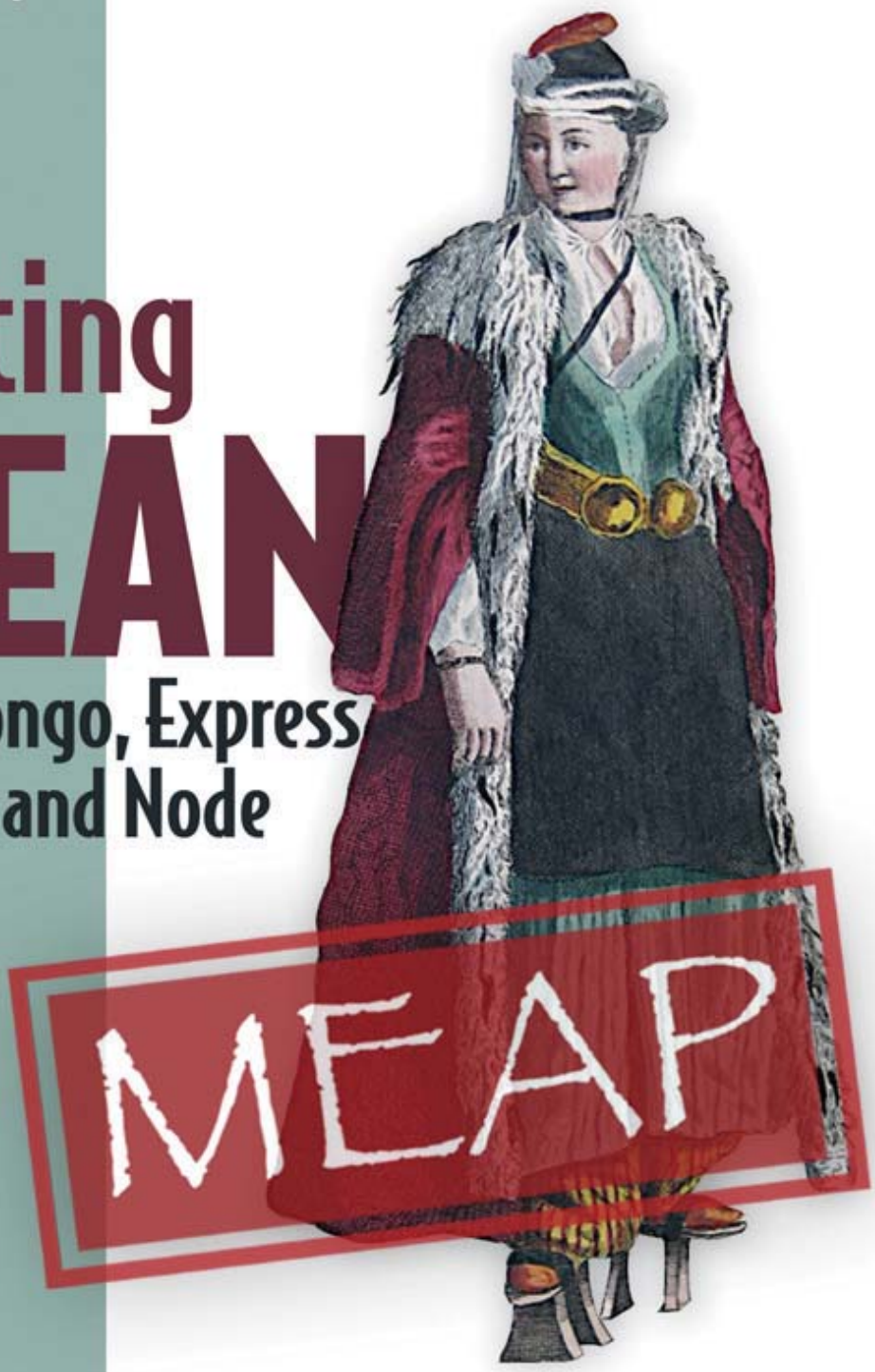
MANNING

Getting MEAN

with Mongo, Express Angular and Node



Simon Holmes

MEAP Edition
Manning Early Access Program
Getting MEAN with Mongo, Express, Angular, and Node
Version 1

For more information on this and other Manning titles go to
www.manning.com

Welcome

Thank you for purchasing the MEAP for *Getting MEAN with Mongo, Express, Angular, and Node*. 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 and to have a quick refresher on the important and relevant parts of JavaScript before diving in to building an application.

We're releasing the first two chapters to start. Chapter 1 covers what full stack development means, and what it looks like using the MEAN stack. By the end of Chapter 1 you'll have a good vision of how MongoDB, Express, AngularJS and Node.js work together to form the MEAN stack, understanding the role each part plays.

Chapter 2 takes a look at the most important parts of JavaScript, showing some best practices and exploring some of the concepts that are central to developing on the MEAN stack. By the end of Chapter 2 you should be confident in your ability to use the key concepts of writing JavaScript, and understand why the best practices are considered best practices.

Looking ahead, Part 2 of the book will 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 AngularJS 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

1

Introducing full stack development

This chapter covers

- The benefits of full stack development
- An overview of the components making up the MEAN stack
- 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 we mean by "full stack development" and look at the component parts of the stack to make sure we've got everything covered.

When we talk about "full stack development" we are 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 tech. The 'M', 'E', 'A' and 'N' are:

- **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 TJ Holowaychuk and has since become the most popular framework for Node.js. It is open-sourced with over 100 contributors and is actively developed and supported.

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

Node.js was created in 2009, and has it development and maintenance 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 is also very rewarding. And with the MEAN stack it is not 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 usage of the Internet started to spread, businesses started to take more of an interest in how their online presence portrayed them. In combination with the increased browser support of 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 HTML together, 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.

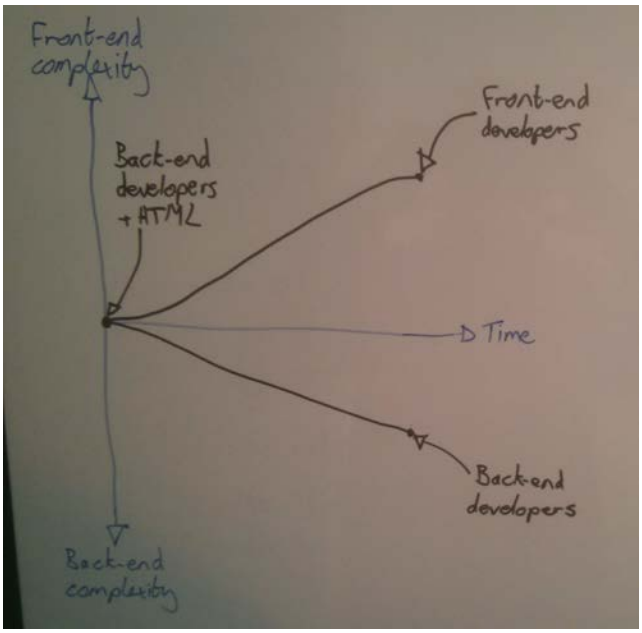


Figure 1.1 The divergence of front-end and back-end developers over time

So 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 2000's libraries and frameworks started to become popular and prevalent for the most common languages, on both the front-end and back-end. Think Dojo and jQuery for front-end JavaScript, CodeIgniter for PHP or 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 towards simplification has resulted in a resurgence of full-stack developers, who build both the front-end and the application logic behind it, as we can see in Figure 1.2.

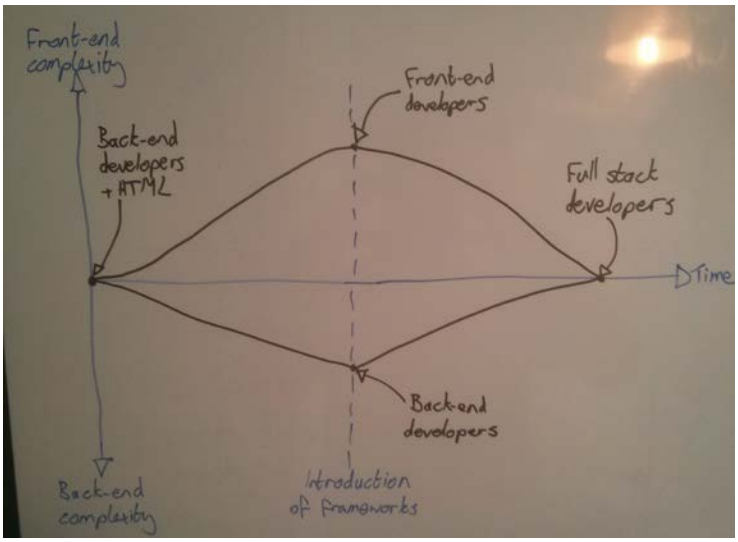


Figure 1.2 Impact of frameworks on the separated web development factions

Figure 1.2 illustrates a trend rather than proclaiming a definitive "all web developers should be full-stack developers" maxim. There were of course full-stack developers throughout the entire time so far, and moving forward it is 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 that 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 individuals can be incredibly productive, as they have an all-encompassing vision of the application and how it ties together.

MOVING THE APPLICATION CODE FORWARD IN THE STACK

Following on 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. You can 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 the traditional front-end developers and back-end developers. One of the reasons that people like to use this approach is that it reduces the load on your servers, thus reducing cost. What you are in effect doing is crowd-sourcing the computational power required for your application by pushing into the users' browsers.

We will discuss the pros and cons of this approach later in this 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 we have seen, the paths of front-end developers and back-end developers are coming back together, and it is entirely possible to be fully proficient in both disciplines. If you are a freelancer, consultant or part of a small team being multi-skilled 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 have not been built in isolation by separate teams.

If you work as part of a large team then the chances are that you will need to specialize in (or at least focus on) one area. It is, however, 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 to us today enhance this experience, and empower us to build great web applications relatively quickly and easily.

1.1.3 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 code both the front-end and the 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 is the least important - it is actually the foundation of the stack!

In a nutshell, Node.js is a software platform that allows you to create your own webserver and build web applications on top of it. Node.js is not itself a webserver, 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 IIS. This ultimately gives you greater control over how your web server works, but does increase the complexity of getting it up and running – particularly in a live environment.

With PHP for example, you can easily find a shared-server webhost running Apache, send some files up over FTP and – all being well – your site is running. This works because the webhost has already configured Apache for you and others to use. With Node.js this is not the case, as you configure the Node.js server when you create the application. Many of the traditional webhosts are behind the curve on Node.js support, but a number of new 'Platform as a Service' hosts are springing up to address this need. These include Heroku, Nodejitsu and Modulus. The approach to deploying live sites on these is different to 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. Up 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 backend.

> **Microsoft's foray into server-side JavaScript**
>
> In the late 1990's 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. This leads to the majority of books and online resources were for VBScript, so it snowballed into becoming the 'standard' language for Classic ASP.

Now, 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 already!

There is of course a learning curve when taking on Node.js, even if you are an experienced front-end JavaScript developer. The challenges and obstacles in server-side programming are different to 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 are 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 that – when coded correctly – it is 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 as 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 is single threaded, whereas traditional web servers are multithreaded. Let's take a look at what that means, starting off with the traditional multithreaded approach.

THE TRADITIONAL MULTITHREADED 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 8MB.

Thinking of a real world analogy, imagine two people going into a bank wanting to do separate things. In a multithreaded model they would each go to a separate bank teller who would deal with their requests. Take a look at Figure 1.3 that illustrates this.

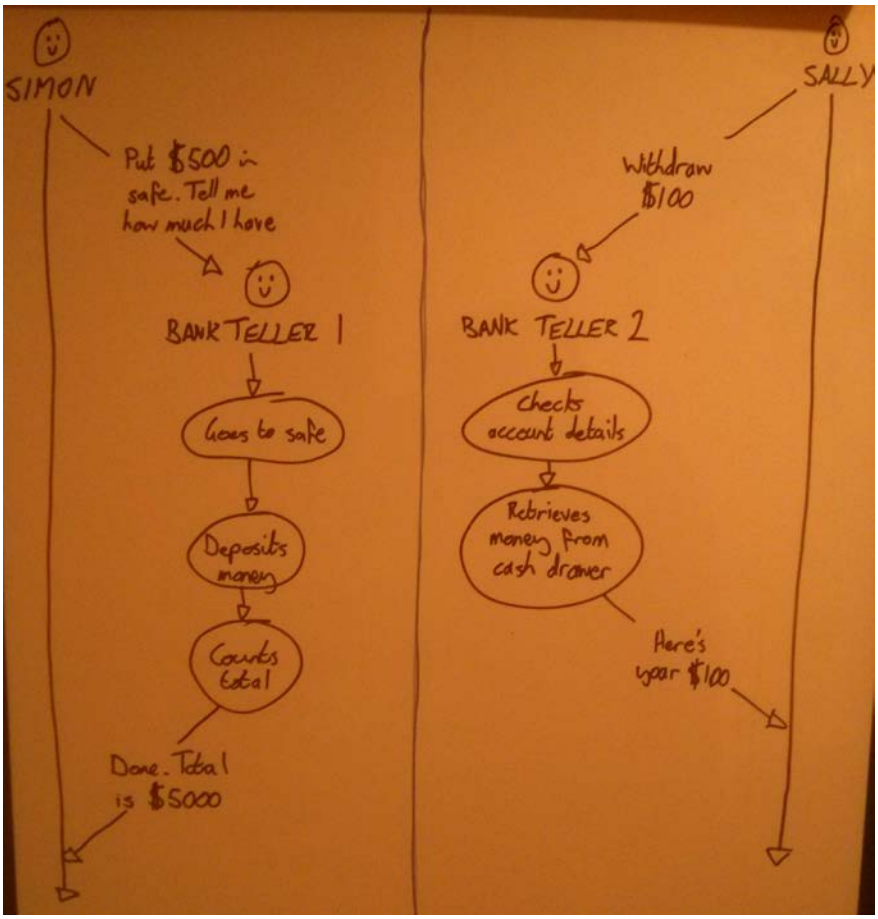


Figure 1.3 Example of a multithreaded approach: visitors use separate resources. Each visitor and their dedicated resources have no awareness of - or contact with – other visitors and their resources.

We can see here 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, so long as you have enough Tellers to service the customers. When the bank gets busy and the customers outnumber the Tellers, that is when the service starts to slow down and the customers have to wait to be seen. Whilst banks don't always worry about this too much, and seem happy to make you queue, the same is not true of websites. If a website is slow to respond you are likely to leave and never come back.

This is one of the reasons why webservers are often overpowered and have so much RAM, even though for 90% of the time you don't need it. 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 is a bit more scalable? Here's where a single threaded approach comes in.

A SINGLE THREADED WEBSERVER

A Node.js server is single threaded and works differently to the multithreaded way. Rather than giving each visitor a unique thread and a separate silo of resources, every visitor joins the same thread. The 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 our multithreaded example.

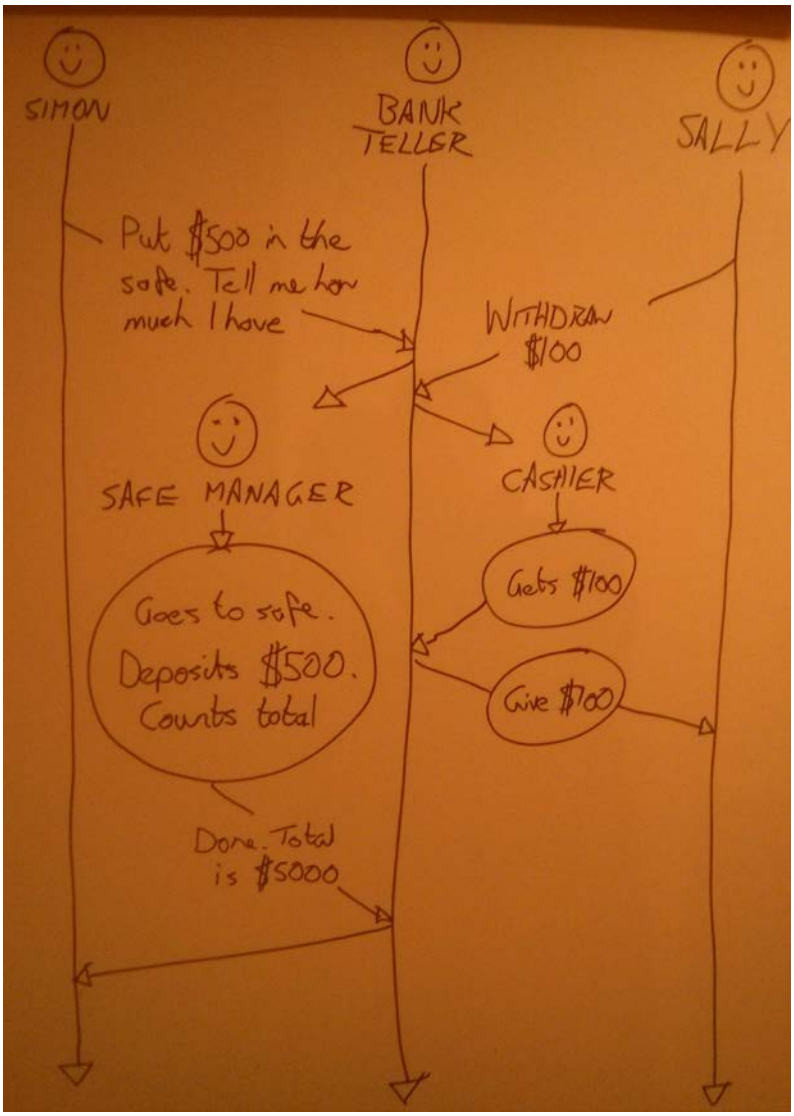


Figure 1.4 Example of a single threaded approach: visitors use the same central resource. The central resource must be well disciplined to prevent one visitor from impacting others.

In this single threaded approach, Sally and Simon both give their requests to the same Bank Teller. But instead of dealing with one of them entirely 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, they then pass this straight back to the visitor who requested it.

Despite there being just a single Teller, neither of the visitors is aware of the other, and neither of them were impacted by the requests of the other. This approach means that the bank doesn't need a huge number of Tellers constantly on hand. Now, this model isn't infinitely scalable of course, but it is 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, but we'll explore that further in Chapter 2, when we look at callbacks.

1.2.3 Using pre-built packages via npm

NPM is a package manager that gets installed when you install Node.js. What npm gives you is the ability to download Node.js modules or 'packages' to extend the functionality of your application. At the time of writing there are over 46,000 packages available through npm, giving you an indication of just how much depth of knowledge and experience you can bring into your application.

Packages in npm vary widely in what they give you. We will 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 such as *colors* which adds color support to Node.js console logs.

As we have seen, Node.js is extremely powerful and flexible, but doesn't give you much help when trying to create a website or application. Express has been created to give us 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. However, 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**, which has been designed to do this in a well-tested and repeatable way.

1.3.1 Easing your server setup

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

Express abstracts this difficulty away by setting up a webserver to listen to incoming requests and return relevant responses. On top of this it also defines a directory structure. One of these folders is set up to serve static files in a non-blocking 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 server 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 our code quicker to write and easier to maintain.

1.3.3 Views: HTML responses

It is likely that you will 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.

USING TEMPLATING ENGINES

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 complies 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. As it stands this would make it difficult to create a personalized experience 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'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 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 is renowned for being fast and scalable.

1.4.1 Relational vs. document databases

If you have used a relational database before, or even a spreadsheet you will be used to the concept of columns and rows. Typically a column defines the name and type of data 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 – the indented layout is for readability, not a visualization of columns.

Table 1.2 In a document database there is no concept of columns. Each document 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 we've got a better idea of what we're talking about.

1.4.2 MongoDB documents: JavaScript data store

MongoDB stores documents as BSON, which is binary JSON. Don't worry for now if you're not fully familiar with JSON, we'll take a look at it in Chapter 2. In short, JSON is a JavaScript way of holding data, hence why MongoDB fits so well into the JavaScript-centric MEAN stack!

The following 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 set of data 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 example MongoDB document. The _id entity is a unique identifier that MongoDB will assign to any new document when it is created.

We will 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 is not a transactional database, and should not 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 completes. 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 are 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*. You 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".

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 first name, last name, email address and phone number. But you

only *need* 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 apps. Mongoose makes it easier to manage the connections to your MongoDB database, as well as making it easier to save data and read data. We'll use all of this later. We'll also see 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, as 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 datastore 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.

We could use Node.js, Express and MongoDB to build a fully functioning data-driven web application. And we will do just this throughout this book. However, we can put some icing on this 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 just a moment when we introduce 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 vs. 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 DOM 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's known as an *opinionated framework*. This means that it forces its opinion on you, as to how it needs to be used.

We mentioned that 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 will 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 we were 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 into HTML that is then delivered to the server. This process is illustrated in Figure 1.5.

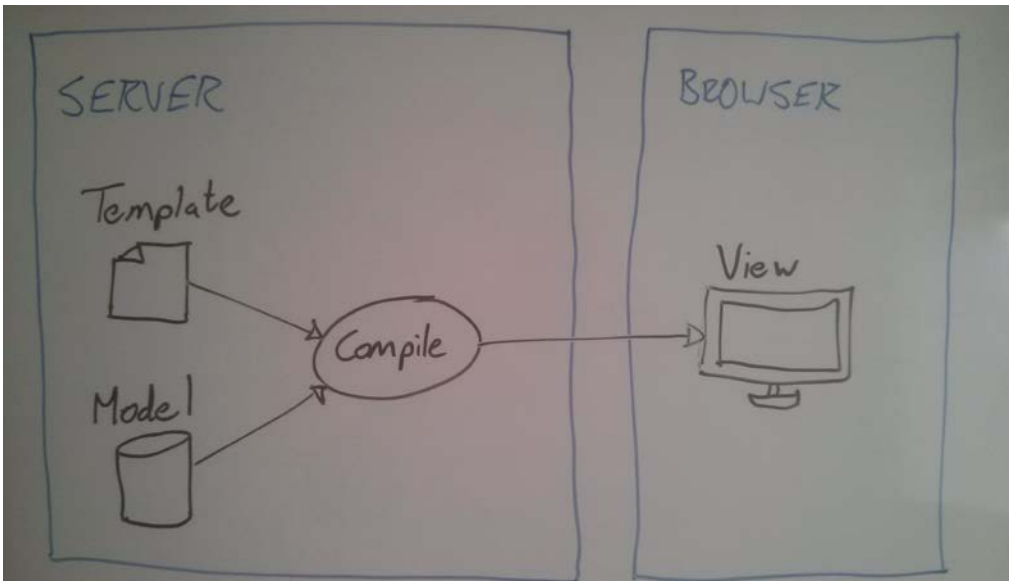


Figure 1.5 One-way data binding. The template and model are compiled on the server before being sent to the browser.

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.

Two-way data binding is different. Firstly, 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 that if the model changes the view changes instantly. On top of this, if the view changes then the model also changes. This is the two-way binding, and is illustrated in Figure 1.6.

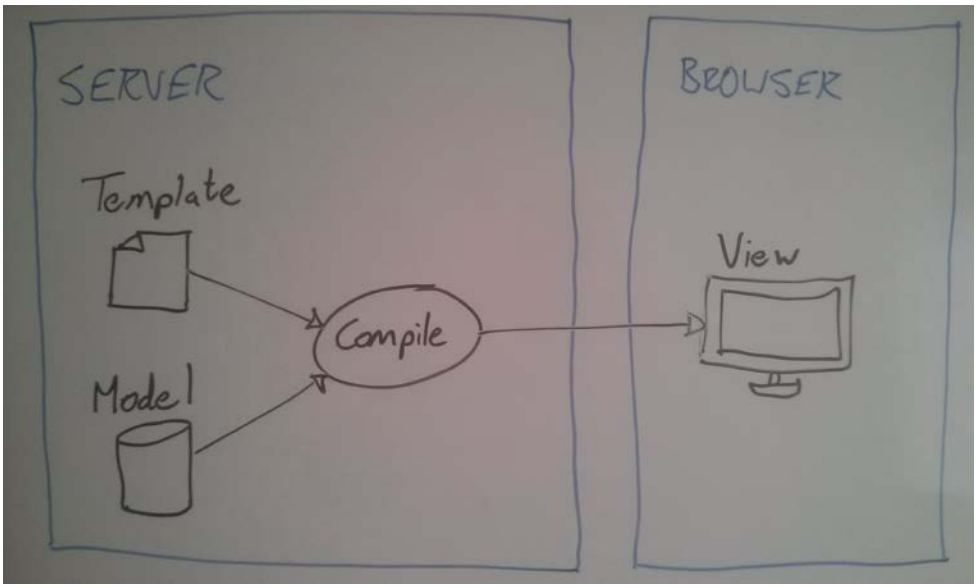


Figure 1.6 Two-way data binding. The model and the view are processed in the browser and bound together, each instantly updating the other.

As we go through Part 4 of the book you will 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** functionality. In real terms a Single Page Application 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 a Single Page Application. 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 are 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 greater 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 is not the case with AngularJS, or indeed any other Single Page Application 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 templates 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 would recommend against doing so.

One thing you can do is use AngularJS for some things and not others. There is 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 is ideal for building in AngularJS. You might also have a blog or some marketing pages around your app. 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 The supporting cast

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

1.6.1 Twitter Bootstrap for User Interface

In this book we're going to use Twitter Bootstrap to help us 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. However, 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. Amongst 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 pre-defined 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 is not an aim for this book, but we'll point out various features as and when we use them.

ADDING THEMES FOR A DIFFERENT FEEL

Bootstrap has a default look and feel, which provides a really neat baseline. This is so commonly used that your site could end up looking like anybody else's. Fortunately it is 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, but is also possible to buy premium themes from a number of sites online to give your application an even more 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 will 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 will 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.

Firstly, the source code of the sample application in this book will be stored on GitHub, with different branches for various milestones. You will be able to clone the master or the separate branches to use the code.

Secondly, we will use Git as the method of deploying our application to a live web server for the world to see. For hosting we will 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 apps, but the servers are generally not setup 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 *Platform as a Service* provider that is aimed specifically at hosting Node.js.

In this book we're going to go for the latter. We are going to use **Heroku** – www.heroku.com - as our hosting provider. Heroku is one of the leading hosts for Node.js applications, and 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 we've already mentioned a few times, throughout the course of this book you'll build a working application on the MEAN stack. This will give you a good grounding in each of the technologies as well as seeing how they all fit together.

1.7.1 Introducing the example application

So what are you actually going to be building as you go through the book? You'll be building an app called Loc8r. Loc8r will list nearby places with WiFi, where you 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.

REAL OR FAKE DATA?

Okay, so we're going to fake the data for Loc8r in this book, but you could collate the data, crowdsource 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 the process up.

THE END PRODUCT

We will use all layers of the MEAN stack to create Loc8r, including of course Twitter Bootstrap to help us create a responsive layout. Figure 1.7 shows some screenshots of what you're going to be building throughout the book.

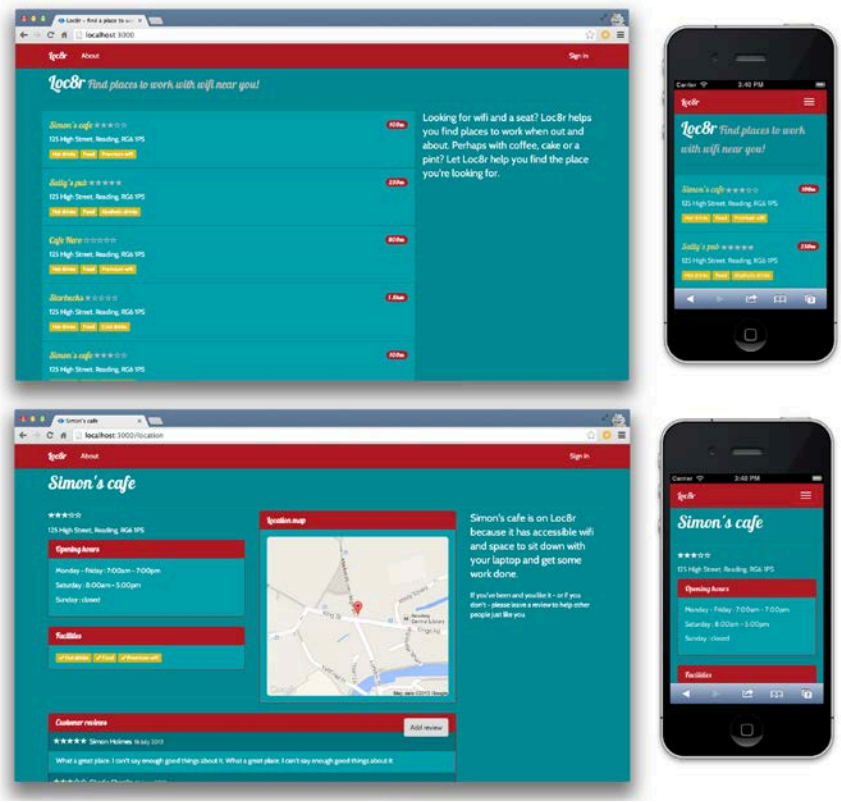


Figure 1.7 Loc8r is the application we are going to build through this book. It will display differently on different devices, showing a list of places, details about each place and have the ability for visitors to log in and leave reviews.

1.7.2 Breaking the development into stages

We will approach the project in the way I would personally go about building a rapid prototype. I break down the process into a number of stages, which lets you concentrate on one thing a time, increasing your chances of success. This approach also allows us to introduce the layers of that stack at different stages. You might choose to do your

development in a different way, but I find this approach works well for making an idea a reality.

STAGE 1: BUILDING A STATIC SITE

The first stage is to build a static version of the application, which is essentially a number of HTML screens. The aim of this stage is to quickly figure out the layout, and ensure that the user flows makes sense. At this point you are not concerned with a database or flashy effect on the user interface, all you want to do is create a working mockup of the main screens and journeys that a user will take through the application.

In the MEAN stack we do this using Node.js and Express, with a helping hand from Bootstrap to speed up the layout creation.

STAGE 2: CREATE THE DATABASE AND DESIGN THE DATA MODEL

Once you have a working static prototype that you are happy with, the next this to do is remove any hard-coded data from the application and put it into a database. The first part of this is to define the data model. Stepping back to a bird's eye view, what are the objects you need data about, how are the objects connected and what data is held in them?

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

In the MEAN stack we use MongoDB for this stage, relying heavily on Mongoose for the data modeling. The data models themselves will actually be defined inside the Express application.

STAGE 3: HOOK THE DATABASE INTO THE APPLICATION

After stages 1 and 2 you have a static site on one hand, and a database on the other. This next stage takes the natural step of linking them together. 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 is done, you'll have a data-driven application!

In the MEAN stack this stage in mainly done in Node.js and Express, with quite a bit of help from Mongoose. We use Mongoose to interface with MongoDB, rather than dealing with MongoDB directly.

STAGE 4: ENABLE USERS TO LOGIN

Stage 4 is optional, and really depends on the type of application you are building. It is quite common to want users to be able to log in to you application, perhaps to manage their account, get personalized information or submit data back to the application. Logging in also requires that the application maintains a session state, remembering the visitor from page to page. This stage is where you add this login ability and session management.

In the MEAN stack this stage uses Express to manage user sessions, with optional support from MongoDB. User authentication will typically use Node.js, Express, MongoDB and

Mongoose, but there are a number of third-party modules that you can plug in to your application so that you don't have to do all of the hard work.

STAGE 5: TURBO-CHARGE YOUR FRONT-END WITH ANGULARJS

By the time you get to this stage you should already have a fully functioning web application. Sometimes you will stop here, but a lot of the time you will want to carry on. This stage is where you get to ramp up your user interface, and bring it closer to the data. Perhaps you want to add instant filtering on lists of data, create an auto-suggest on text inputs or even convert your application into a Single Page Application.

In the MEAN stack this stage is all about using AngularJS. To support this you will most likely also change some of your Node.js and Express setup, and possibly move some of the functionality from your Express application into an AngularJS application.

1.7.3 The final MEAN stack architecture

By the time you've gone through all of the development stages you will 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 we have AngularJS, which again is JavaScript. You can see this flow and connection illustrated in Figure 1.8.

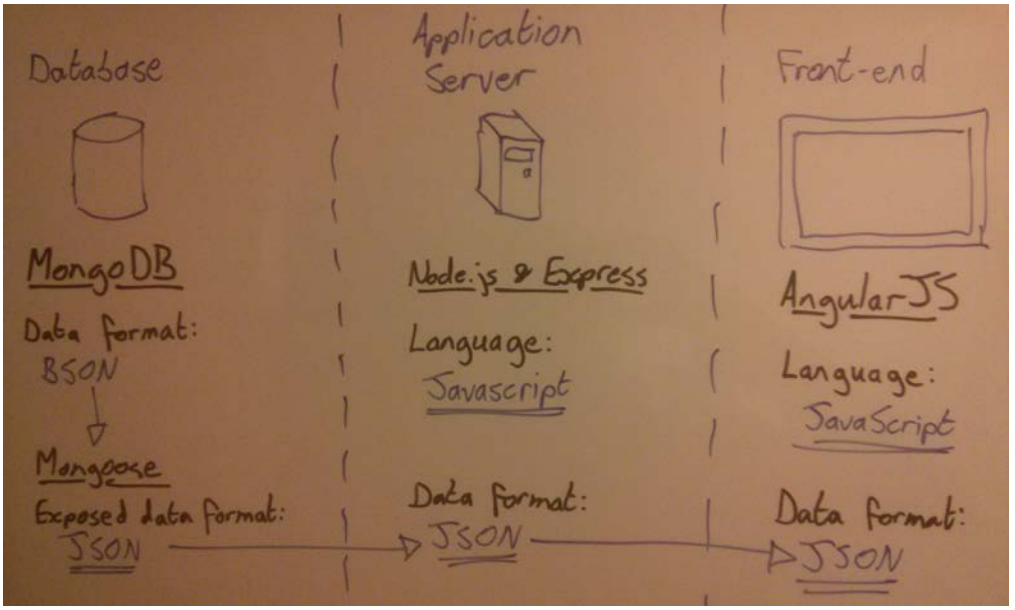


Figure 1.8 JavaScript is the common language throughout the MEAN stack, and JSON is the common data format

1.8 Summary

The MEAN stack gives developers the ability to create all aspects of a web application using just one language, JavaScript. Each of the components serves a distinct purpose, but they all work together to provide a very compelling end-to-end solution. MongoDB is used for the database layer, Node.js and Express work together to provide the application server layer, and AngularJS provides an amazing front-end data binding layer.

The MEAN stack provides a truly excellent core stack, but it is not a closed environment. Throughout this book we will use a few additional technologies to help us build a complete application. We'll only be scratching the surface of how the stack can be extended and manipulated to help you achieve your goals.

As JavaScript plays such a pivotal role in the stack, before we get stuck into building something let's start with a refresher on JavaScript pitfalls and best practice.