SAMPLE CHAPTER

MANNING

Getting MEAN

with Mongo, Express Angular, and Node

Simon Holmes

Getting MEAN with Mongo, Express, Angular, and Node

by Simon Holmes

Chapter 1

brief contents

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 comprised of four main technologies, with a cast of supporting technologies:

- **M**ongoDB—the database
- **E**xpress—the web framework
- **A**ngularJS—the front-end framework
- **N**ode.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.

AngularJS is open source and backed by Google. It has been around since 2010 and is constantly being developed and extended.

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. 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.

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.

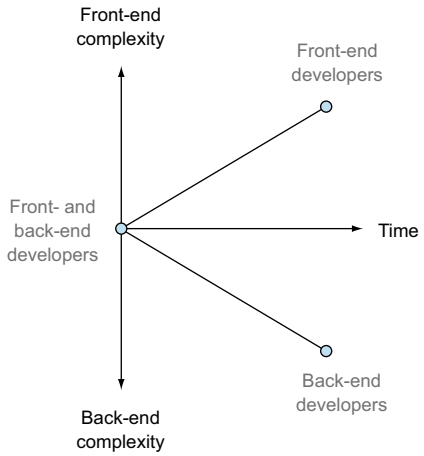

Figure 1.1 Divergence of front-end and back-end developers over time

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 CodeIgniter 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.

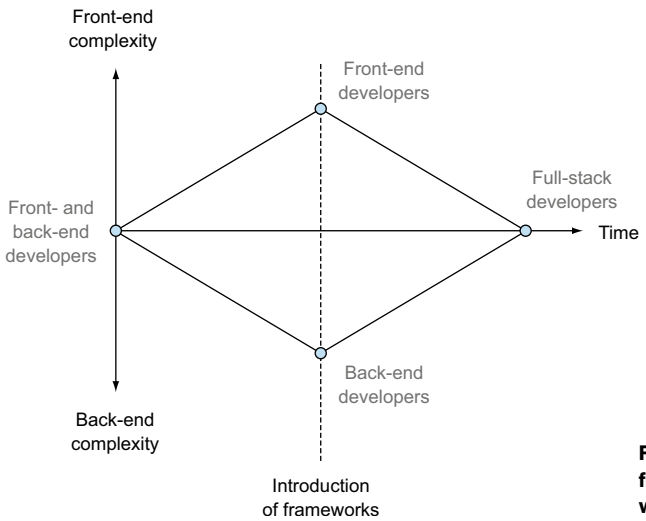

Figure 1.2 Impact of frameworks on the separated web development factions

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, as 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 AngularJS, Backbone, and Ember.

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 in effect doing 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, as they haven't been built in isolation by separate teams.

If you work as part of a large team then 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 data-driven application all by yourself.

The benefits when working in a team include

- 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. A majority of the most 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.

The principle technology allowing this 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 actually 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 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, as 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 while you could independently swap out any of the other components with an alternative technology, if you take Node.js out then 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 now, 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. The majority of 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. So 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 take a 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 1.3.

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,

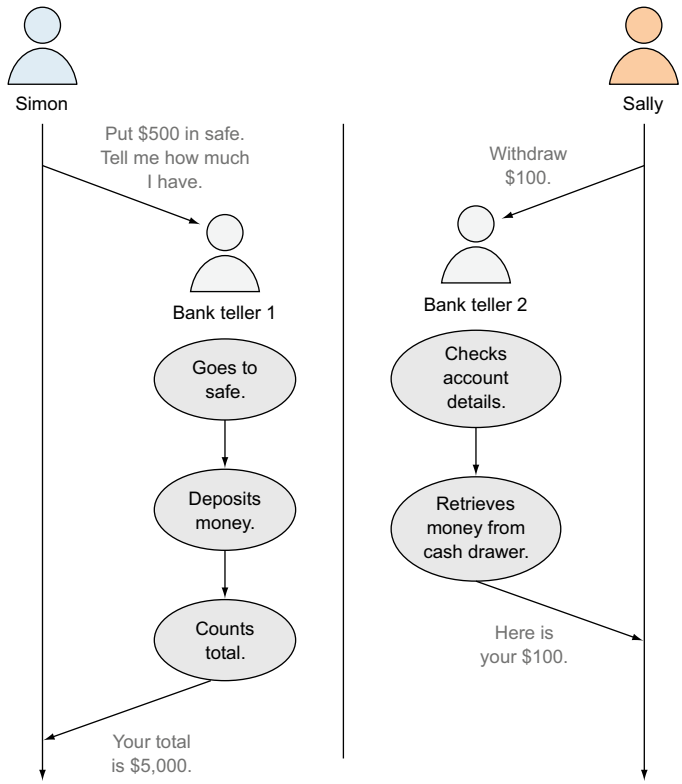

Figure 1.3 Example of a multithreaded approach: visitors use separate resources. Visitors and their dedicated resources have no awareness of or contact with other visitors and their resources.

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 way. Rather than giving each visitor a unique thread and a separate silo of resources, every visitor joins the same thread. A visitor and thread only interact 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 1.4 illustrates how this might work, using the same two requests from the multithreaded example.

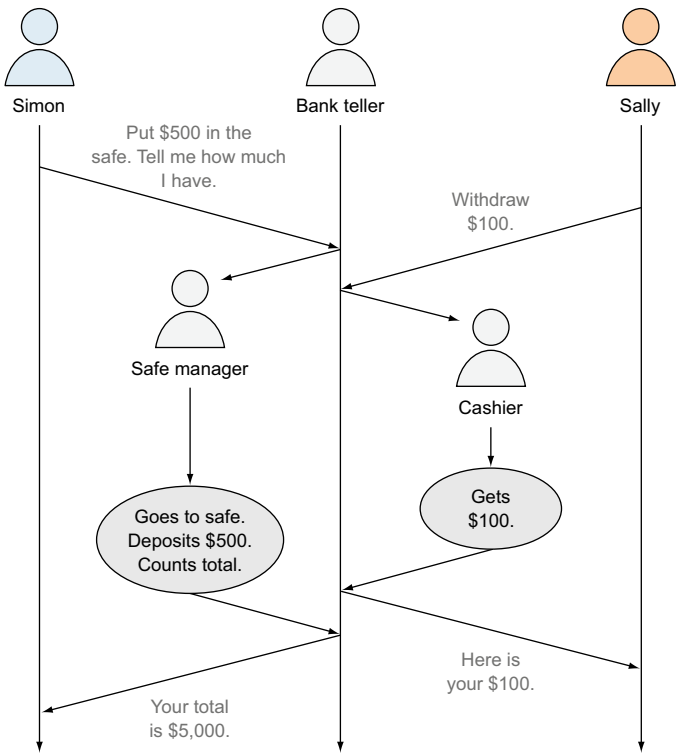

Figure 1.4 Example of a single-threaded approach: visitors use the same central resource. The central resource must be well disciplined to prevent one visitor from affecting others.

In the single-threaded approach shown in figure 1.4, 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 of them is impacted by the requests of the other. This approach means that the bank doesn't need a large number of 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 appendix D, particularly the section on callbacks.

1.2.3 Using prebuilt packages via npm

npm is a package manager that 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 46,000 packages available through npm, giving you an indication of just how much depth of knowledge and experience you can bring into the application.

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. Express has been created to give you a hand here. Express is installed using npm.

1.3 Introducing Express: The framework

Express is the E in MEAN. As 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 here 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 a number of 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 just sees a series of HTTP requests. HTTP is a stateless protocol, so there's no concept of storing a session state there. 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 a number of 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"`
`lastName: "Panayiotou"`	`middleName: "June"`	`firstName: "Sally"`	
`maidenName: "Holmes"`	`firstName: "Rebecca"`	`lastName: "Norman"`	`nickname: "Bec"`

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 appendix D, which can be found online at https://www.manning.com/books/getting-mean-with-mongo-express-angular-and-node. 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 me, 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, as 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 will assign 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 a number of 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 will take each of the operations independently; if one fails then it alone fails and the rest of the operations will 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 only *need* 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, as well as to save data 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 only allow 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 AngularJS: The front-end framework

AngularJS is the A in MEAN. In simple terms, AngularJS is a JavaScript framework for working with data directly in the front end.

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

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

1.5.1 jQuery versus AngularJS

If you're familiar with jQuery, you might be wondering if AngularJS 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. AngularJS comes in a step earlier and helps put together the HTML 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. AngularJS 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, AngularJS 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 start with a look at the traditional approach of one-way data binding. One-way data binding is what you're aiming for when looking at using Node.js, Express, and MongoDB. Node.js gets the data from MongoDB, and Express then uses a template to compile this data into HTML that's then delivered to the server. This process is illustrated in figure 1.5.

This one-way model is the basis for most database-driven websites. In this model most of the hard work is done on the server, leaving the browser to just render HTML and run any JavaScript interactivity.

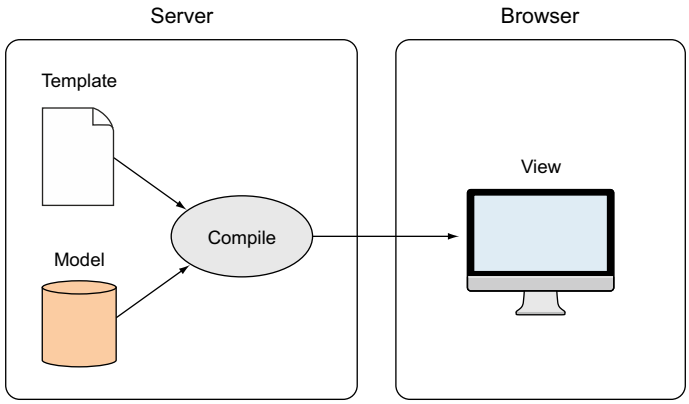

Figure 1.5 One-way data binding—the template and model are compiled on the server before being sent to the browser.

Two-way data binding is different. First, the template and data are sent independently to the browser. The browser itself compiles the template into the view and the data into a model. The real difference is that the view is "live." The view is bound to the model, so if the model changes the view changes instantly. On top of this, if the view changes then the model also changes. Two-way binding is illustrated in figure 1.6.

As your data store is likely to be exposed via an API and not tightly coupled to the application, there's typically some processing involved before adding it to the model. You want to make sure that you're binding the correct, relevant data to the view.

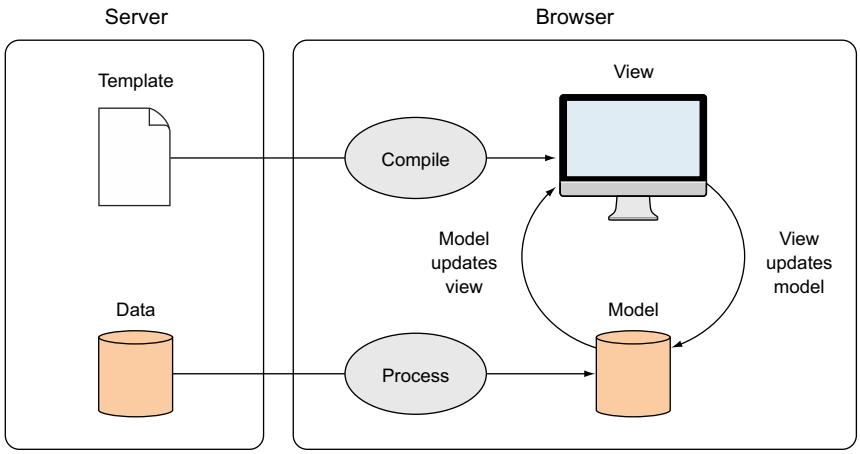

Figure 1.6 Two-way data binding—the model and the view are processed in the browser and bound together, each instantly updating the other.

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 AngularJS to load new pages

Something that AngularJS 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, as 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 AngularJS?

1.5.4 Are there any downsides?

Despite its many benefits, AngularJS 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 do use makes the experience better. That isn't the case with AngularJS, or indeed any other SPA framework. AngularJS 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 any JavaScript, and with AngularJS the only thing you get before JavaScript takes over is the template from the server. If you want your content and data indexed by search engines rather than just your templates, you'll need to think whether AngularJS 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 AngularJS—but if you don't *need* to fight this battle, I'd recommend against doing so.

One thing you can do is use AngularJS for some things and not others. There's nothing wrong with using AngularJS 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 AngularJS. You might also have a blog or some marketing pages around your application. These don't need to be built in AngularJS, 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 AngularJS 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.

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, flash-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 either need 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) 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. Once 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. Foursquare 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 1.7 shows some screenshots of what we're going to be building throughout the book.

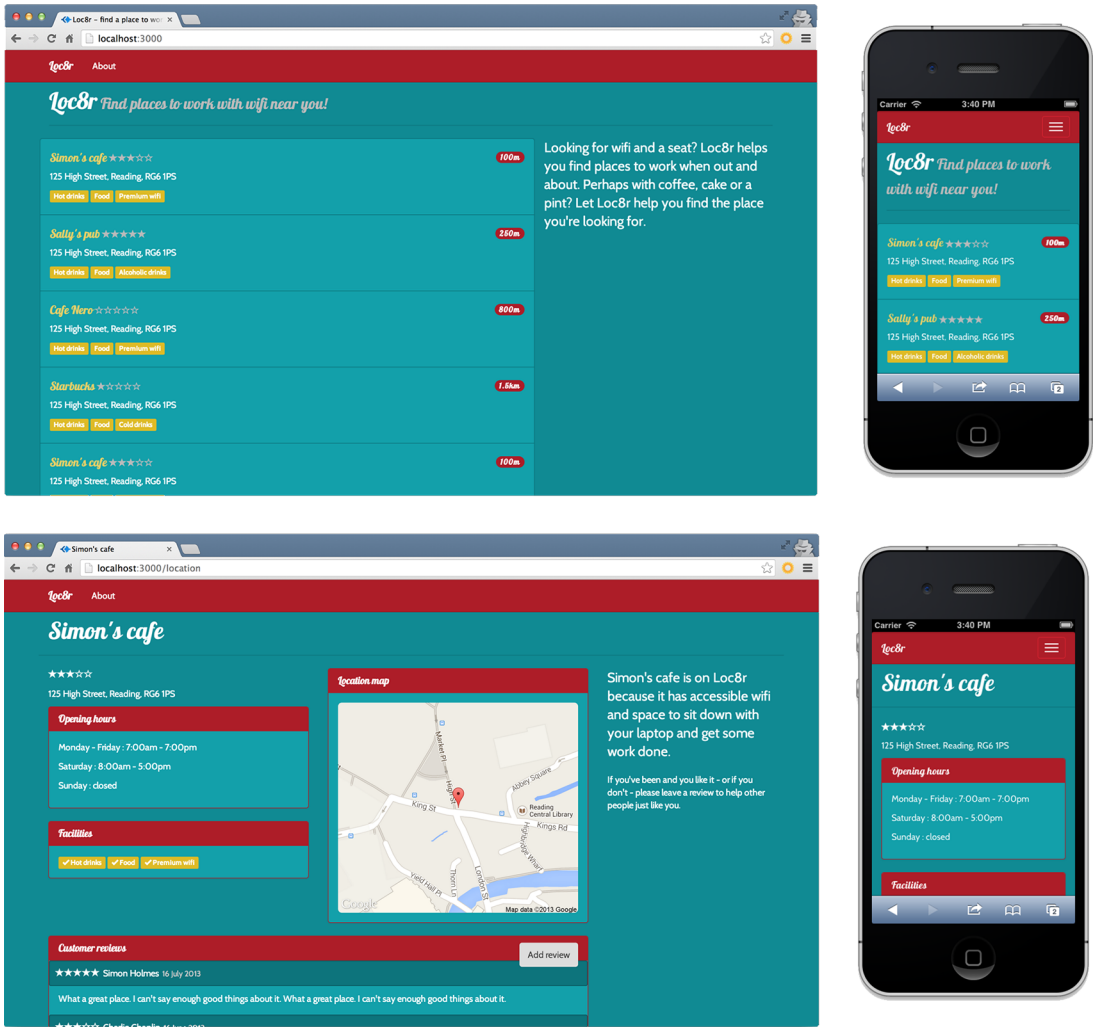

Figure 1.7 Loc8r is the application we're going to build throughout this book. It will display differently on different devices, showing a list of places and details about each place, and will allow visitors to log in and leave reviews.

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 of 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 the code is all written in JavaScript. In the front end is AngularJS, which again is JavaScript. Figure 1.8 illustrates this flow and connection.

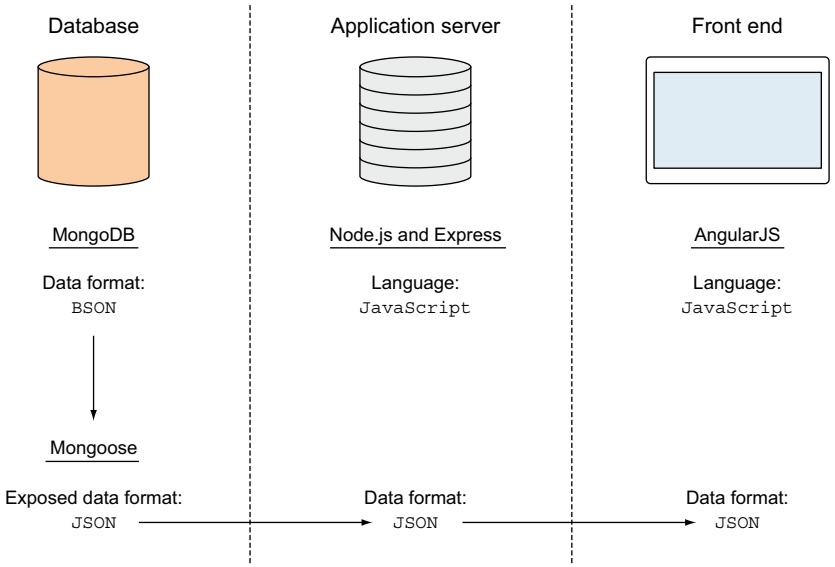

Figure 1.8 JavaScript is the common language throughout the MEAN stack, and JSON is the common data format.

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 we've covered

- 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
- AngularJS 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

As JavaScript plays such a pivotal role in the stack, please take a look at appendix D (available online), 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.

WEB DEVELOPMENT

Getting MEAN

with Mongo, Express, Angular, and Node

Simon Holmes

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

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

What's Inside

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

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

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

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

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

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

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

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

Free eBook
SEE INSERT

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

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]