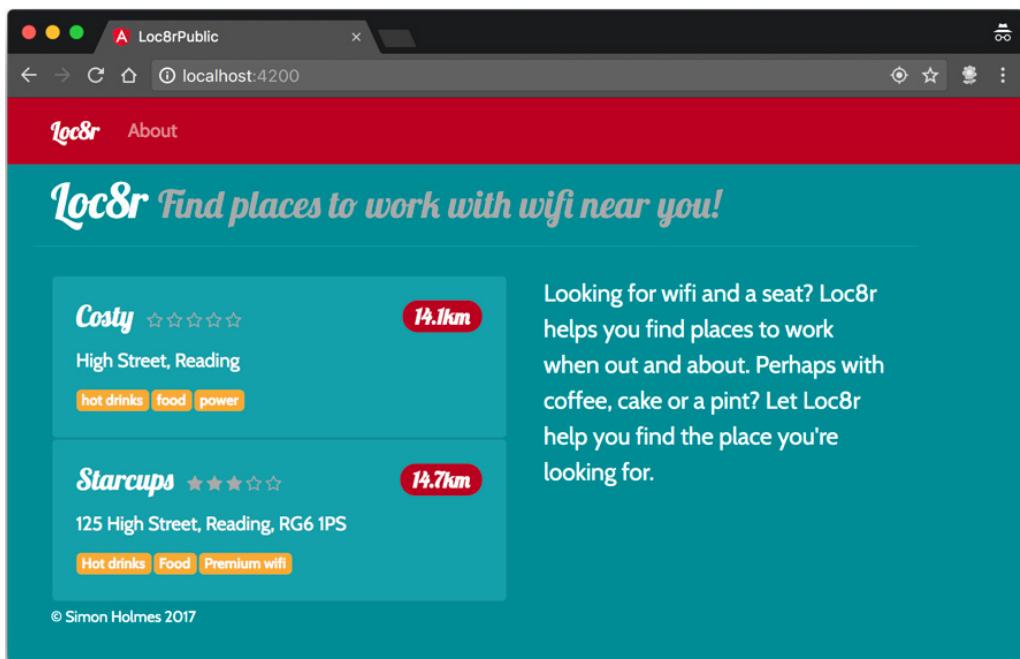


Figure 9.12 The Loc8r homepage, as an Angular app, using geolocation to find places nearby from our own API

And that's the last piece of the puzzle for the homepage. Loc8r now finds your current location and lists the places near you, which was the whole idea from the very start. The last thing we said we'd do in this chapter was sort out the About page, where we'll explore the challenges of injecting HTML through Angular bindings.

## 9.4 Safely bind HTML content

The current status of the About page in the Angular SPA, is that it only exists as a default skeleton page, as we created it to demonstrate navigation and routing in Angular. In this section we'll complete the page.

### 9.4.1 Adding the About page content to the app

The About page should be fairly straightforward. We'll just need to add the content to the component definition and create the simple markup with the bindings to display it. Easy right?

We'll start by adding the content to the component definition. In the following listing we can see the class definition in `about.component.ts`. We are defining a `pageContent` member to hold all of the text information like we've done before. I've trimmed down the text in the main content area to save ink and trees.

#### **Listing 9.21 Creating the Angular controller for the About page**

```
export class AboutComponent implements OnInit {
  constructor() { }

  ngOnInit() {
  }

  pageContent = {
    header : {
      title : 'About Loc8r',
      strapline : ''
    },
    content : 'Loc8r was created to help people find places to sit down and get a bit
              of work done.\n\nLorem ipsum dolor sit amet, consectetur adipiscing elit.'
  };
}
```

As components go this is pretty simple. No magic going on here, just note that we've still got the `\n` characters to denote line breaks.

Next up we need to create the HTML layout. From our original Pug templates we know what the markup needs to be - there just needs to be a page header and then a couple of divs to hold the content. For the page header we can re-use the `pageHeader` component that we created earlier, and pass the data through just like we did for the homepage. There's not much to the rest of the markup; the entire contents of `about.component.html` are shown in the following snippet.

```
<app-page-header [content]="pageContent.header"></app-page-header>
<div class="row">
  <div class="col-12 col-lg-8">{{ pageContent.content }}</div>
</div>
```

Again, nothing unusual here. Just the page header, some HTML and a standard Angular binding. If we take a look at this page in the browser we'll see that the content is coming through, but the line breaks aren't displaying, as illustrated in figure 9.13.

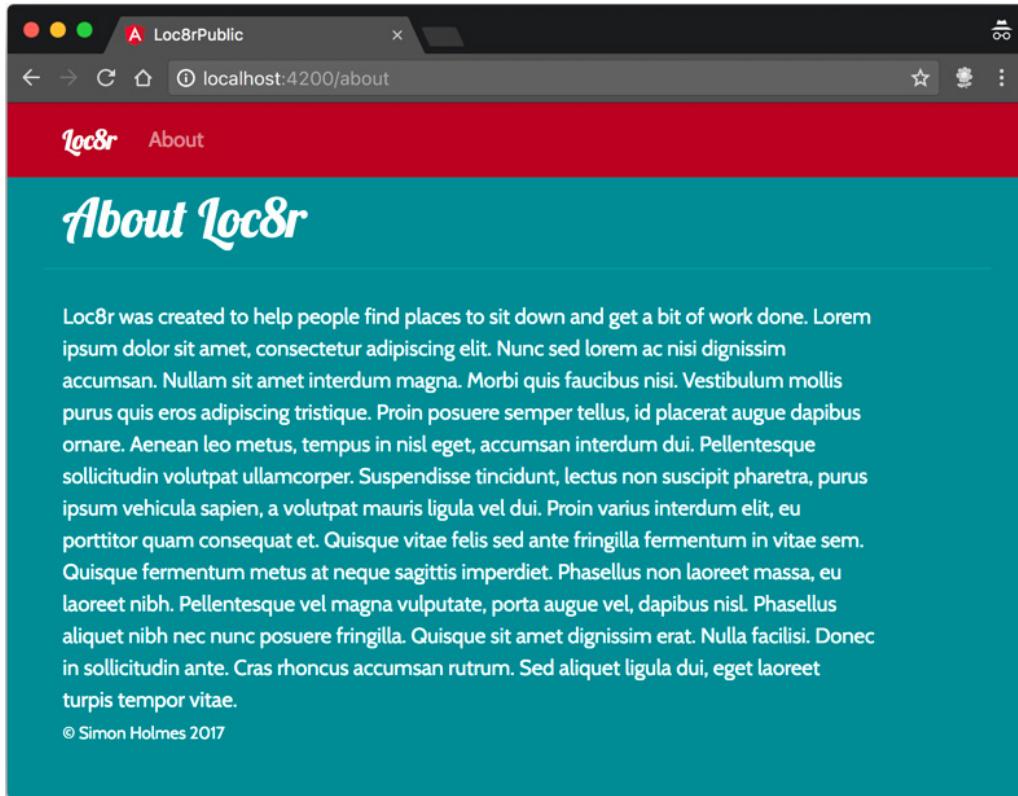


Figure 9.13 The content for the About page is coming through from the controller, but the line breaks are being ignored.

This isn't ideal. We want our text to be readable, and shown as originally intended. If we can change the way the distances appear on the homepage using a pipe, why not do the same thing to fix the line breaks? Let's give it a shot and create a new pipe.

#### 9.4.2 Create a pipe to transform the line breaks

So, we want to create a pipe that will take the provided text and replace each instance of `\n` with a `<br/>` tag. We've actually already solved this problem in Pug, using a JavaScript `replace` command as shown in the following code snippet:

```
p !{(content).replace(/\n/g, '<br>')}
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>

With Angular we can't do this inline, instead we need to create a pipe and apply it to the binding.

### **CREATE HTMLLINEBREAKS PIPE**

As we've already seen, pipes are best created by the Angular CLI, so run the following command in terminal to generate the files and register the pipe with the application.

```
$ ng generate pipe html-line-breaks
```

The pipe itself is fairly straightforward: it needs to accept incoming text as a string value, replace each \n with a <br/> and then return a string value. Update the main content of html-line-breaks.html to look like the following snippet.

```
export class HtmlLineBreaksPipe implements PipeTransform {

  transform(text: string): string {
    const output: string = text.replace(/\n/g, '<br/>');
    return output;
  }
}
```

When you've done that, let's try using it.

### **APPLY THE PIPE TO THE BINDING**

Applying a pipe to a binding is pretty simple—we've already done it a few times. In the HTML we just add the pipe character (|) after the data object being bound, and follow it with the name of the filter like this:

```
<div class="col-12 col-lg-8">{{ pageContent.content | htmlLineBreaks }}</div>
```

Simple, right? But if we try it in the browser all isn't quite as we'd hoped. As we can see in figure 9.14, the line breaks are being replaced with <br/> but they're being displayed as text instead of rendering as HTML.

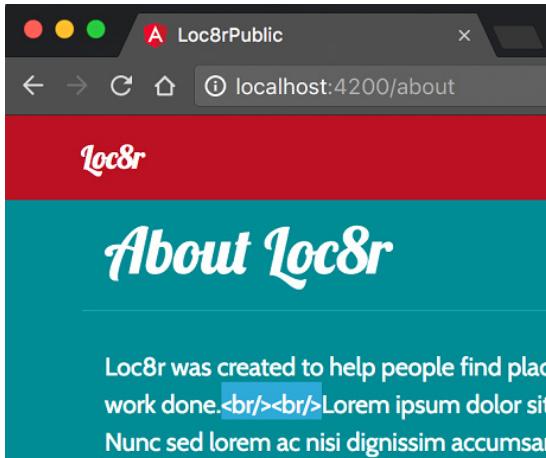


Figure 9.14 The `<br/>` tags being inserted with our filter are being rendered as text rather than HTML tags.

Hmmmm. Not quite what we wanted, but at least the pipe seems to be working! There's a very good reason for this output: security. Angular protects you and your application from malicious attacks by preventing HTML being injected into a data binding. Think about when we let visitors write reviews for locations, for example. If they could put any HTML in that they wanted, someone could easily insert a `<script>` tag and run some JavaScript hijacking the page.

But there's a way to let a subset of HTML tags through into a binding, which we'll look at now.

#### 9.4.3 Safely bind HTML using a property binding

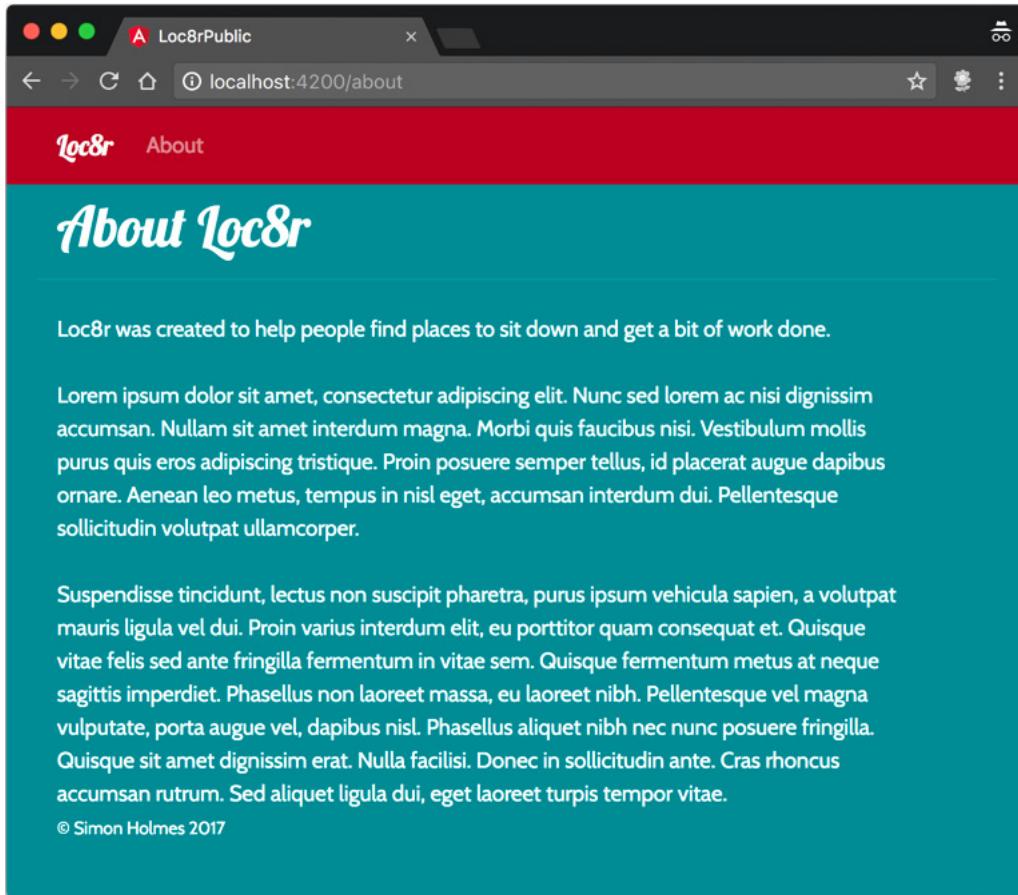
Angular will let us pass through some HTML tags if we use a *property binding*, instead of the default bindings we normally use for content. This only works for a sub-set of HTML tags, to prevent XSS hacks, attacks and weaknesses. Think of property binding as "one-way" binding. The component can't read the data back out and use it, but it can update it and change the data in the binding.

We have actually seen and used property bindings already, when passing data in nested components. There we were binding data to a property we defined in the nested component - we called it `content`. Here we are binding to a native property of a tag, in this case `innerHTML`.

Property bindings are denoted by wrapping square brackets around them, and then passing the value. So we can remove the content binding in `about.component.html` and use a property binding as shown in the following snippet.

```
<div class="col-12 col-lg-8" [innerHTML]="pageContent.content |  
htmlLineBreaks"></div>
```

Note that we are able to apply pipes to this type of binding it too, so we are still using our `htmlLineBreaks` pipe. Finally, this time when we view the About page in the browser we'll see the line breaks in place, looking like figure 9.15.



**Figure 9.15** Using the `htmlLineBreaks` pipe in conjunction with the property binding we now see the line breaks rendering as intended.

Success! Okay, there we go. A great start towards building Loc8r as an Angular SPA. We've got a couple of pages, some routing and navigation, geolocation and a great modular application architecture. Let's keep on moving!

## 9.5 Challenge

Take what you have learned about Angular so far, and create a new component called `rating-stars`. This will be used in the homepage listing section, and in the other places where we display rating stars - which we'll be building out in the next section.

This new component should:

- Accept an incoming number value (the rating)
- Display the correct number of solid stars based on the rating
- Be re-usable many times on a single page
- As a clue, your elements should look something like this:

```
<app-rating-stars [thisRating]="location.rating"></app-rating-stars>
```

Good luck! The code - should you need it - is available in GitHub, in the `chapter-09` branch.

## 9.6 Summary

In this chapter, we've covered the following:

- Setting up the Angular router to enable navigation between pages
- Defining where to show the routable content in the HTML
- How to build a functional and use site navigation
- Making good use of nested components to create a modular and scalable application architecture
- Using property bindings to pass data into nested components, and to safely bind some HTML
- How to work with external interfaces like the browser's geolocation capabilities

Coming up next in chapter 10 we'll continue with building out the Angular SPA, encountering more complex page layouts, modal popups and accepting user input via forms.

# 10

## *Building a Single Page Application with Angular: The next level*

### This chapter covers

- Routing with URL parameters in Angular
- Querying the API with URL parameter data
- Building more complex layouts
- Handling forms submissions in an SPA way
- Creating a separate router configuration file
- Replacing the Express UI with the Angular app

In this chapter we're following on from the work we started in chapter 9 with building a single-page application (SPA). By the end of this chapter the Loc8r application will be a single Angular application that uses our API to get the data.

Figure 10.1 shows where we're at in the overall plan, still recreating the main application as an Angular SPA.

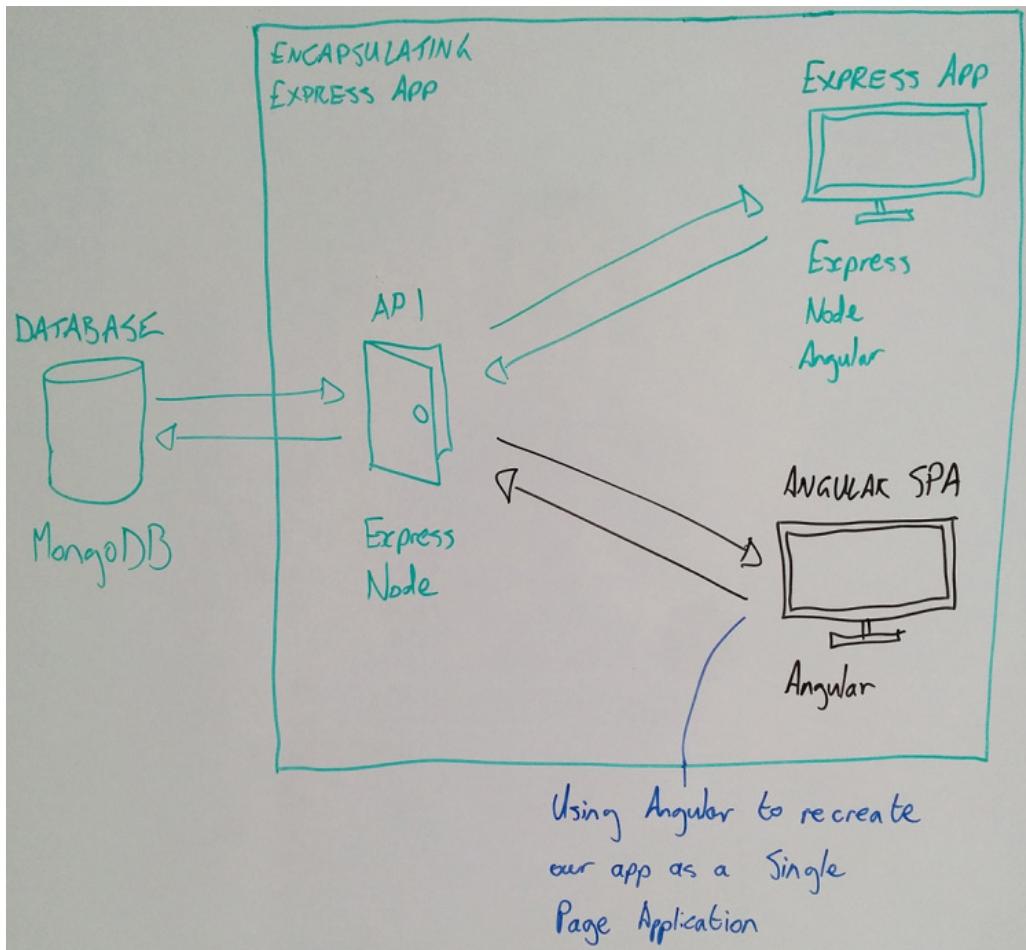


Figure 10.1 This chapter continues the work we started in chapter 9 of recreating the Loc8r application as an Angular SPA, moving the application logic from the back end to the front end.

We'll start off by creating the missing pages and functionality, and see how to use URL parameters in routes - including using them when querying the API. When we've got most of the functionality in place we'll add in the form to add new reviews, but rather than having a separate page like we had in Express, we'll include the form inline and add reviews without ever leaving the location details page. This is a very SPA-way of doing things and avoids extra roundtrips to the server. When everything's running we'll look at a couple of ways to improve the architecture to follow some Angular and TypeScript best practices.

To finish off we'll take our Angular application and use it as the front-end for Loc8r, removing the need for the public-facing part of the Express application.

## 10.1 More complex views and routing parameters

In this section we're going to add the Details page to the Angular SPA. One of the crucial aspects here will be retrieving the location ID from the URL parameter to ensure we get the correct data. Using URL parameters in this way is common practice, and is a very useful technique to know in any framework. We'll also have to update the data service to hit the API asking for specific location details. As we translate the Pug view into an Angular template we'll also discover some additional things that Angular does to help us lay out things.

We've got a lot to do, so before we get into the fun stuff we'd better plan it out.

### 10.1.1 Planning the layout

The location details page has quite a bit more to it than the others we've made in Angular so far, but as we know what it looks like we can start to plan it out from a high level. When that's done it'll be easier to add in the details.

By looking at the layout and what we've done already we can start to see the different components we'll need, and how to nest them. We'll keep the existing framework component on the outside of course, containing the navigation and footer. In the routable area we'll have a new *details page component* containing the page header, sidebar, and main content. Figure 10.2 shows a sketch of this layout plan overlaid on a screenshot of the details page itself.

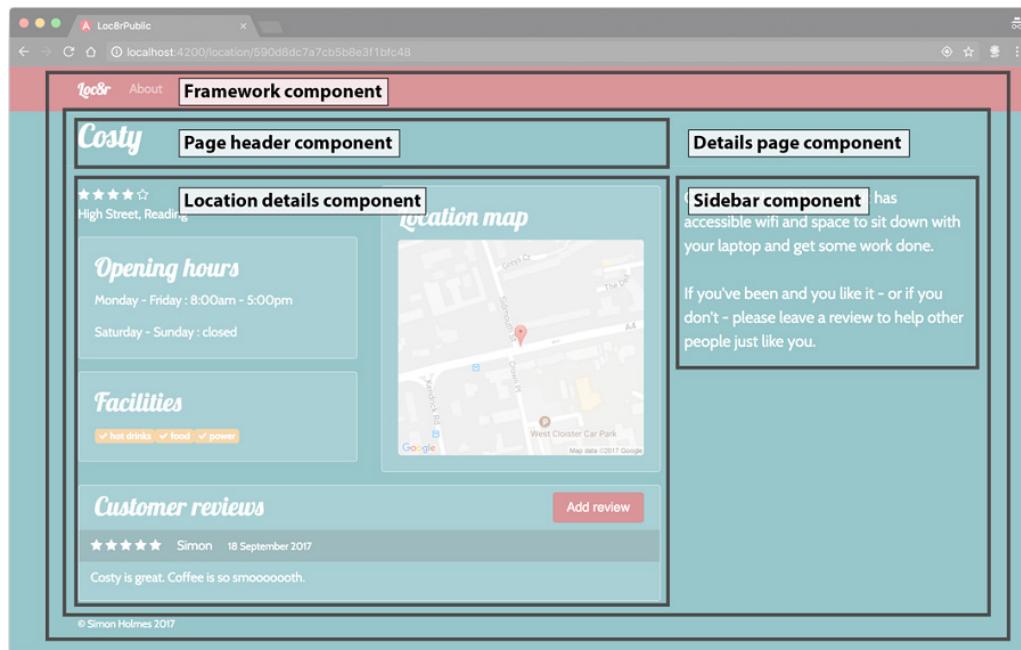


Figure 10.2 Planning the components and nesting needed for building the details page in Angular.

This plan gives us a good idea about what we need to build, and what we can reuse - hopefully you're starting to see why creating reusable components is a good idea! At this point, note that we need some of the location data in three of the components - the page header, content area, and sidebar. We'll need to take this into consideration when coding the page.

Of the five components in the plan, there are two we need to create - the page component to organize all of the others and the content area to display the actual details. We'll create basic versions of these now, so that we have a page to route to.

### 10.1.2 Creating the required components

We know we want a page component, containing the details along with the sidebar and header. This missing one is the details, so we'll create a skeleton of that first. We can then create the framework component ready for routing.

Use the Angular CLI to create the location details component, by running the following command in terminal.

```
$ ng generate component location-details
```

We can leave the default content in this new component for the time being, as we'll be building it out properly soon. Next we'll create the details page component, and add the skeleton layout to it.

In terminal use the Angular CLI again, with the following command:

```
$ ng generate component details-page
```

We're going to add some content to this component, as it's going to hold the other components for the page - the page header, location details and sidebar. Listing 10.1 shows how we want to lay these out in details-page.component.html. We'll also add in the `content` bindings for the page header and sidebar so that we can pass in information from this component.

#### **Listing 10.1 The basic layout for details-page.component.html**

```
<app-page-header [content]="pageContent.header"></app-page-header> ①
<div class="row">
  <div class="col-12 col-md-8">
    <app-location-details>Loading...</app-location-details> ②
  </div>
  <app-sidebar class="col-12 col-md-4" [content]="pageContent.sidebar"></app-sidebar>
</div> ③
```

- ① Page header component including a property binding
- ② Location details component
- ③ Sidebar component including a property binding

So that we'll be able to see the content in the header and sidebar, we'll create some default content. In the HTML for the details page component we use the bindings `pageContent.header` and `pageContent.sidebar`, so in the component class we'll create a corresponding `pageContent` member containing `header` and `sidebar` properties. Listing 10.2 shows what this looks like in `details-page.component.ts`, also giving the content properties some default text.

#### **Listing 10.2 The starting content for the details page in `details-page.component.ts`**

```
export class DetailsPageComponent implements OnInit {
  constructor() { }

  ngOnInit() {
    pageContent = {
      header : {
        title : 'Location name',
        strapline : ''
      },
      sidebar : 'is on Loc8r because it has accessible wifi and space to sit down with
                your laptop and get some work done.\n\nIf you\'ve been and you like it - or if
                you don\'t - please leave a review to help other people just like you.' ③
    };
  }
}
```

- ① The new `pageContent` member containing...
- ② ... `header` details and ...
- ③ ... `sidebar` content

Okay, now we have everything set up ready for our plan! We've got our details page component, containing the three nested components we need to layout the page. We've even got some starting data being passed into two of the nested components.

Now we're ready to set up the routing so that we can see the page.

#### **10.1.3 Setting up and defining routes with URL parameters**

Defining routes with URL parameters is as easy in Angular as it is in Express. Even the syntax is the same - not something you hear very often in the programming world!

Our routes for the app are defined in `app.module.ts`, so that's where we'll add the new one. As we want to accept a URL parameter we'll define the route in the same way we did in Express, by putting a `locationId` variable at the end of the path, preceded by a colon. The new route in situ is shown in the following listing.

#### **Listing 10.3 Adding the details page route to `app.module.ts`**

```
RouterModule.forRoot([
```

```
{
  path: '',
  component: HomepageComponent
},
{
  path: 'about',
  component: AboutComponent
},
{
  path: 'location/:locationId', ①
  component: DetailsPageComponent
}
])
```

- ① Define a 'locationId' URL parameter in the route by prefixing it with a colon

With that in place we can go to location/something in the browser, and Angular will route us to the details page component. At the moment, this looks like figure 10.3.

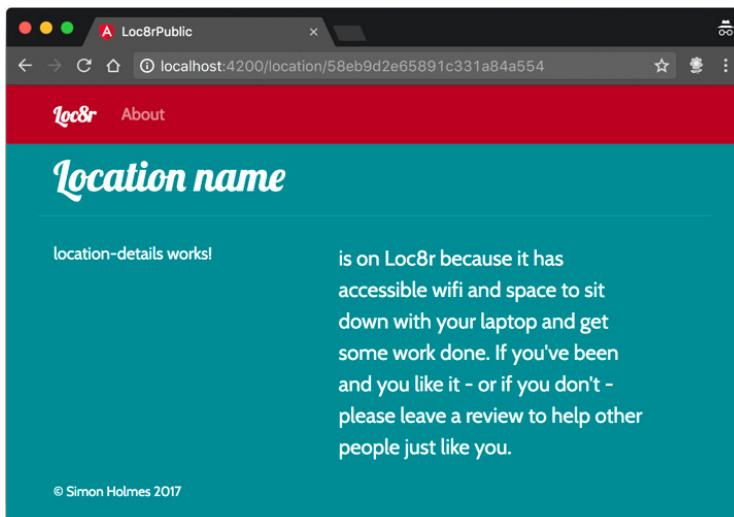


Figure 10.3 Testing the new location details route; seeing the default content we added to the components.

If we remember from our original layouts, the side bar content in this page should be in two paragraphs. This means our line breaks aren't coming through. Fortunately, we've already created a pipe for that, we just need to update the sidebar component to use. In sidebar.component.html change the Angular binding to an `innerHTML` property binding, passing in the content and the `htmlLineBreaks` pipe like this:

```
<p class="lead" [innerHTML]="content | htmlLineBreaks"></p>
```

Now the `\n` parts of the sidebar content are converted to `<br/>` tags and rendered as HTML, looking like figure 10.4.

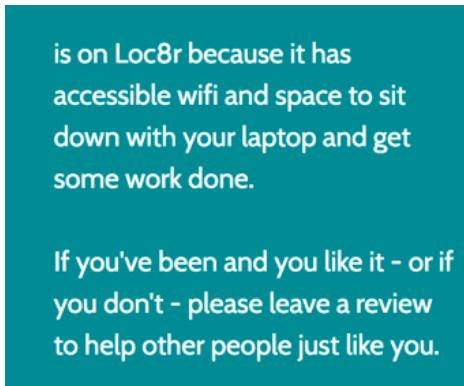


Figure 10.4 Enabling line breaks in the sidebar by making use of our custom pipe

The general page layout looks good, and we can see that working. Before we build it out, it would be useful to be able to navigate to this page with real location IDs in the URL. To do that we need to update the links in the homepage listings.

#### **CREATING ANGULAR LINKS TO THE DETAILS PAGE**

The homepage listing does currently display links to this page, and if you try them they will actually take you there. However, you may well notice that when you do, the page flickers. This is because the links are just standard `href` attributes in an `<a>` tag, so the browser just follows them like normal links. The result is the page is having a full reload, and reloading the Angular application back in - not what you want in an SPA!

We want Angular to capture clicks on these links, and handle the navigation and routing. When we created the navigation we used `routerLink` instead of `href` in the `<a>` tags, and we need to do the same here. In `home-list.component.html` find the link to the location and swap out the `href` attribute as shown here:

```
<a routerLink="/location/{{location._id}}">{{location.name}}</a>
```

The rest of the code can stay the same. With that simple change, we've made our app even more like a proper SPA. Now we're ready to start using the URL parameter in the page.

#### **10.1.4 Using URL parameters in components and services**

The plan here is to get the location ID URL parameter, and use that in a call to the API to get the details for a specific location. When the data comes back, we want to display it on the page.

Where's the best place to put this logic? Any of the components in the routable area could be configured to get the URL parameter and call the API, but we want to display data in all three nested components. So we'll go for the approach of using the "parent" details page component to get the data, and then pass it through into the three child components.

First up, we'll add a method to our data service that will call the API to get a single location by ID.

### **CREATING THE DATA SERVICE TO CALL THE API**

The data service that we created in chapter 8 currently has a single method `getLocations`. This retrieves a list of locations when given a pair of coordinates. The new method we need has a very similar construct, so make a copy of this method in `loc8r-data.service.ts` and call it `getLocationById`.

We just need to make a few small adjustments to get this working:

1. Change the expected input parameters to be a single `locationId` of type string
2. Change the return type to be a single `Location` instance, instead of an array
3. Change the API URL to call, using `locationId` as a URL parameter
4. Set the JSON response to be a single `Location` instance

Listing 10.4 shows how this looks in code, which should be in `loc8r-data.service.ts`.

#### **Listing 10.4 Adding a method to get a location by ID in `loc8r-data.service.ts`**

```
public getLocationById(locationId: string): Promise<Location> {
  const url: string = `${this.apiBaseUrl}locations/${locationId}`; ①
  return this.http
    .get(url)
    .toPromise() ②
    .then(response => response.json() as Location)
    .catch(this.handleError);
}
```

- ① Set the correct input parameters and expected return type, both single items
- ② Change the API URL to just use the location ID as a URL parameter
- ③ Set the JSON response to be a single Location instance

With the data service method ready, we can import the service into the details page component ready for us to use.

### **IMPORTING THE DATA SERVICE INTO THE COMPONENT**

We've imported a service into a component before - the data service into the home-list component - so we won't dwell on the process too much here. We'll need to import the data service into the details page component, add it to the providers and then make it available by declaring it in the class constructor.

While we're here, we'll also import the Location class from the home-list component and empty out the default page content. All of these updates to details-page.component can be seen in the following listing.

#### **Listing 10.5 Importing our data service in details-page.component.ts**

```
import { Component, OnInit } from '@angular/core';
import { Loc8rDataService } from './loc8r-data.service'; ①
import { Location } from '../home-list/home-list.component'; ②

@Component({
  selector: 'app-details-page',
  templateUrl: './details-page.component.html',
  styleUrls: ['./details-page.component.css'],
  providers: [Loc8rDataService] ③
})
export class DetailsPageComponent implements OnInit {

  constructor(
    private loc8rDataService: Loc8rDataService ④
  ) { }

  ngOnInit(): void { }

  pageContent = { ⑤
    header: {
      title: '',
      strapline: ''
    },
    sidebar: ''
  };
}
```

- ① Import our data service
- ② Import the Location class definition
- ③ Add the data service to the component providers
- ④ Create a private local instance of the data service
- ⑤ Clear out the default page content

The only real thing to be careful with here is the case of `loc8rDataService` in the constructor - the class type definition is with an uppercase 'L', and the local instance is defined with a lowercase 'l'. Now we're ready to get the URL parameter into the component.

#### **USING URL PARAMETERS IN A COMPONENT**

Given that using URL parameters in an app is a very common requirement, it is surprisingly complicated. There are three new pieces of functionality we need and a few steps to go through.

Here's what we'll need:

- `ActivatedRoute` from the Angular router to get us the value of the current route from inside the component.
- `ParamMap` from the Angular routers to get us the URL parameters of the active route as an observable.
- `switchMap` from RxJS to get the values from the `ParamMap` observable, and use them to call our API creating a second observable.

The following snippet shows in bold the additions needed in `details-page.component.ts` to import these piece of functionality.

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { Loc8rDataService } from '../loc8r-data.service';

import 'rxjs/add/operator/switchMap';
```

As well as this, we also need to make the activated route available to the component by defining a private member `route` of type `ActivatedRoute` in in the constructor.

```
constructor(
  private loc8rDataService: Loc8rDataService,
  private route: ActivatedRoute
) {}
```

Now the complicated bit. These are the steps we need to take to get a location ID from the URL parameter and turn it into location data from the API:

1. When the component initializes, use `switchMap` to subscribe to the `paramMap` observable of the activated route.
2. When the `paramMap` observable returns a `ParamMap` object, get the value of the `locationId` URL parameter.
3. Call the `getLocationsById` method of our data service, passing it the ID.
4. Return the API call, so that it returns an observable.
5. Subscribe to listen for when the observable returns the data from our API. The result of this should be a single object of type `Location`.
6. Set the content for the page header and sidebar using the location name returned from the API.

Phew! That's quite a lot of steps for a seemingly simple process. All of this takes place in the `ngOnInit` lifecycle hook in the `details-page.component.ts`. Listing 10.6 shows what this looks like in code.

#### **Listing 10.6 Getting and using the URL parameter in `details-page.component.ts`**

```
ngOnInit(): void {
  this.route.paramMap
    .switchMap((params: ParamMap) => {
      let id = params.get('locationId');
      return this.loc8rDataService.getLocationById(id);
```

1  
2  
3  
4

```

    })
    .subscribe((newLocation: Location) => {
      this.pageContent.header.title = newLocation.name;
      this.pageContent.sidebar = `${newLocation.name} is on Loc8r because it has
        accessible wifi and space to sit down with your laptop and get some work
        done.\n\nIf you've been and you like it - or if you don't - please leave a
        review to help other people just like you.`;
    });
}

```

- ➊ Get the paramMap observable of the activated route
- ➋ Use switchMap to subscribe to this, and return a paramMap
- ➌ Use the `get` method to get the value of the locationId URL parameter from the paramMap
- ➍ Make the call to our new data service method, returning it as an observable
- ➎ Subscribe to the API call observable, expecting a Location back
- ➏ Send the location name to the page header and sidebar

That's some pretty dense code; a lot is happening in very few lines and commands. I recommend reading through the plan and the annotated code a few times to piece it all together. It's very powerful, a little different from what we've seen so far, and about as complex as we'll see in this book. In particular note the two chained observables: first the route `paramMap` being subscribed to by the `switchMap`, which then returns the second.

The good news is that when this is done, a location detail page will now show the location name in the page header and the sidebar as shown in Figure 10.5.

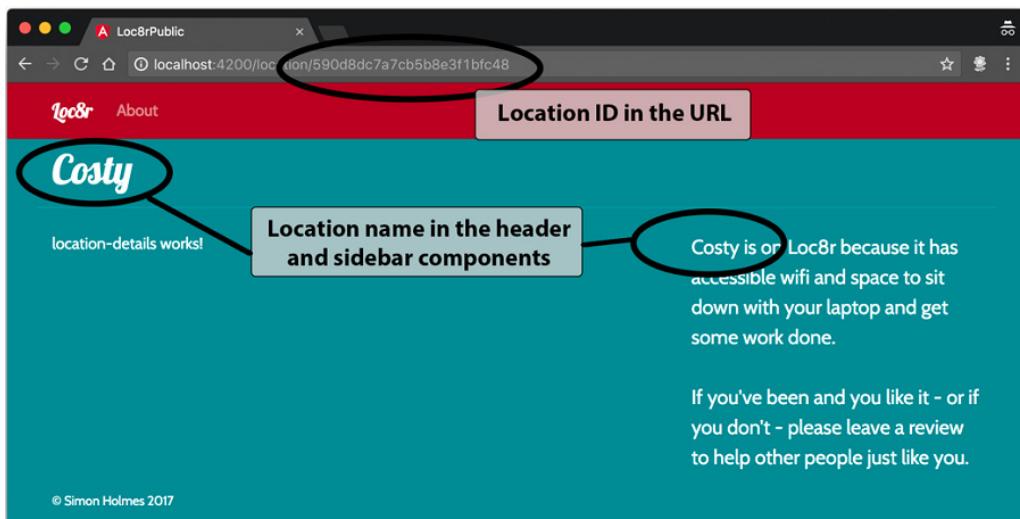


Figure 10.5 Displaying the location name in the header and sidebar after getting the location ID from the URL and sending it to the API

We are now using the location ID in the URL to query the database, and passing a bit of the returned data to two of the components on the page. Before we build out the main part of the location details page, let's make sure that final component is getting the data it needs.

### PASSING DATA TO THE DETAILS COMPONENT

To pass the location data from the details page component into the nested location details component we will need to do three things:

1. Add a class member to the details page component to hold the location data when we get it back from the data service.
2. Pass the data into the child component using a property binding in the HTML.
3. Update the location details component to accept this incoming data.

So first of all, as shown in listing 10.7, define a new member `newLocation` of type `Location` in `details-page.component.ts`, and give it a value when we get a location back from the API call.

#### **Listing 10.7 Exposing the found location details in details-page.component.ts**

```
newLocation: Location;

ngOnInit(): void {
  this.route.paramMap
    .switchMap((params: ParamMap) => {
      let id = params.get('locationId');
      return this.loc8rDataService.getLocationById(id);
    })
    .subscribe((newLocation: Location) => {
      this.newLocation = newLocation;
      this.pageContent.header.title = newLocation.name;
      this.pageContent.sidebar = `${newLocation.name} is on Loc8r because it has
        accessible wifi and space to sit down with your laptop and get some work
        done.\n\nIf you've been and you like it - or if you don't - please leave a
        review to help other people just like you.`;
    });
}
```

With the location details being exposed through this new `newLocation` class member, we can pass this through to the nested component by adding a property binding to the element in `details-page.component.html` like this:

```
<app-location-details [location]="newLocation">Loading...</app-location-details>
```

We've seen these before. This will pass the contents of `newLocation` in the details page component into the `location` member of the location details component.

Our location details component doesn't have a `location` member yet, so we'll need to add it to the component definition and set it up to be an input member of type `Location`. We've done these actions before, so listing 10.8 serves as a handy reminder, showing all of this in place in `location-details.component.ts`.

**Listing 10.8 Accept incoming location data in location-details.component.ts**

```

import { Component, Input, OnInit } from '@angular/core';           ①
import { Location } from '../home-list/home-list.component';        ②
@Component({
  selector: 'app-location-details',
  templateUrl: './location-details.component.html',
  styleUrls: ['./location-details.component.css']
})
export class LocationDetailsComponent implements OnInit {

  @Input() location: Location;                                     ③

  constructor() { }

  ngOnInit() {
  }

}

```

- ① Import 'Input' from the Angular core
- ② Import our 'Location' class definition
- ③ Define 'location' as an input member of type 'Location'

The page is still working and looks like it did before, but now the page component is getting the data from the database and passing it into all three of the nested components. It's time to build out the nested view.

### **10.1.5 Building the Details page view**

For the location details we've already got a Pug template with Pug data bindings, and we need to transform this into HTML with Angular bindings. There are quite a few bindings to put in place and some loops using `*ngFor`. We'll also use the `rating-stars` component again to show the overall rating and the rating for each review. And we'll need to allow line breaks in the review text by using the `htmlLineBreaks` pipe.

#### **GETTING THE MAIN TEMPLATE IN PLACE**

Listing 10.9 shows everything in place with the bindings in bold. This code should make up the entire contents of `location-details.component.html`. There are some pieces we have left out, such as the opening times, which we'll fill in when we've got this in place and tested.

**Listing 10.9 Angular template for the location details in location-details.component.html**

```

<div class="row">
  <div class="col-12 col-md-6">
    <app-rating-stars [thisRating]="location.rating"></app-rating-stars>          ①
    <p>{{ location.address }}</p>
    <div class="card card-primary">
      <div class="card-block">

```

```

<h2 class="card-title">Opening hours</h2>
<!-- Opening times to go here -->
</div>
</div>
<div class="card card-primary">
  <div class="card-block">
    <h2 class="card-title">Facilities</h2>
    <span *ngFor="let facility of location.facilities" class="badge badge-warning">
      <i class="fa fa-check"></i>
      {{facility}}
    </span>
  </div>
</div>
</div>
<div class="col-12 col-md-6 location-map">
  <div class="card card-primary">
    <div class="card-block">
      <h2 class="card-title">Location map</h2>
      
    </div>
  </div>
</div>
</div>
<div class="row">
  <div class="col-12">
    <div class="card card-primary review-card">
      <div class="card-block"><a href="/location/{{location._id}}/review/new" class="btn btn-primary float-right">Add review</a>
        <h2 class="card-title">Customer reviews</h2>
        <div *ngFor="let review of location.reviews" class="row review">
          <div class="col-12 no-gutters review-header">
            <app-rating-stars [thisRating]="review.rating"></app-rating-stars> ③
            <span class="reviewAuthor">{{review.author}}</span>
            <small class="reviewTimestamp">{{review.createdOn}}</small>
          </div>
          <div class="col-12">
            <p [innerHTML]="review.reviewText | htmlLineBreaks"></p> ⑤
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
</div>

```

- ① Use rating-stars component to show average rating for location
- ② Loop through facilities
- ③ Loop through reviews
- ④ Using rating-stars component to show rating for each review
- ⑤ Apply htmlLineBreaks pipe to review text and bind as HTML

Now that's quite a long code listing! But that's to be expected, as there's quite a lot going on in the Details page. If you look at the page in the browser now it looks about right - there are a few things to fix but we know about those.

Although the page looks good, if you open the JavaScript console you'll see that the page has thrown a lot of errors along the lines of *Cannot read property 'rating' of undefined*. These are binding errors, happening because the nested details component is trying to bind to data as soon as the page loads, but we don't have any data until after the API call has completed.

### **HIDING COMPONENTS TO AVOID PREMATURE BINDING ERRORS**

The binding errors are occurring because the component is trying to bind to data, before the data has been provided. So how do we stop this from happening? A good way is to hide the component in the HTML until the data has been received from the API, and we have the location details ready to display.

Angular includes a very helpful native directive called `*ngIf` which can be added to an element in the HTML. `*ngIf` is given an expression; if the expression resolves to `true` then the element is shown, otherwise it is hidden.

For our situation, we only want to show the location-details component when the location data exists. So we can add an `*ngIf` directive to the `location-details` element in `details-page.component.html` like so:

```
<div class="col-12 col-md-8">
  <app-location-details *ngIf="newLocation" [location]="newLocation">Loading...</app-
    location-details>
</div>
```

And with that small change, no more binding errors! Now onto fixing the remaining page template issues, as we're not showing opening times yet, the reviews are coming through oldest first, and data of the reviews needs formatting.

### **ADDING IF-ELSE STYLE LOGIC WITH NGSWITCHCASE TO SHOW THE OPENING TIMES**

It's not unusual to want some type of `if-else` logic in a template, to let you show different chunks of HTML depending on a certain parameter. For each opening time we want to display the days in the range, and either a closed message or the opening and closing times. In our Pug template we had a bit of logic, a simple `if` statement checking whether `closed` was `true`, as shown in the following code snippet:

```
if time.closed
| closed
else
| #{time.opening} - #{time.closing}
```

We want to do something similar in our Angular template. We've just seen how `*ngIf` can work for a one-off case, but for `if-else` logic Angular works along the line of JavaScript's

`switch` method. With this you define which condition you want to check at the top, and then provide different options depending on the value of the condition.

The key parts here are an `[ngSwitch]` binding for defining the condition to switch on, an `*ngSwitchCase` directive for providing a specific value, and an `*ngSwitchDefault` directive for providing a backup option if none of the specific values matched. We can see all of these in action in the following listing where we add the opening times to `location-details.component.html`.

#### **Listing 10.10 Using ngSwitch to display opening times in location-details.component.html**

```
<div class="card card-primary">
  <div class="card-block">
    <h2 class="card-title">Opening hours</h2>
    <p class="card-text" *ngFor="let time of location.openingTimes"
      [ngSwitch]="time.closed">
      {{ time.days }} :
      <span *ngSwitchCase="true">closed</span>
      <span *ngSwitchDefault>{{ time.opening + " - " + time.closing }}</span>
    </p>
  </div>
</div>
```

- ➊ Run switch based on value of `time.closed`
- ➋ When `time.closed` is true just output closed
- ➌ Otherwise default action is to output opening and closing times

And with that we now have a bit of logic in the template. Note that as all of the `ngSwitch` commands are property bindings and directives they need to be added to HTML tags. Okay, let's get the reviews showing most recent first.

#### **CHANGING THE DISPLAY ORDER OF A LIST USING A CUSTOM PIPE**

If you have experience with AngularJS you may be expecting an update of the old `orderBy` filter, which could be used to magically re-order a repeated list in almost any way imaginable. It was very flexible and powerful, but this came with a downside. With any large sets of data this flexible filter became very slow. For this reason, the Angular team decided not to include it in the new versions.

Without a native way to change the order of a list, the options are to write some code in the component, or to create a new pipe. A pipe is often best - especially if you think you might want to re-use the functionality somewhere else - and we also know that a pipe will always be applied if the data changes.

So we'll create a new pipe specifically to order the reviews by date - the most recent first. We'll create the new pipe the normal way, by running the following command in terminal, in the `app_public` folder:

```
$ ng generate pipe most-recent-first
```

When the pipe is generated, add it to the `*ngFor` directive looping the reviews in `location-details.component.html`, like so:

```
<div *ngFor="let review of location.reviews | mostRecentFirst" class="row review">
```

And now to code up the pipe itself. Remember that it comes with a `transform` hook that accepts a value and returns a value. For our purposes we want to accept and return an array, as reviews are returned from the database as an array

As we're working with arrays, we can use JavaScript's native array `sort` method, which accepts a function as a parameter. This function takes two items from the array at a time and can do whatever you code to compare them. The return value of the function should be either a positive or negative number, where a negative means the order stays the same, positive means the order changes.

We're comparing dates, and want the most recent first. In terms of comparison operators, a more recent date is "greater than" an older date. So, if the date of the first parameter is greater than (more recent than) the date of the second, then we return a negative number to keep the order the same. Otherwise return a positive number to swap them round.

That's more complicated to explain that it is to code! Listing 10.11 shows what the pipe code looks like, creating a comparison function called `compare`, and using it to sort the array of reviews before returning the updated array.

#### **Listing 10.11 Creating most-recent-first.pipe.ts to change display order of reviews**

```
export class MostRecentFirstPipe implements PipeTransform {
    private compare(a, b) {
        const createdOnA = a.createdOn;
        const createdOnB = b.createdOn;
        let comparison = 1;
        if (createdOnA > createdOnB) {
            comparison = -1;
        }
        return comparison;
    }

    transform(reviews: [any]): [any] {
        if (reviews && reviews.length > 0) {
            return reviews.sort(this.compare);
        }
        return null;
    }
}
```

- ➊ Our comparing function, taking two values from the array
- ➋ Get the creation date of each review
- ➌ If a is more recent than b, return -1, otherwise return 1
- ➍ The transform method, accepting and returning arrays of reviews
- ➎ Use our compare function to sort the array, returning the re-ordered version

And now if you reload the page you should see your reviews showing in the correct order with most recent first. It's a little hard to tell though, as the date format is not exactly user friendly. Let's fix it.

### ***FIXING THE DATE FORMAT USING THE DATE PIPE***

Fortunately formatting dates is much more simple than ordering by them! One of Angular's default pipes is the `date` pipe, which will format a given date in the style that you want. This takes just one argument: the format for your date.

To apply your formatting, you send a string describing the output you want. There are too many different options to go into here, but the format is quite easy to get the hang of. To get the format "1 September 2017" we'll send the string '`'d MMMM yyyy'`' as shown in the following listing.

#### ***Listing 10.12 Applying a data pipe to format review dates in location-details.component.html***

```
<div *ngFor="let review of location.reviews" class="row review">
  <div class="col-12 no-gutters review-header">
    <app-rating-stars [thisRating]="review.rating"></app-rating-stars>
    <span class="reviewAuthor">{{ review.author }}</span>
    <small class="reviewTimestamp">{{ review.createdOn | date : 'd MMMM yyyy' }}</small>
  </div>
  <div class="col-12">
    <p [innerHTML]="review.reviewText | htmlLineBreaks"></p>
  </div>
</div>
```

And with that we're done with the layout and formatting of the Details page, which should now look like figure 10.6.

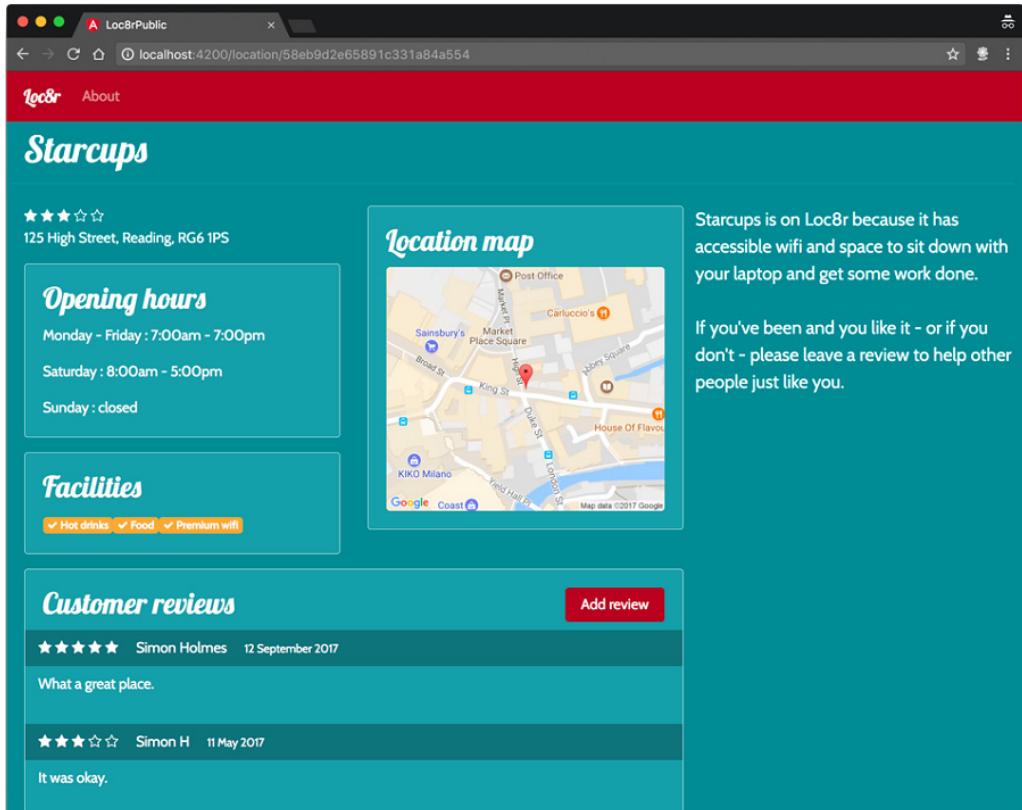


Figure 10.6 All of the location details are now being shown in the Angular page

The next and final step is to allow reviews to be added, but we're going to drop the concept of an extra page to do this, which is how we did it in Express. Instead we're going to do it inline on the page, to provide a slicker experience.

## 10.2 Working with forms and handling submitted data

In this section we are going to create the “add review” form in Angular, and have it submit data to the API. Rather than navigate to a separate form page when clicking the add review button, we'll get it to display a form inline in the page. When the form is submitted we'll get Angular to handle the data, submit it to the API and display the new review at the top of the list.

We'll start by seeing what's involved with creating the form in Angular.

### 10.2.1 Creating the review form in Angular

To create the review form, we'll get the HTML in place, add data bindings to the input fields, make sure they all work as expected, and finally ensure that the form starts off hidden and is only displayed if the button is clicked.

#### **PUT THE FORM HTML IN PLACE**

For the inline form, we'll add it into the page just after the "Customer reviews" `<h2>` tag as shown in listing 10.13. Much of the layout is taken from the form we used in Express, including the form input names and IDs.

#### **Listing 10.13 Add the review form to location-details.component.html**

```

<h2 class="card-title">Customer reviews</h2>
<div>
  <form action="">
    <hr>
    <h4>Add your review</h4>
    <div class="form-group row">
      <label for="name" class="col-sm-2 col-form-label">Name</label>
      <div class="col-sm-10">
        <input id="name" name="name" required="required" class="form-control">
      </div>
    </div>
    <div class="form-group row">
      <label for="rating" class="col-sm-2 col-form-label">Rating</label>
      <div class="col-sm-10">
        <select id="rating" name="rating">
          <option value="5">5</option>
          <option value="4">4</option>
          <option value="3">3</option>
          <option value="2">2</option>
          <option value="1">1</option>
        </select>
      </div>
    </div>
    <div class="form-group row">
      <label for="review" class="col-sm-2 col-form-label">Review</label>
      <div class="col-sm-10">
        <textarea name="review" id="review" rows="5" class="form-control"></textarea>
      </div>
    </div>
    <div class="form-group row">
      <div class="col-12">
        <button type="submit" class="btn btn-primary float-right" style="margin-left:15px">Submit review</button>
        <button type="button" class="btn btn-default float-right">Cancel</button>
      </div>
    </div>
    <hr>
  </form>
</div>

```

- Add the new div and the form HTML directly after the “Customer reviews” header

Right now we’re not doing anything clever or asking Angular to do anything. We’ve just put raw HTML with some Bootstrap classes into the template. In the browser this looks like figure 10.7.

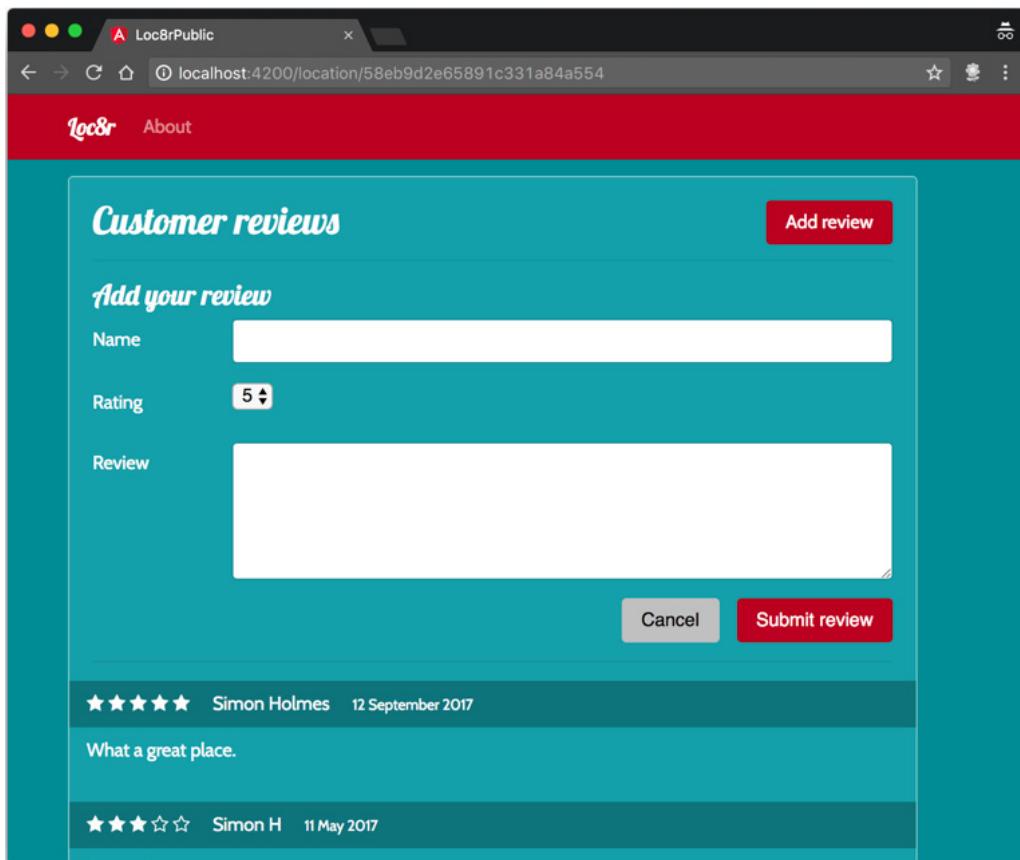


Figure 10.7 The review form displays inline in the details page, between the “Add review” button and the list of reviews.

That’s the basic form in place; now for the data bindings.

## ADD DATA BINDINGS TO FORM INPUTS

In Express we posted the form through to another URL and handled the submitted data there, but with Angular we don't want to change the page at all. With Angular the approach is to add data bindings to all of the fields in a form, so that the component can access the values.

To add a data binding to a form field use a directive with a special syntax like this: `[ (ngModel) ]="bindingName"`. Remembering the order of the brackets can be difficult, so this has become known as "banana in a boat" to help us all remember!

As a side note, to use `ngModel` in your HTML you need to have `FormsModule` imported into the application in `app.module.ts`. The generator adds it by default, so you should see it there already - look for `import { FormsModule } from '@angular/forms';`.

In our component we'll want to keep all of the submitted form data inside a single object so that we can pass it around easily. Define a new public member `newReview` in `location-details.component.html`, giving it properties for the author name, rating, and review content. Each property will need to have a default value, so the definition should look like this:

```
public newReview = {
  author: '',
  rating: 5,
  reviewText: ''
};
```

Now that this `newReview` object and its properties are defined in the component we can use them in the HTML. The following listing shows how to add the bindings to the form in `location-details.component.html`.

### **Listing 10.14 Add data bindings to the review form in location-details.component.html**

```
<form action="">
  <hr>
  <h4>Add your review</h4>
  <div class="form-group row">
    <label for="name" class="col-sm-2 col-form-label">Name</label>
    <div class="col-sm-10">
      <input [(ngModel)]="newReview.name" id="name" name="name" required="required"
        class="form-control" ①
    </div>
  </div>
  <div class="form-group row">
    <label for="rating" class="col-sm-2 col-form-label">Rating</label>
    <div class="col-sm-10">
      <select [(ngModel)]="newReview.rating" id="rating" name="rating" ①
        <option value="5">5</option>
        <option value="4">4</option>
        <option value="3">3</option>
        <option value="2">2</option>
        <option value="1">1</option>
      </select>
    </div>
  </div>
  <div class="form-group row">
```

```

<label for="reviewText" class="col-sm-2 col-form-label">Review</label>
<div class="col-sm-10">
  <textarea [(ngModel)]="newReview.reviewText" name="reviewText" id="reviewText" rows="5" class="form-control"></textarea> ①
  </div>
</div>
<div class="form-group row">
  <div class="col-12">
    <button type="submit" class="btn btn-primary float-right" style="margin-left:15px">Submit review</button>
    <button type="button" class="btn btn-default float-right">Cancel</button>
  </div>
</div>
<hr>
</form>

```

- ① Add the “banana in a boat” model bindings to the form inputs

This all looks good, and on the face of it seems to work. However, we want the rating to be of type number, but in a select option `value="5"` is actually a string containing the character `5`.

### **WORKING WITH SELECT VALUES THAT ARE NOT STRINGS**

So, a select option `value` is by default a string, but our database requires a number for the rating. Angular has a way to help us get different types of data from a select field.

Instead of using `value="STRING VALUE"` inside each `<option>`, use `[ngValue]="ANGULAR EXPRESSION"`. When written out, the value of `[ngValue]` looks like a string, but it's actually an Angular expression. This could be an object or a true boolean, but we want a number.

In `location-details.component.html` update each of the `<option>` tags to use `[ngValue]` instead of `value`.

```

<option [ngValue]="5">5</option>
<option [ngValue]="4">4</option>
<option [ngValue]="3">3</option>
<option [ngValue]="2">2</option>
<option [ngValue]="1">1</option>

```

Now Angular gets the value of the `<select>` as a number, not a string. This will be very useful when we come to submit the data to the API. Next though, let's hide the form by default, as we don't want it showing all of the time.

### **SETTING THE VISIBILITY OF THE FORM**

We don't want the add review on show when the page loads; we want the “Add review” button to show it, and when the form is displayed we want the “Cancel” button to hide it again.

To show and hide the form we can use our old friend `*ngIf`. `*ngIf` needs a Boolean value to decide whether or not to show the element it is applied to, so we'll need to define one in the component.

In `location-details.component.ts`, define a new public member `formVisible` of type `boolean` with a default value of `false`, like so:

```
public formVisible: boolean = false;
```

We've set the default value to be `false`, as we want the form to be hidden by default. To use this Boolean to set the visibility of the form, locate the `<div>` surrounding the `<form>` in `location-details.component.html` and add the `*ngIf` directive to it like this:

```
<h2 class="card-title">Customer reviews</h2>
<div *ngIf="formVisible">
  <form action="">
```

The form is now hidden by default when the page loads.

### **TOGGLING THE VISIBILITY OF THE FORM**

To change the visibility of the form, we need a way to change the value of `formVisible` when the "Add review" and "Cancel" buttons are clicked. Not surprisingly Angular as an on-click handler we can use to track clicks on elements, and then do something.

Angular's click handler is accessed by adding `(click)` to the element, and giving it an Angular expression. This could be calling a public member in the component class or any other kind of valid expression. We simply want to set `formVisible` to `true` when the "Add review" button is clicked, and `false` when the "Cancel" button is clicked.

In `location-details.component.html` change the "Add review" button from an `<a>` tag to a `<button>`, removing the `href` attribute and replacing it with a `(click)` setting `formVisible` to be `true` like this:

```
<button (click)="formVisible=true" class="btn btn-primary float-right">Add
review</button>
```

In a similar way, add a `(click)` to the "Cancel" button, setting `formVisible` to be `false`.

```
<button type="button" (click)="formVisible=false" class="btn btn-default float-
right">Cancel</button>
```

With those click handlers in place, we can use the two buttons to show and hide the review form, using the `formVisible` property of the component to keep track of the status. The final thing we need to do is hook up the form, so that when it's submitted a review is added.

### **10.2.2 Sending submitted form data to an API**

Now is the time to make our review form work and actually add a review to the database when it's submitted. To get to this end point we have a few steps involved:

1. Add a new member to our data service to POST new reviews to the API.
2. Have Angular handle the form when it's submitted.
3. Validate the form so that only complete data is accepted.

4. Send the review data to our service.
5. Push the review into list in Details page.

We'll start with step 1: Add a new member to our data service to POST new reviews to the API.

### **UPDATE THE DATA SERVICE TO ACCEPT NEW REVIEWS**

In order to use the form to post review data we need to add a method to our data service that talks to the correct API endpoint and can post the data. We'll call this new method `addReviewByLocationId` and have it accept two parameters: a location ID and the review data.

The contents of the method will be just the same as the others, except we'll be using `post` instead of `get` to call the API. The following listing shows the new method to be added to `loc8r-data.service.ts`.

#### **Listing 10.15 New public member for adding reviews in loc8r-data.service.ts**

```
public addReviewByLocationId(locationId: string, formData: any): Promise<any> {
  const url: string = `${this.apiBaseUrl}locations/${locationId}/reviews`;
  return this.http
    .post(url, formData)
    .toPromise()
    .then(response => response.json() as any)
    .catch(this.handleError);
}
```

Brilliant, now we'll be able to use this method from our component when we've got the form handling sorted. So, onto step 2: Have Angular handle the form when it's submitted.

### **ADDING THE ONSUBMIT FORM HANDLER**

When working with a form in HTML you'd typically have an action and a method to tell the browser where to send the data and the HTTP request method to use. You might also have an `onSubmit` event handler if you wanted to do anything with the form data using JavaScript before it was sent.

In an Angular SPA we don't want the form to submit to a different URL taking us to a new page. We want Angular to handle everything. For this we can use Angular's `ng-submit` event handler to call a public member in the component. The following code snippet shows how this is used, adding it into the form definition, calling a function in the component that we'll write in just a moment:

```
<form (ngSubmit)="onReviewSubmit()">
```

This will call a public method on the component called `onReviewSubmit` when the form is submitted. We need to create this method, so we'll add a simple method into `location-details.component.ts` to create a console log when the form is submitted.

```
public onReviewSubmit():void {
  console.log(this.newReview);
}
```

Because we bound all of the form fields to properties of `newReview`, this console log will output all of the data submitted. Now that we can capture the form data we'll add in some validation in step 3 so that only complete data is accepted.

### **VALIDATING THE SUBMITTED FORM DATA**

Before we blindly send every form submission to the API to save to the database, we want to do some quick validation to ensure that all of the fields are filled in. If any of them aren't filled in we'll display an error message. Your browser may prevent forms from being submitted with empty required fields; if this is the case for you, temporarily remove the `required` attribute from the form fields to test the Angular validation.

When a form is submitted, we'll start off by removing any existing error messages before checking whether each data item in the form is truthy (i.e. any form of `true` value). If any of the checks return `false`—that is, a field has no data—we'll set a form error message in the component. If all of the data exists we'll continue to log it to the console as before.

The following listing shows the new validation member we need to add to `location-details.component.ts`, and how we need to change the `onReviewSubmit` function to use it.

#### **Listing 10.16 Add validation to the review form in location-details.component.ts**

```
private formIsValid(): boolean {
  if (this.newReview.author && this.newReview.rating && this.newReview.reviewText) { ①
    return true;
  } else {
    return false;
  }
}

public onReviewSubmit():void {
  this.formError = '';
  if (this.formIsValid()) { ②
    console.log(this.newReview);
  } else { ③
    this.formError = 'All fields required, please try again'; ④
  }
}
```

- ① Private member to check that all form fields have content
- ② Reset any existing error messages
- ③ If form validation passes log submitted data to console
- ④ Otherwise set an error message

Now that we're creating an error message we want to show it to users when it's generated. For this we'll add in a new Bootstrap alert `div` into the form template, and bind the message as the content. We only want to show the `div` when there's an error message to display, so

we'll use the familiar `*ngIf` directive to handle this, checking to see whether `formError` has a value or not.

The following code snippet shows the addition we need to make to the review form template, adding the alert right near the top of the form:

```
<h4>Add your review</h4>
<div *ngIf="formError" class="alert alert-danger" role="alert">
  {{ formError }}
</div>
<div class="form-group row">
```

Now if you click the submit button without adding details to the form you'll get an alert, looking like figure 10.8.

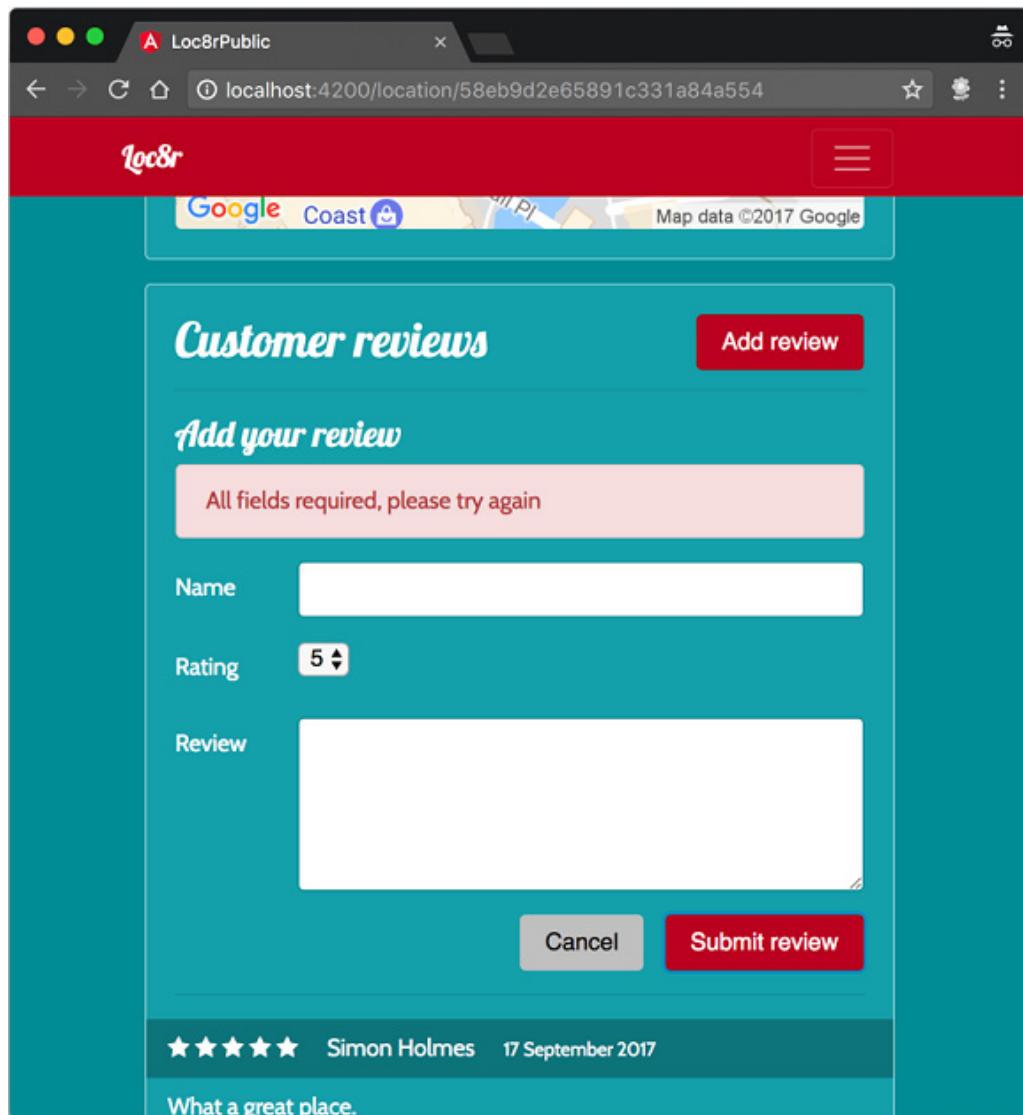


Figure 10.8 When trying to submit an incomplete form an error message is displayed

So we've got invalid covered, now to deal with valid data and send it to the API in step 4: Send the review data to our service.

## SENDING THE FORM DATA TO THE DATA SERVICE

We've got our form data being posted and we've got a data service ready to post it to the API. Let's hook these two up. We'll use the data service just like we've done before; using this new method is no different.

But first we need to import the data service into location-details.component.ts, and add it to the decorator as shown in the following listing.

### **Listing 10.17 Importing and providing the data service to location-details.component.ts**

```
import { Component, Input, OnInit } from '@angular/core';
import { Location } from '../home-list/home-list.component';
import { Loc8rDataService } from '../loc8r-data.service';

@Component({
  selector: 'app-location-details',
  templateUrl: './location-details.component.html',
  styleUrls: ['./location-details.component.css'],
  providers: [Loc8rDataService]
})
```

In the same file, we also need to add the service to the constructor so that we can use it:

```
constructor(private loc8rDataService: Loc8rDataService) { }
```

With the service now available in the component, we are able to call our new `addReviewByLocationId` method. The method expects the location ID and review details and resolves a promise, which will return the review record as saved in the database. This is all shown in listing 10.18. To validate that it's working, we'll also add a console log outputting the returned review.

### **Listing 10.18 Sending new reviews to the service in location-details.component.ts**

```
public onReviewSubmit():void {
  this.formError = '';
  if (this.formIsValid()) {
    console.log(this.newReview);
    this.loc8rDataService.addReviewByLocationId(this.location._id, this.newReview)
      .then(review => {
        console.log('Review saved', review);
      })
    } else {
      this.formError = 'All fields required, please try again';
    }
}
```

①  
②  
③

- ① Call the data service method, passing the location ID and new review data
- ② The method resolves a promise returning the saved review
- ③ Log the saved review data

And now we can send reviews to the database, and see the console logs as demonstrated in figure 10.9 - note the `createdOn` and `_id` in the console log that are generated by Mongoose when the record is saved.

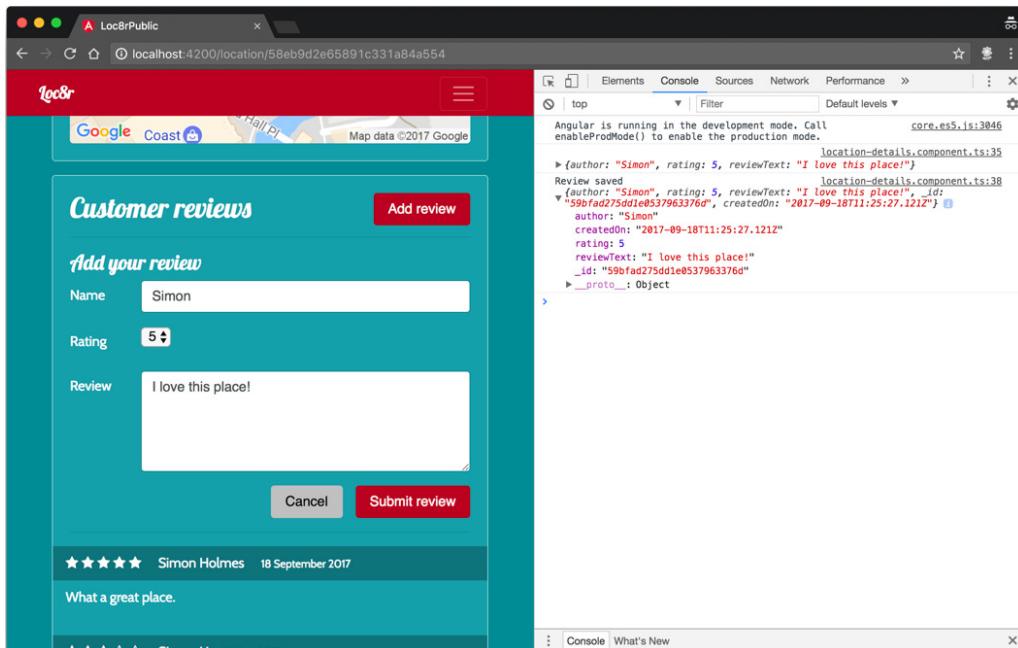


Figure 10.9 Console logs validating that reviews are being added to the database.

Just one last thing to make it slick in step 5: Push the review into list in Details page. In other words, when the review is sent we want to hide the form and add the review to the list that the user can see.

## **DISPLAYING THE REVIEW AND HIDING THE FORM**

Displaying the new review is thankfully a very simple task. We've got the list of reviews as an array, which is already sorted the most recent first. So we'll just need to use the native JavaScript `unshift` method to add the new review into the first spot in the array.

To hide the form we can just change `formVisible` to `false`, as that's what's controlling the `*ngIf` on the form. While we're at it, we can reset the values of the form so that it becomes blank again. The following listing shows all of the additions we need to put in `location-details.component.ts`.

**Listing 10.19 Hiding the form and showing the review in location-details.component.ts**

```

private resetAndHideReviewForm(): void {
    this.formVisible = false;
    this.newReview.author = '';
    this.newReview.rating = 5;
    this.newReview.reviewText = '';
}

public onReviewSubmit():void {
    this.formError = '';
    if (this.formIsValid()) {
        console.log(this.newReview);
        this.loc8rDataService.addReviewByLocationId(this.location._id, this.newReview)
        .then(review => {
            console.log('Review saved', review);
            this.location.reviews.unshift(review); ②
            this.resetAndHideReviewForm(); ③
        })
    } else {
        this.formError = 'All fields required, please try again';
    }
}

```

- ① A new private member to hide and reset the form
- ② Use unshift to add the review to the front of the array
- ③ Call the private member to hide and reset the form

And that is it! Almost. This won't work until `reviews` is in the class definition for the `Location` type, so we'll add it in as an array of type `any` in `home-list.component.ts` like this:

```

export class Location {
    _id: string;
    name: string;
    distance: number;
    address: string;
    rating: number;
    facilities: [string];
    reviews: [any];
}

```

And now that really is it. Our Angular SPA is complete, and fully functional. Well done! But there are a couple of things we can do to improve the architecture and follow some best practices.

## 10.3 Improving the architecture

We've now got a fully functioning SPA, which is awesome! But before we use it instead of the Express front-end, we can improve the architecture by taking the route configuration out of the `app.module.ts` file and the `location` class definition out of the `home-list.component.ts` file.

### 10.3.1 Using a separate route configuration file

Our first mission for improving the architecture and following an Angular best practice, is to move the route configuration into a separate file. Why is this a best practice? It largely comes down to separation of concerns. The purpose of the app.module.ts file is to tell the Angular compiler all about the app and the files it needs. If you just have a couple of routes it's okay to keep them in the app.module.ts file, but if you add more routes they end up taking over the file and masking the original purpose.

We've only got three routes in our application at the moment, but we'll explore this best practice by moving the routing configuration into a separate file. We'll also add more to this file when we look at authentication in the next chapter.

#### **CREATING A ROUTING CONFIGURATION FILE**

We can use the Angular CLI to generate the routing configuration file, this time using the `module` template. Run the following command in terminal in the `app_public` folder.

```
$ ng generate module app-routing
```

This will generate an `app-routing` folder (in `app/src`) containing an `app-routing.module.ts` file. We haven't seen one of these before, so listing 10.20 shows the default content of this file.

#### **Listing 10.20 The default module template of `app-routing.module.ts`**

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class AppRoutingModule { }
```

To add the application routing to this file we need to do the following:

1. Import the router module and routes type definition from Angular router
2. Import the components used for each the three routes
3. Define the paths and components for routes
4. Add the routes (using `routerModule.forRoot`) to module imports
5. Export `RouterModule` so that the setup can be used

This seems like quite a few steps, but it's not using anything we haven't already seen. We've used the router module and defined routes before, we're just putting them in a different place. All of the updates to the `app-routing.module.ts` are shown in the following listing.

**Listing 10.21 Completing the routing configuration in app-routing.module.ts**

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterModule, Routes } from '@angular/router'; ①

import { AboutComponent } from '../about/about.component'; ②
import { HomepageComponent } from '../homepage/homepage.component'; ②
import { DetailsPageComponent } from '../details-page/details-page.component'; ②

const routes : Routes = [ ③
  {
    path: '',
    component: HomepageComponent
  },
  {
    path: 'about',
    component: AboutComponent
  },
  {
    path: 'location/:locationId',
    component: DetailsPageComponent
  }
];

@NgModule({
  imports: [
    CommonModule,
    RouterModule.forRoot(routes) ④
  ],
  exports: [RouterModule],
  declarations: []
}) ⑤
export class AppRoutingModule { }

```

- ① Import the router module and route type definition
- ② Import the components for the routes
- ③ Define the routes as an array of type Routes ...
- ④ ... and import them using the router module
- ⑤ Export the router module

That's all there is to a routing configuration file. Next we need to update the main app.module.ts file to use this instead of the inline route definitions.

**TIDY UP THE APP.MODULE.TS FILE**

We don't want or need the route definitions in two files, so we can now delete them from the main module file. We also don't need to import the router from Angular router, so we can delete that line too - our new route configuration file handles importing that.

Although we're deleting the routes, we do need to keep the imports for all of the components. These are still required by app.module.ts, as this file that tells the compiler what to use and where to find the source files.

Finally, we need to add an import for the new router file, instead of the inline configuration. This import is normally placed between the core imports and component imports so that it is easy to spot when looking in the file. Also add the router file to the imports part of the decorator.

Listing 10.22 shows the final app.module.ts, with all of the additions and deletions having been made.

### **Listing 10.22 Removing inline route definitions from app.module.ts**

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { AppRoutingModule } from './app-routing/app-routing.module'; ①

import { HomeListComponent } from './home-list/home-list.component';
import { RatingStarsComponent } from './rating-stars/rating-stars.component';
import { DistancePipe } from './distance.pipe';
import { FrameworkComponent } from './framework/framework.component';
import { AboutComponent } from './about/about.component';
import { HomepageComponent } from './homepage/homepage.component';
import { PageHeaderComponent } from './page-header/page-header.component';
import { SidebarComponent } from './sidebar/sidebar.component';
import { HtmlLineBreaksPipe } from './html-line-breaks.pipe';
import { LocationDetailsComponent } from './location-details/location-
details.component';
import { DetailsPageComponent } from './details-page/details-page.component';
import { MostRecentFirstPipe } from './most-recent-first.pipe';

@NgModule({
  declarations: [
    HomeListComponent,
    RatingStarsComponent,
    DistancePipe,
    FrameworkComponent,
    AboutComponent,
    HomepageComponent,
    PageHeaderComponent,
    SidebarComponent,
    HtmlLineBreaksPipe,
    LocationDetailsComponent,
    DetailsPageComponent,
    MostRecentFirstPipe
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    AppRoutingModule ②
  ],
  providers: [],
  bootstrap: [FrameworkComponent]
})
export class AppModule { }
```

- ① Import our new routing module, containing the route configuration for the application
- ② Add it as an import for the app module

And that's all there is to it. The application will work as it did before, but I'm sure you'll agree that both the route configuration and main app module files are made better by this change. For a small application you may not want or need to do this, but if you're planning something big it's definitely worthwhile.

Next we'll take a look at how we can improve our `location` class definition.

### 10.3.2 Improving the location class definition

Our definition for the location class is currently held in `home-list.component.ts`. This stems from when we created the homepage listing component in chapter 8, and it was the only component really doing anything in the application. Now we are importing the location class definition in many places in the application; it is becoming a key part of the application in its own right. As such, it makes sense to separate it out into its own file.

When we do this we'll also add in the missing properties, as it currently only defines the properties that were used in the homepage listing - things like reviews and opening times are missing. Also, we'll create a nested class for reviews that we will be use to both in the class definition and in the application when dealing directly with reviews.

When all of this is done, we'll have a much better TypeScript application.

#### DEFINING THE LOCATION CLASS IN ITS OWN FILE

The first step is to create the file for the class definition, using the Angular CLI again:

```
$ ng generate class location
```

This will generate a file `location.ts` in the `src` folder of the application, and it's very sparse! It should look something like this:

```
export class Location {  
}
```

It's a little bit underwhelming, but at least there's nothing complex or unexpected! All we need to do is get the `Location` definition from `home-list.component.ts` and paste it in, as shown in the following listing.

#### **Listing 10.23 Adding the basic location class definition in `location.ts`**

```
export class Location {  
  _id: string;  
  name: string;  
  distance: number;  
  address: string;  
  rating: number;  
  facilities: [string];  
  reviews: [any];  
}
```

That's still pretty simple. The definition for the location class is now in its own file. We'd better start using it.

### **USE THE NEW CLASS FILE WHERE NEEDED**

The first place to use the new class definition file is in `home-list.component.ts` as that's where it was initially defined. To do this, simply delete the original inline definition from this file, and replace it with a simple import command as shown in the following snippet.

```
import { Component, OnInit } from '@angular/core';
import { Loc8rDataService } from '../loc8r-data.service';
import { GeolocationService } from '../geolocation.service';

import { Location } from '../location';
```

That's the replaced the location definition in the homepage listing, which is a good start. But if you're still running `ng serve` at this point you'll get Angular compilation errors along these lines:

```
Failed to compile.
/FILE/PATH/T0/LOC8R/app_public/src/app/location-details/location-details.component.ts
(3,10): Module ''/FILE/PATH/T0/LOC8R/app_public/src/app/home-list/home-
list.component'' has no exported member 'Location'.
```

This tells us that `location-details.component.ts` was using the `Location` class exported from `home-list`, so we need to update that too. All that needs to be done is change the file we're importing `Location` from, as shown in the following snippet.

```
import { Component, Input, OnInit } from '@angular/core';

import { Location } from '../location';
import { Loc8rDataService } from '../loc8r-data.service';
```

When that's done, do the same in the other places it's imported: `details-page.component.ts`, `location-details.component.ts` and `loc8r-data.service.ts`. Now let's add in the missing properties.

### **ADD MISSING PATHS FOR THE LOCATION CLASS DEFINITION**

When there are class properties that you use in your application but don't declare in the class definition, you run the risk of problems at build time. This can happen despite the fact that it may work just fine under `ng serve`.

We are currently missing `coords` and `openingTimes` from our class definition. `coords` is a simple addition, as that's just an array of numbers. `openingTimes` is a different deal though, as that's a complex object in its own right.

Remember how with Mongoose we could use nested schemas to define subdocuments? It's section 5.3.3 if you don't. Well, we can do the same thing with classes in TypeScript. Listing 10.24 shows how to update the `location.ts` file to define a class called `OpeningTimes`, and

define a property of the same name on the `Location` class, to be an array of the `OpeningTimes` type. It also adds in the `coords` property.

#### **Listing 10.24 Add missing properties and a nested class definition to location.ts**

```
class OpeningTimes {  
    days: string;  
    opening: string;  
    closing: string;  
    closed: boolean;  
}  
  
export class Location {  
    _id: string;  
    name: string;  
    distance: number;  
    address: string;  
    rating: number;  
    facilities: [string];  
    reviews: [any];  
    coords: [number];  
    openingTimes: [OpeningTimes];  
}
```

- ① Define a new `OpeningTimes` class
- ② Add the missing `coords` property to `Location`
- ③ Add the `openingTimes` property to the `Location` class, to be an array of the `OpeningTimes` class

Looking good. The class definition has all of the properties we need and use. Note that the `OpeningTimes` class is not available to be imported into other files by itself, as it is not declared as an `export`. Whilst this has everything we need, we can improve the `reviews` property definition.

#### **DEFINE A REVIEW CLASS, AVOIDING THE 'ANY' TYPE**

We've got `reviews` defined as an array of type `any`. This should be a bit of a red flag, as best practice in TypeScript is to try and avoid using `any` wherever possible as it weakens the class structure.

Here it's possible to avoid using `any`, as we know the schema of a review, and we've just seen how to define and use nested classes. Unlike the `OpeningTimes` definition, we'll want to use the `Review` class definition elsewhere in the application, so we'll declare this one as an `export`.

The following listing shows how to define the `Review` class, export it and use it inside the `Location` class definition. Note that the source code should also include the `OpeningTimes` definition, but I've left it out of this listing for brevity.

**Listing 10.25 Define, use and export a class for reviews in locations.ts**

```

export class Review { ①
  author: string;
  rating: number;
  reviewText: string;
}

export class Location {
  _id: string;
  name: string;
  distance: number;
  address: string;
  rating: number;
  facilities: [string];
  reviews: [Review]; ②
  coords: [number];
  openingTimes: [OpeningTimes];
}

```

- ① Define and export the class definition for reviews  
 ② Declare location reviews to be of type Review

Now our `Location` class is fully complete. We've got a nested class for reviews, which is available to be used elsewhere, and another nested class for opening times which is only available to this file. One final thing to tighten up our usage of the location class is to use the `Review` class within the application.

***EXPLICITLY IMPORT AND USE THE REVIEW CLASS WHERE NEEDED***

There are two places where we could make good use of the `Review` class, in the location details component where we use the form to add new reviews, and in our data service where we push the new review data to the API.

In the files for each of these - `location-details.component.ts` and `loc8r-data.service.js` - update the `Location` import to also import the `Review` class like this:

```
import { Location, Review } from '../location';
```

There are two places in the location details component where we can use the `Review` definition to add types to our variables, as shown in listing 10.26. The first is when we define `newReview` and give it default values, and the second is when the saved review is returned from the API.

**Listing 10.26 Update location-details.component.ts to use the new Review type**

```

public newReview: Review = {
  author: '',
  rating: 5,
  reviewText: ''
};

public onReviewSubmit():void {
  this.formError = '';
}

```

```

if (this.formIsValid()) {
  console.log(this.newReview);
  this.loc8rDataService.addReviewByLocationId(this.location._id, this.newReview)
    .then((review: Review) => {
      console.log('Review saved', review);
      this.location.reviews.unshift(review);
      this.resetAndHideReviewForm();
    })
} else {
  console.log('Not valid');
  this.formError = 'All fields required, please try again';
}
}

```

- ① Add the Review type to the newReview definition
- ② The saved review returned from the API should also be of type Review

In a similar way, we can tighten up the `addReviewByLocationId` method in our data service by specifying that the inputs and outputs should be of type `Review`, changing them from `any`. The three changes are shown in the following listing.

#### **Listing 10.27 Use the Review type to tighten up the definitions in loc8r-data.service.ts**

```

public addReviewByLocationId(locationId: string, formData: Review): Promise<Review> {
  const url: string = `${this.apiBaseUrl}locations/${locationId}/reviews`;
  return this.http
    .post(url, formData)
    .toPromise()
    .then(response => response.json() as Review)
    .catch(this.handleError);
}

```

- ① The incoming form data should be of type Review, as should the expected return value of the method
- ② The response of the API should also be of type Review, not any

That wasn't too painful, and we now have a much tighter application following some good TypeScript and Angular best practices. Using type definitions is very helpful for avoiding unexpected mistakes when passing around data, as it's very easy forget which parameter is supposed to be a string or an array, which properties an object should have and so on. This approach saves us from these problems, which is especially helpful when someone else is trying to read your code, or you return to it after a break and forget the finer details.

We are now in a position where we're happy with our SPA and want to use it as the front-end of our Loc8r application, replacing the current Express version.

## **10.4 Using the SPA instead of the server-side application**

In the final section of this chapter we'll build our Angular app for production and update Express to deliver this as the front-end instead of the Pug templates. As we go, we'll have to make adjustments to ensure direct access to deep URLs in the application without compromising the API routing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/getting-mean-with-mongo-express-angular-and-node-second-edition>

**Licensed to Jacob Munkholm Hansen <aau518331@uni.au.dk>**

Before we can do any of that we need to build the application. As we did at the end of chapter 8, run the `ng build` command in the `app_public` folder in terminal, specifying options to flag it as a production build with the output folder of `build`.

```
$ ng build -prod -op build
```

When that has finished running, you'll find a compiled version of the SPA in the folder `app_public/build`. This folder has everything the SPA needs to run, including the HTML page, JavaScript files, CSS, and fonts. Now to tell Express to use it.

#### **10.4.1 Routing Express requests to the build folder**

In order to get Express to serve the Angular app for the front-end, we need to do two things. Firstly, disable all of the previous routes for the front-end application, and secondly tell Express that our Angular build folder should serve static files.

To disable the Express-based routes for the front-end, find these two lines in `app.js` and delete them or comment them out.

```
const index = require('./app_server/routes/index');
app.use('/', index);
```

We also no longer need the `/public` folder for serving static files, as all of the files the Angular app needs are sitting inside the Angular build folder. Don't just delete that line though, as we need Express to serve the contents of the build folder as static files. So, find the following line in `app.js`:

```
app.use(express.static(path.join(__dirname, 'public')));
```

And update it to use the `app_public/build` folder like so:

```
app.use(express.static(path.join(__dirname, 'app_public', 'build')));
```

Alright. Run the Express app if it's not already running under nodemon, and head to `localhost:3000` in the browser. Everything we see there now is the Angular app, which we can validate by inspecting the elements of the page, as shown in figure 10.10.

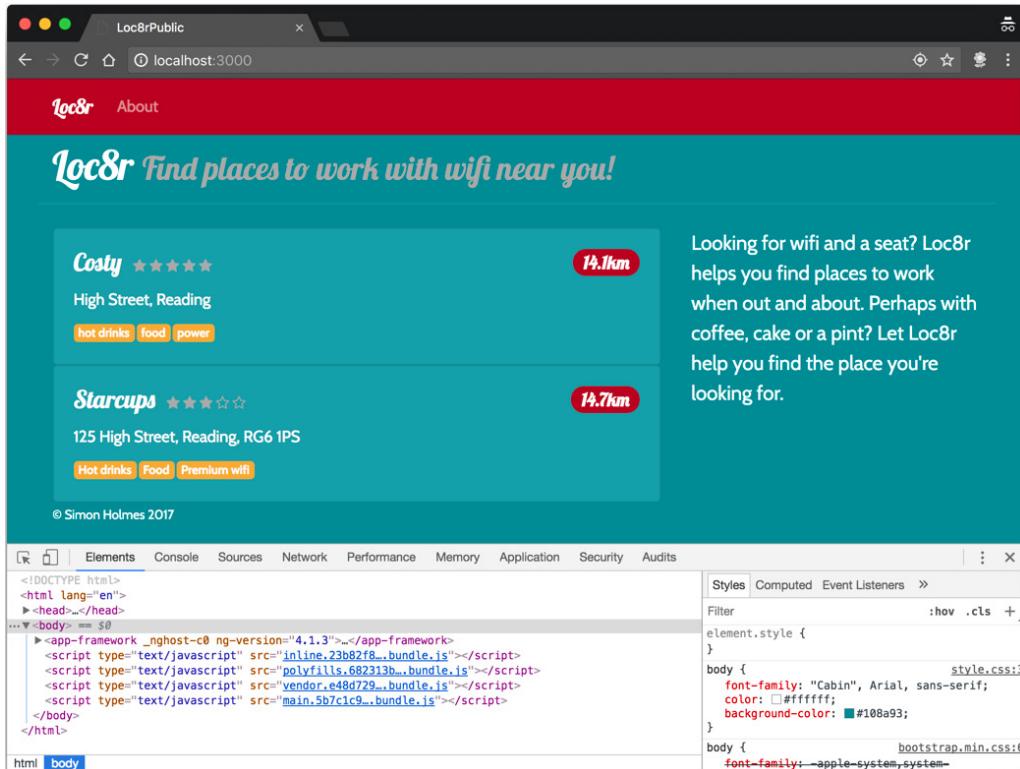


Figure 10.10 Visiting the homepage of the running Express app now delivers the Angular SPA

With these changes, when the homepage is requested Express serves the first matching resource, which is the index.html file in the app\_public/build folder. It is no longer matching an Express route and using a Pug template.

This works great for the homepage, and we can navigate through the app just fine. But if we take the URL for the about page or a location detail page and paste it into the URL bar we get a 404 error.

We need to fix this inability to directly access deep URLs, as it's not a very useful site if we only allow people to come in through the homepage.

#### 10.4.2 Making sure deep URLs work

This routing problem shouldn't be a great surprise. We've told Express to serve a static file for the homepage, but there's no *about* folder inside the *build* folder so it couldn't possibly know to show the Angular app.

A simple way to address this, is to let Express try to match the routes against everything it knows to exist, and then add a “catch all” route at the end to serve anything that hasn’t matched yet. This catch all route can be defined by using a \* as a wildcard for unmatched GET requests, and should respond by sending the index.html file for the Angular app.

The following snippet shows how to add the catch all route after all of the other route-matching statements in app.js - in this case after the definition for the API routes.

```
app.use('/api', apiRoutes);
app.get('*', function(req, res, next) {
  res.sendFile(path.join(__dirname, 'app_public', 'build', 'index.html'));
});
```

With this in place, if any URL isn’t matched by Express in the Angular build folder or the API routes, it will respond with the index page for the Angular app. This is good, but we can make it a bit better.

Rather than using a \* to match everything, we can actually use a regular expression to define the URLs it should do this for. The regular expression to match the /about route is quite simple, we just need to add start and end string delimiters and escape the forward slash, so it looks like ^\about\$.

The regular expression for a location detail page is a bit more complicated, due to the location ID. The location ID is a MongoDB ObjectId which is a 24 characters long, seemingly random mixture of number and lowercase letters. A regular expression to match these is [a-z0-9]{24}. Using the same approach as the about page regular expression, the complete one for the location details pages is ^\location\[a-z0-9]{24}\\$.

The following snippet shows how to update the catch all route in app.js with a combined regular expression to match either the about page or a locations detail page.

```
app.get(/(\about)|(\location\[a-z0-9]{24})/, function(req, res, next) {
  res.sendFile(path.join(__dirname, 'app_public', 'build', 'index.html'));
});
```

That’s a good change, as now Express will only send the Angular app as a response when a valid URL is entered. And with that, our SPA is fully working, being served up by Express, talking to the Express API, which in turn is getting data in and out of MongoDB.

In other words ... we’ve got a full MEAN stack application! Congratulations!

## 10.5 Summary

In this chapter we’ve covered the following:

- Taking the whole application code into the client-side
- Working with URL parameters in Angular, from defining the routes to using the parameters in components and services to query the API
- Angular template display logic in the form of `*ngIf` and `ngSwitch`
- Creating custom pipes to sort arrays of data

- Binding form data to component properties, and handling the submission of data
- Creating a separate route configuration file to improve the architecture
- Creating standalone class definitions, including nested classes, improving the use of custom type definitions through the application
- How to get Express to deliver an Angular application instead of server-side routes for certain URL requests

Coming up next in the final chapter we are going to see how to manage authenticated sessions, by adding the ability for users to register and log in before leaving reviews.

# A

## *Installing the stack*

### This appendix covers

- Installing Node and npm
- Installing Express globally
- Installing MongoDB
- Installing Angular

Before you can build anything on the MEAN stack you'll need to install the software to run it. This is really easy to do on Windows, Mac OS X, and the more popular Linux distributions like Ubuntu.

As Node underpins the stack, that's the best place to start. Node now also ships with npm included, which will be very useful for installing some of the other software.

### A.1 Installing Node and npm

The best way to install Node and npm depends on your operating system. When possible it's recommended to download an installer from the Node website at <http://nodejs.org/download/>. This location always has the latest version as maintained by the Node core team.

#### A.1.1 Long Term Support versions of Node

It's recommended to use a Long Term Support (LTS) version of Node; these are the releases with even major numbers – for example, Node 6 and Node 8. These are the stable branches of Node and will be maintained and patched with non-breaking changes for a period of 18 months. This book is built on Node 6, so that's probably the best LTS version to use.

### A.1.2 Installing Node on Windows

Windows users should simply download an installer from the Node website.

### A.1.3 Installing Node on Mac OS X

The best option for Mac OS X users is to simply download an installer from the Node website. Alternatively, you can install Node and npm using the Homebrew package manager, as detailed on Joyent’s Node wiki on GitHub at <https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager>.

### A.1.4 Installing Node on Linux

There aren’t any installers for Linux users, but you can download binaries from the Node website if you’re comfortable working with them.

Alternatively, Linux users can also install Node from package managers. Package managers don’t always have the latest version, so be aware of that. A particularly out-of-date one is the popular apt system on Ubuntu. There are instructions for using a variety of package managers, including a fix for apt on Ubuntu, on Joyent’s Node wiki on GitHub at <https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager>.

### A.1.5 Verifying installation by checking version

Once you have Node and npm installed you can check the versions you have with a couple of terminal commands:

```
$ node --version
$ npm --version
```

These will output the versions of Node and npm that you have on your machine. The code in this book is built using Node 6.9.4 and npm 3.10.10.

## A.2 Installing Express globally

To be able to create new Express applications on-the-fly from the command line, you need to install Express generator. You can do this from the command line, using npm. In terminal you simply run the following command:

```
$ npm install -g express-generator
```

If this fails due to a permissions error you’ll need to run this as an administrator. On Windows right-click the command prompt icon and select Run As Administrator. Now try the preceding command again in this new window. On Mac and Linux you can prefix the command with sudo as shown in the following code snippet; this will prompt you for a password:

```
$ sudo npm install -g express-generator
```

When the generator has finished installing Express you can verify it by checking the version number from terminal:

```
$ express --version
```

The version of Express used in the code samples in this book is 4.14.0.

If you run into any problems with this installation process, the documentation for Express is available on its website at <http://expressjs.com/>.

## A.3 Installing MongoDB

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

### A.3.1 Installing MongoDB on Windows

There are some direct downloads available from <http://docs.mongodb.org/manual/installation/> for Windows, depending on which version of Windows you're running.

### A.3.2 Installing MongoDB on Mac OS X

The easiest way to install MongoDB for Mac OS X is to use the Homebrew package manager, but if you prefer, you can also choose to install MongoDB manually.

### A.3.3 Installing MongoDB on Linux

There are also packages available for a few Linux distributions as detailed at <http://docs.mongodb.org/manual/installation/>. If you're running a version of Linux that doesn't have MongoDB available in a package, you can choose to install it manually.

### A.3.4 Running MongoDB as a service

Once you have MongoDB installed, you'll probably want to run it as a service so that it automatically restarts whenever you reboot. Again, there are instructions for doing this in the MongoDB installation documentation.

### A.3.5 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. You can check the version number of MongoDB and the Mongo shell independently. To check the shell version, run the following in terminal:

```
$ mongo --version
```

To check the version of MongoDB run this:

```
$ mongod --version
```

This book uses version 3.4.1 of both MongoDB and the Mongo shell.

## A.4 Installing Angular

Angular is very simple to install, as long as you have Node and npm already installed. What we actually install is the Angular CLI, as a global npm package. To do so, run the following in terminal:

```
$ npm install -g @angular/cli
```

# B

## *Installing and preparing the supporting cast*

### This appendix covers

- Adding Twitter Bootstrap and some custom styles
- Adding Font Awesome to give us a set of icons
- Installing Git
- Installing a suitable command-line interface
- Signing up for Heroku
- Installing Heroku CLI

There are several technologies that can help with developing on the MEAN stack, from front-end layouts to source control and deployment tools. This appendix covers the installation and setup of the supporting technologies used throughout this book. As the actual install instructions tend to change over time, this appendix will point you toward the best place to get the instructions and anything you need to look out for.

### B.1 Twitter Bootstrap

Bootstrap is not really installed as such, but rather added to your application. This is as simple as downloading the library files, unzipping them, and placing them into the application.

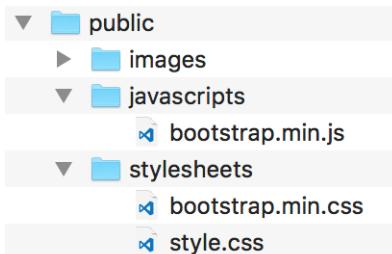
The first step is to download Bootstrap. This book uses version 4, which at the time of writing is an alpha release. You can get this from v4-alpha.getbootstrap.com but in time this will be available on the main website [www.getbootstrap.com](http://www.getbootstrap.com). Make sure that you download

the “ready to use files” and not the source. At the time of writing Bootstrap is on version 4.0.0-alpha.6 and the distribution zip contains two folders: css and js.

When you have it downloaded and unzipped, one file from each folder need to be moved into the public folder in your Express application:

1. Copy bootstrap.min.css into your public/stylesheets folder
2. Copy bootstrap.min.js into your public/js folder

Figure B.1 shows how the public folder in your application should now look.



**Figure B.1 The structure and contents of the public folder after Bootstrap has been added**

That will give you access to the default look and feel of Bootstrap, but you probably want your application to stand out from the crowd a bit. You can do this by adding in a theme or some custom styles.

### B.1.1 Adding some custom styles

The Loc8r application in this book uses some custom styles I have created, largely due to the lack of themes this early in the release of Bootstrap 4. As this version of Bootstrap 4 matures, hundreds of themes will become available, just as there are now for Bootstrap 3.

To add the custom styles, edit the style.css file in your public/stylesheets folder. Listing B.1 shows a good starting point.

#### B.1 Custom styles to give Loc8r a more distinctive look

```

@import url("//fonts.googleapis.com/css?family=Lobster|Cabin:400,700");

h1, h2, h3, h4, h5, h6 {
  font-family: 'Lobster', cursive;
}

.navbar-brand {
  font-family: 'Lobster', cursive;
}

legend {
  font-family: 'Lobster', cursive;
}
  
```

```
body {
  font-family: "Cabin", Arial, sans-serif;
  color: #ffffff;
  background-color: #108a93;
}
```

This is just a starting point, and will be added to as the application grows – to save you from typing you’ll be able to get the latest version of this files from GitHub in the project repo – <https://github.com/simonholmes/getting-MEAN-2>

## B.2 Font Awesome

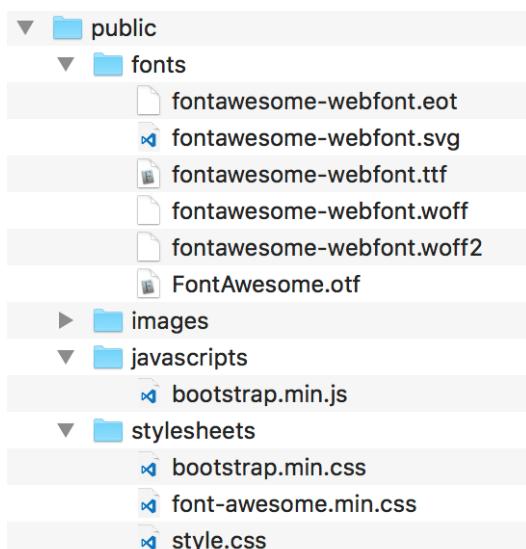
Font Awesome is an awesome way to get scalable icons into your application using fonts and CSS instead of images. Just like Bootstrap, there are a just a few files that need downloading and put in the right place.

First head over to [fontawesome.io](http://fontawesome.io) and click the download button to download a zip file. At the time of writing, we are using version 4.7.0, and the zip file contains two folders: css and fonts.

When it’s downloaded and unzipped, follow these two steps:

1. Copy the entire fonts folder into the public folder in your application
2. Copy the font-awesome.min.css file from the css folder into public/stylesheets

When that’s all done – and you’ve got Bootstrap installed as well – your public folder should look like figure B.2.



**Figure B.2** The structure and contents of the public folder after Font Awesome is added

Note that with Font Awesome the name and position of the fonts folder relative to the font-awesome.min.css file is very important. The CSS file references the fonts using a relative path of ../fonts/ so if this is broken the font icons won't work in your application.

### B.3 Installing Git

The source code for this book is managed using Git, so the easiest way to access it is with Git. Also, Heroku relies on Git for managing the deployment process and pushing code from your development machine into a live environment. So you need to install Git if you don't already have it.

You can verify if you have it with a simple terminal command:

```
$ git --version
```

If this responds with a version number then you already have it installed and can move onto the next section. If not, then you'll need to install Git.

A good starting point for Mac OS X and Windows users who are new to Git is to download and install the GitHub user interface from <https://help.github.com/articles/set-up-git>.

You don't need a GUI though, and you can install just Git by itself using the instructions found on the main Git website at <http://git-scm.com/downloads>.

### B.4 Installing a suitable command-line interface

You can get the most out of Git by using a CLI, even if you've downloaded and installed a GUI. Some are better than others, and you can't actually use the native Windows command prompt, so if you're on Windows then you'll definitely need to run something else. Here's what I use in a few different environments:

- Mac OS X Mavericks and later: native terminal
- Mac OS X pre-Mavericks (10.8.5 and earlier): iTerm
- Windows: GitHub shell (this comes installed with the GitHub GUI)
- Ubuntu: native terminal

If you have other preferences and the Git commands work, then by all means use what you already have and you're used to.

### B.5 Setting up Heroku

This book uses Heroku for hosting the Loc8r application in a live production environment. You can do this too—for free—so long as you sign up, install the CLI, and log in through terminal.

### B.5.1 Signing up for Heroku

To use Heroku you'll need to sign up for an account, of course. For the purposes of the application you'll be building through this book a free account will be fine. Simply head over to [www.heroku.com](http://www.heroku.com) and follow the instructions to sign up.

### B.5.2 Installing Heroku CLI

Heroku CLI contains the Heroku command-line shell and a utility called Heroku Local. The shell is what you'll use from terminal to manage your Heroku deployment, and Local is very useful for making sure what you've built on your machine is set up to run properly on Heroku. You can download the toolbelt for Mac OS X, Windows, and Linux from <https://devcenter.heroku.com/articles/heroku-cli>

### B.5.3 Logging in to Heroku using terminal

Once you've signed up for an account and installed the CLI on your machine, the last step is to log in to your account from terminal. Enter the following command:

```
$ heroku login
```

This will prompt you for your Heroku login credentials. Log in and you're all set up and ready to go with Heroku.

# C

## *Dealing with all of the views*

### This appendix covers

- Removing the data from all views, except the homepage
- Moving the data into the controllers

Chapter 4 covers setting up the controllers and the views for the static, clickable prototype. The “how” and “why” are covered in that chapter in more detail, so this appendix will really focus on what the end results should be.

### C.1 Moving the data from the views to the controllers

Part of this includes moving the data back down the MVC flow from the views into the controllers. The example in chapter 4 deals with this in the Loc8r homepage, but it needs to be done for the other pages too. We’ll start with the Details page.

#### C.1.1 Details page

The Details page is the largest and most complex of the pages, with the most data requirements, but following the homepage is the most logical place for it to go. The first step is setting up the controller.

##### **SETTING UP THE CONTROLLER**

The controller for this page is called `locationInfo` in the `locations.js` file in `app_server/controllers`. When you’ve analyzed the data in the view, and collated it into a JavaScript object, your controller will look something like the following listing.

**Listing C.1** locationInfo controller

```
const locationInfo = function(req, res){
  res.render('location-info', {
    title: 'Starcups',
    pageHeader: {title: 'Starcups'},
    sidebar: {
      context: 'is on Loc8r because it has accessible wifi and space to sit
[CA] down with your laptop and get some work done.',
      callToAction: 'If you\'ve been and you like it - or if you don\'t -
[CA] please leave a review to help other people just like you.'
    },
    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',
          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
[CA] about it.'
        },
        {
          author: 'Charlie Chaplin',
          rating: 3,
          timestamp: '16 June 2013',
          reviewText: 'It was okay. Coffee wasn\'t great, but the wifi was
[CA] fast.'
        }
      ]
    });
};
```

① Include latitude and longitude coordinates to use in Google Map image

② Add array of open times, allowing for different data on different days

③ Array for reviews left by other users

A part to note here is the latitude and longitude being sent through. You can get your current latitude and longitude from this website: <http://www.where-am-i.net/>.

You can geocode an address—that is, get the latitude and longitude of it—from this website: <http://www.latlong.net/convert-address-to-lat-long.html>. Your views will actually be using the `lat` and `lng` to display a Google Map image of the correct location, so it's worthwhile doing this for the prototype stage.

### **UPDATING THE VIEW**

As this is the most complex, data-rich page, it stands to reason that it will have the largest view template. You've already seen most of the technicalities in the homepage layout, such as looping through arrays, bringing in includes, and defining and calling mixins. There are a couple of extra things to look out for in this template though, both of which are annotated and highlighted in bold.

First, this template uses an `if-else` conditional statement. This looks like JavaScript without the brackets. Second, the template uses a JavaScript `replace` function to replace all line breaks in the text of reviews with `<br/>` tags. This is done using a simple regular expression, looking for all occurrences of the characters `\n` in the text. The following listing shows the `location-info.pug` view template in full.

#### **Listing C.2** `location-info.pug` view template in `app_server/views`

```
extends layout
include _includes/sharedHTMLfunctions
block content
  .row.banner
    .col-12
      h1= pageHeader.title
  .row
    .col-12.col-lg-9
      .row
        .col-12.col-md-6
          p rating
            +outputRating(location.rating)
          p 125 High Street, Reading, RG6 1PS
          .card.card-primary
            .card-block
              h2.card-title Opening hours
              each time in location.openingTimes
                p.card-text
                  | #{time.days} :
                  if time.closed
                    | closed
                  else
                    | #{time.opening} - #{time.closing}
            .card.card-primary
              .card-block
                h2.card-title Facilities
                each facility in location.facilities
                  span.badge.badge-warning
                    i.fa.fa-check
                    | &nbsp;#{facility}
                  | &nbsp;
```

```

.col-12.col-md-6.location-map
  .card.card-primary
    .card-block
      h2.card-title Location map
      img.img-
        fluid.rounded(src='http://maps.googleapis.com/maps/api/staticmap?center=${location.coords.lat},${location.coords.lng}&zoom=17&size=400x350&sensor=false&markers=${location.coords.lat},${location.coords.lng}&scale=2') ④
    .row
      .col-12
        .card.card-primary.review-card
          .card-block
            a.btn.btn-primary.float-right(href='/location/review/new') Add review
            h2.card-title Customer reviews
            each review in location.reviews ⑤
              .row.review
                .col-12.no-gutters.review-header
                  span.rating
                    +outputRating(review.rating)
                  span.reviewAuthor #{review.author}
                  small.reviewTimestamp #{review.timestamp}
                .col-12
                  p !{(review.reviewText).replace(/\n/g, '<br/>')} ⑥
            .col-12.col-lg-3
              p.lead #{location.name} #{sidebar.context}
              p= sidebar.callToAction

```

- ① Bring in sharedHTMLfunctions include, which contains outputRating mixin
- ② Call outputRating mixin, sending it rating of current location
- ③ Loop through array of open times, checking whether location is closed using an inline if-else statement
- ④ Build URL for Google Maps static image, inserting lat and lng using an ES2015 template string
- ⑤ Loop through each review, calling outputRating mixin again to generate markup for stars
- ⑥ Code replaces any line breaks in review text with <br/> tag so that renders as intended by author

A question that may arise from this is, why replace line breaks with <br/> tags every time? Why don't you just save the data with <br/> tags in? That way you only have to run the replace function once, when the data is saved. The answer is that HTML is just one method of rendering text; it just happens to be the one we're using here. Further down the line you may want to pull this information into a native mobile application. You don't want the source data tainted with HTML markup that you don't use in that environment. So the answer, really, is to keep the data clean.

### C.1.2 Add Review page

The Add Review page is really simple at the moment; there's only one piece of data in it: the title in the page header. So updating the controller shouldn't pose much of a problem. See the following listing for the full code of the `addReview` controller, in `locations.js` in the `app_server/controllers` folder.

**Listing C.3** addReview controller

```
const addReview = function(req, res){
  res.render('location-review-form', {
    title: 'Review Starcups on Loc8r',
    pageHeader: { title: 'Review Starcups' }
  });
};
```

Not much to talk about here; we've just updated the text inside the titles. The following listing shows the corresponding view, `location-review-form.pug`, in `app_server/views`.

**Listing C.4** location-review-form.pug template

```
extends layout
block content
  .row.banner
    .col-12
      h1= pageHeader.title
  .row
    .col-12.col-md-8
      form(action="/location", method="get", role="form")
        .form-group.row
          label.col-10.col-sm-2.col-form-label(for="name") Name
          .col-12.col-sm-10
            input#name.form-control(name="name")
        .form-group.row
          label.col-10.col-sm-2.col-form-label(for="rating") Rating
          .col-12.col-sm-2
            select#rating.form-control.input-sm(name="rating")
              option 5
              option 4
              option 3
              option 2
              option 1
        .form-group.row
          label.col-sm-2.col-form-label(for="review") Review
          .col-sm-10
            textarea#review.form-control(name="review", rows="5")
            button.btn.btn-primary.float-right Add my review
    .col-12.col-md-4
```

Again, nothing complicated or new here, so let's move on to the About page.

**C.1.3 About page**

The About page doesn't contain a huge amount of data either, just a title and some content. So let's pull that out of the view and into the controller. Note that the content in the view currently has some `<br/>` tags in it, so replace each `<br/>` tag with `\n` when you put it into the controller. These are highlighted in bold in the following listing; the `about` controller is in `app_server/controllers/others.js`.

**Listing C.5** about controller

```
const about = function(req, res){
  res.render('generic-text', {
    title: 'About Loc8r',
    content: 'Loc8r was created to help people find places to sit down and
[CA] get a bit of work done.\n\nLorem ipsum dolor sit amet, consectetur
[CA] adipiscing elit. Nunc sed lorem ac nisi dignissim accumsan. Nullam
[CA] sit amet interdum magna. Morbi quis faucibus nisi. Vestibulum mollis
[CA] purus quis eros adipiscing tristique. Proin posuere semper tellus, id
[CA] placerat augue dapibus ornare. Aenean leo metus, tempus in nisl eget,
[CA] accumsan interdum dui. Pellentesque sollicitudin volutpat ullamcorper.'
  });
};
```

Aside from removing the HTML from the content there's not much going on here. So let's take a quick look at the view, and we'll be done. The following listing shows the final generic-text view that is used for the About page in `app_server/views`—the view will have to use the same piece of code as we saw in the reviews section to replace the `\n` line breaks with HTML `<br/>` tags.

**Listing C.6** generic-text.pug template

```
extends layout
  .row.banner
    .col-12
      h1= title
  .row
    .col-12.col-lg-8
      p !{(content).replace(/\n/g, '<br/>')} ①
```

① Replace all line breaks with `<br/>` tags when rendering HTML

This is a really simple, small, reusable template for whenever you just want to output some text on a page.