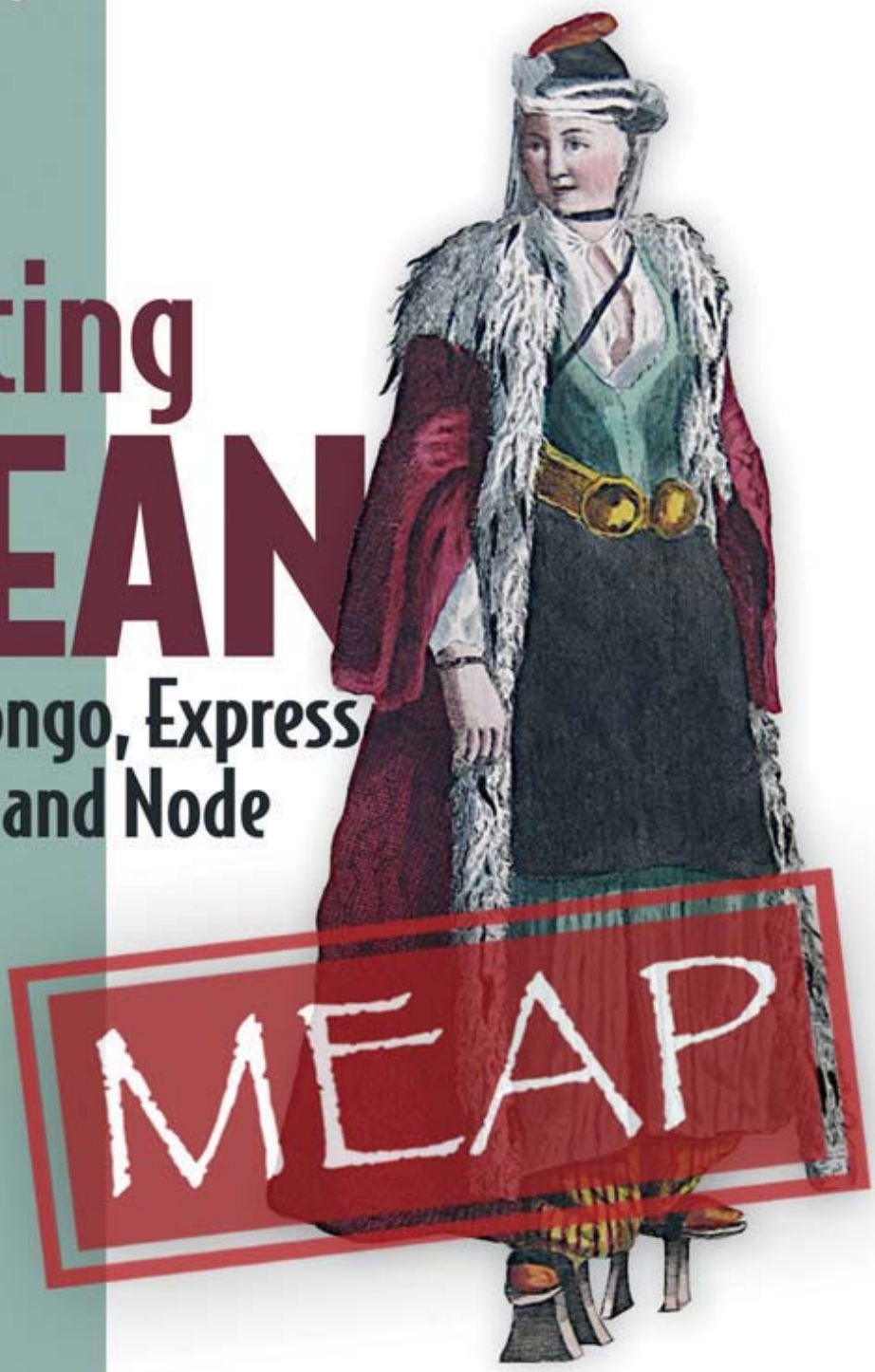
MANNING

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 2

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

PART 1: SETTING THE BASELINE

1 Introducing full stack development

2 Reintroducing JavaScript

PART 2: BUILDING A NODE WEB APPLICATION

3 Creating and setting up a MEAN project

4 Developing a static site with Node.js and Express

5 Building a data model with MongoDB and Mongoose

6 Writing an API: Exposing your MongoDB database to the application

7 Using your API from inside your application

8 Logging in users with Facebook and Twitter

PART 3: ADDING A DYNAMIC FRONT-END WITH ANGULARJS

9 Doing cool stuff with data in the browser

10 Changing pages without reloading

APPENDIXES:

Appendix A: Installing the stack

Appendix B: Installing and preparing the supporting cast

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.

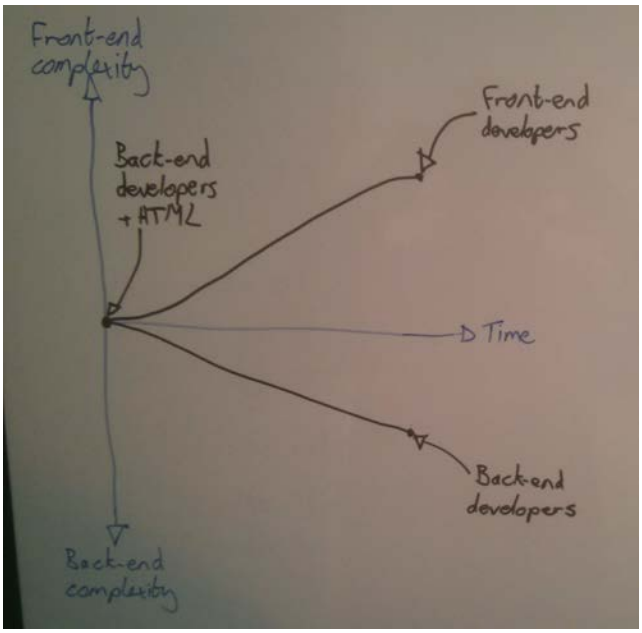


Figure 1.1 The divergence of front-end and back-end developers over time

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.

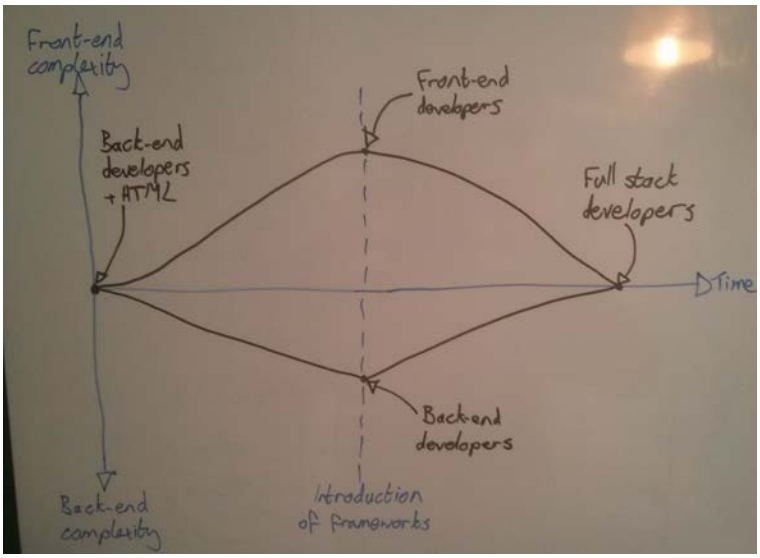


Figure 1.2 Impact of frameworks on the separated web development factions

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.

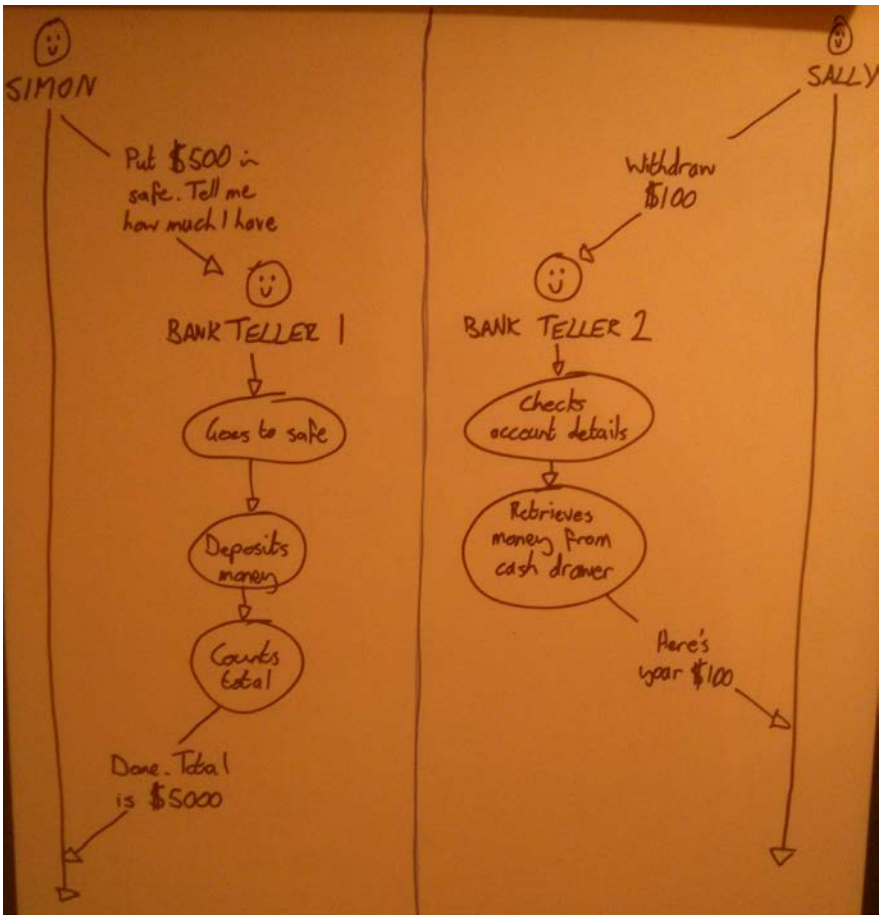


Figure 1.3 Example of a multithreaded approach: visitors use separate resources. Each visitor and their dedicated resources have no awareness of - or contact with – other visitors and their resources.

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.

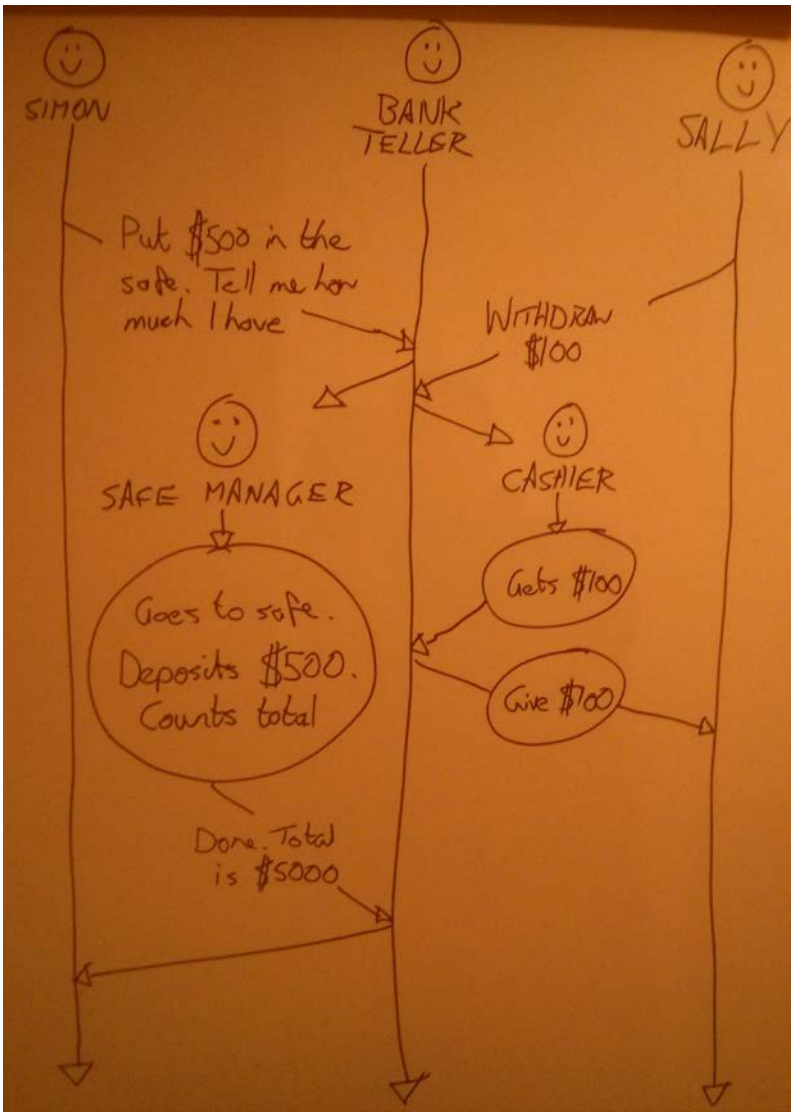


Figure 1.4 Example of a single threaded approach: visitors use the same central resource. The central resource must be well disciplined to prevent one visitor from impacting others.

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 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 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.

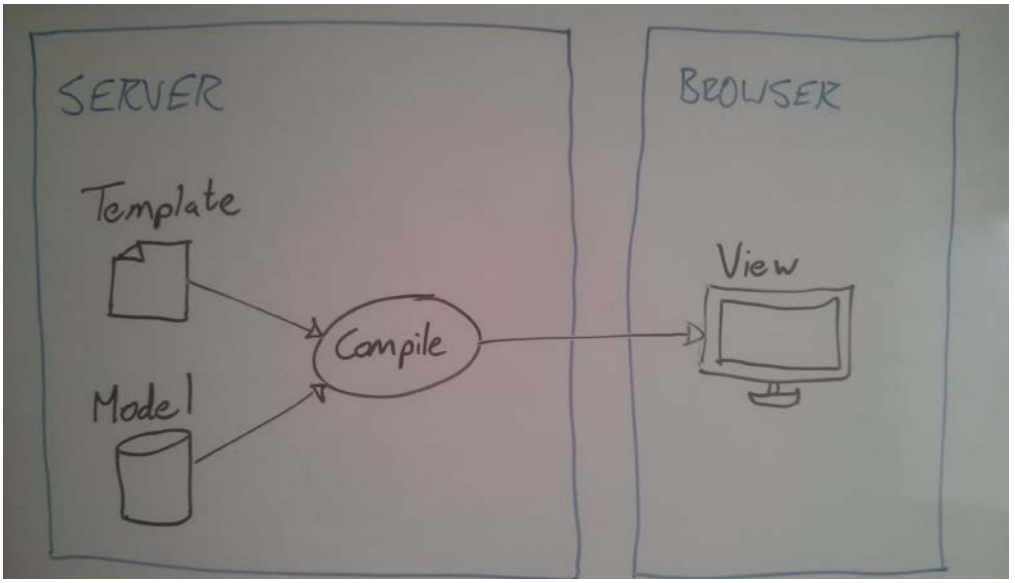


Figure 1.5 One-way data binding. The template and model are compiled on the server before being sent to the browser.

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.

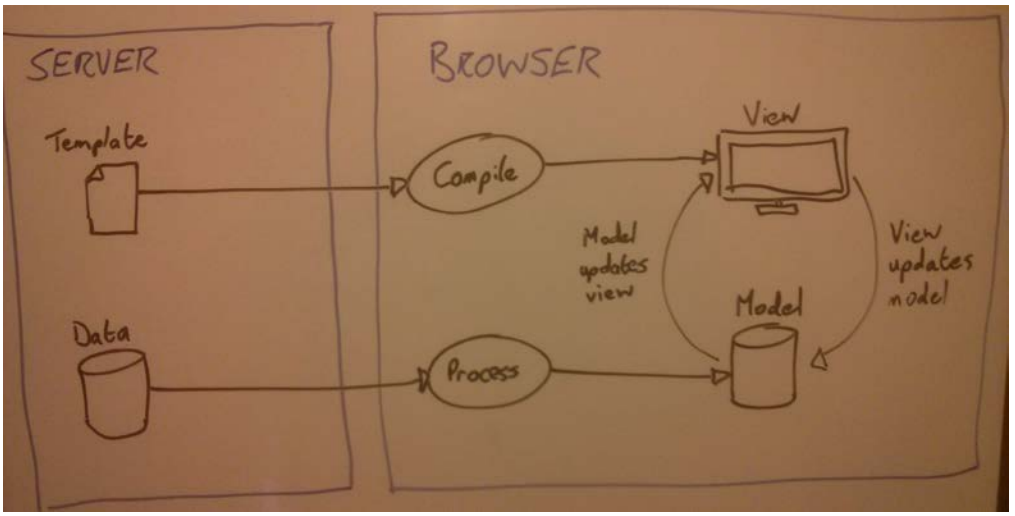


Figure 1.6 Two-way data binding. The model and the view are processed in the browser and bound together, each instantly updating the other.

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.

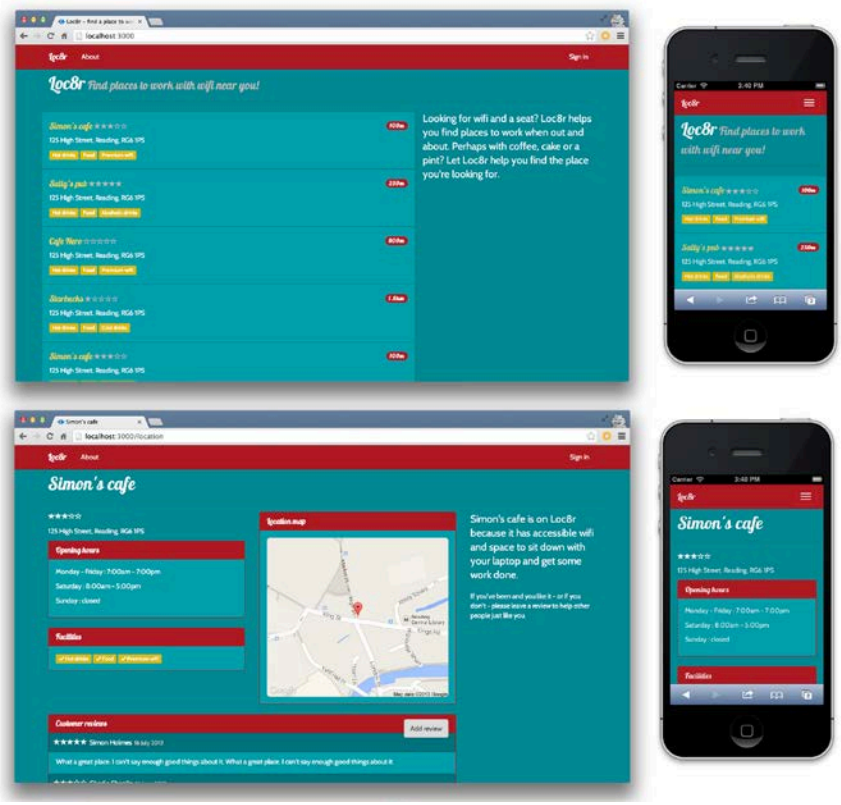


Figure 1.7 Loc8r is the application we are going to build through this book. It will display differently on different devices, showing a list of places, details about each place and have the ability for visitors to log in and leave reviews.

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.

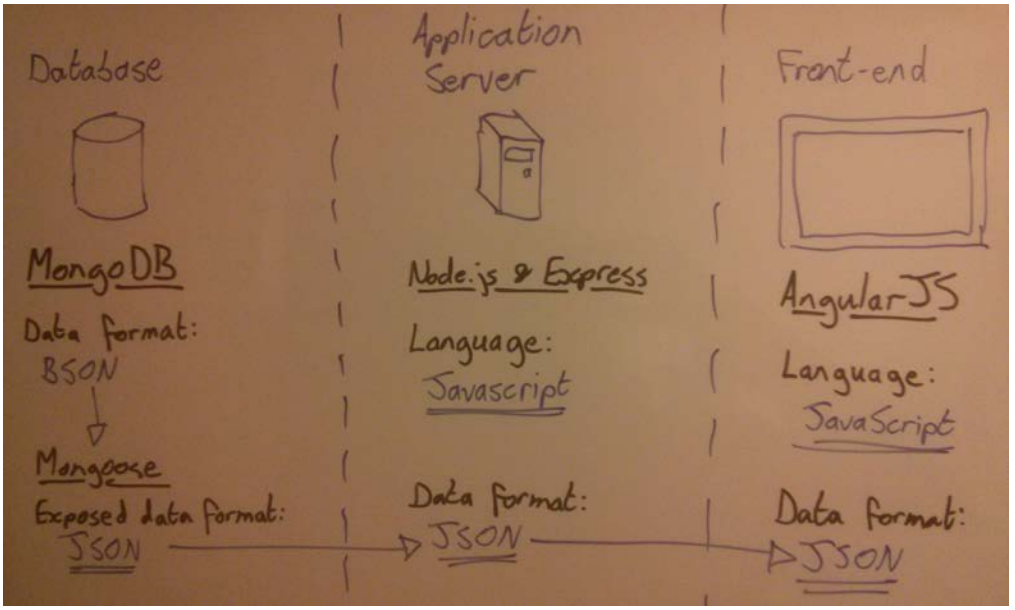


Figure 1.8 JavaScript is the common language throughout the MEAN stack, and JSON is the common data format

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.

2 Reintroducing JavaScript

This chapter covers

- Best practices when writing JavaScript
- Using JSON effectively to pass data
- Uncovering callbacks and escaping callback hell
- Writing modular JavaScript with closures and patterns

JavaScript is such a fundamental part of the MEAN stack that we're going to spend a little bit of time looking at it. We need to cover the bases as the successful MEAN development depends on it. JavaScript is such a common language that it seems everybody knows some of it. This is partly because JavaScript is really easy to get started with, and seems quite forgiving with the way that it is written. Unfortunately this looseness and low barrier to entry can encourage bad habits and sometimes have some quite unexpected results.

The aim of this chapter is not to teach JavaScript from scratch; if you don't know JavaScript at all you may struggle and find it hard going. On the other hand, not everybody will need to read this chapter in detail, particularly experienced JavaScript developers. If this is you, it's probably worth your while skimming through it just in case there's something new in there.

2.1 Everybody knows JavaScript, right?

Not everybody knows JavaScript, but the vast majority of web developers have used it at some point. Naturally there are different levels of knowledge and experience. As a quick test take a look at Listing 2.1.1. It contains a chunk of JavaScript code, the aim of which is to output a number of messages to the console. If you can understand the way the code is written, correctly say what the output messages will be, and more importantly understand why they are what they are then you're good for a skim-read!

Listing 2.1.1

```
var myName = {
    first: 'Simon',
    last: 'Holmes'
  },
  age = 37,
  country = 'UK';

console.log("1:", myName.first, myName.last);

var changeDetails = (function () {
  console.log("2:", age, country);
  var age = 35;
  country = 'United Kingdom';
  console.log("3:", age, country);

  var reduceAge = function (step) {
    age = age - step;
    console.log("4: Age:", age);
  };

  var doAgeIncrease = function (step) {
    for (var i = 0; i <= step; i++){
      window.age += 1;
    }
    console.log("5: Age:", window.age);
  },

  increaseAge = function (step) {
    var waitForIncrease = setTimeout(function(){
      doAgeIncrease(step);
    },step * 200);
  };

  console.log("6:",myName.first, myName.last, age, country);

  return {
    reduceAge : reduceAge,
    increaseAge : increaseAge
  };
})();

changeDetails.increaseAge(5);
console.log("7:", age, country);
changeDetails.reduceAge(5);
console.log("8:", age, country);
```

How did you get on with that? There are a couple of intentional ‘bugs’ in there, which JavaScript lets you make. This is all valid JavaScript and will run without throwing an error – you can test it by running it all in a browser if you like. The ‘bugs’ highlight how easy it is to get unexpected results, and also how difficult it can be to spot them if you don’t know what you’re looking for.

Want to know what the output of that code is? If you haven't run it yourself, here's the result of the code.

```
1: Simon Holmes
2: undefined UK                    #A
3: 35 United Kingdom
6: Simon Holmes 35 United Kingdom
7: 37 United Kingdom               #B
4: Age: 30                         #C
8: 37 United Kingdom
5: Age: 43                         #D
```

#A Age is undefined due to scope clashes and variable hoisting
#B Age has not changed but country has, due to variable scopes
#C Runs when called, not when defined. Uses local variables over global
#D Runs later due to setTimeout. Age is wrong due to 'mistake' in the for loop

Amongst other things, we've seen here a private closure exposing public methods, issues of variable scopes, variables not being defined when you think they should be, the effects of delays in asynchronous code and a really easy mistake to make in a for loop. So there was quite a lot to take in when reading the code.

If you're not sure what all of that meant, or didn't get the outcome correct, read on through this chapter and we'll explain all.

2.2 Good habits or bad habits

JavaScript is a pretty easy language to get started with. You can just grab a snippet from the Internet, pop it into your HTML page and you've started the journey. One of the reasons that it is easy to learn is that it's not always quite as strict as it possibly should be. It lets you do things that it possibly shouldn't, and this leads to bad habits. In this section we're going to take a look at some of these and turn them into good habits.

2.2.1 Variables, functions and scope

First stop here is to look at variables, functions and scope. All of these things are closely tied together. In JavaScript there are two types of scope, global scope and function scope. Function scope is often referred to as local scope. JavaScript also has scope inheritance. If you declare a variable in the global scope it is accessible by everything, if you declare a variable inside a function it is accessible only to that function and everything inside it.

WORKING WITH GLOBAL SCOPE AND LOCAL SCOPE, AND SCOPE INHERITANCE

Let's start with a simple example where scope is used incorrectly. See the following snippet.

```
var firstname = 'Simon';                  #A
var addSurname = function(){
  var surname = 'Holmes';                 #B
  console.log(firstname + ' ' + surname); #C
};
addSurname();
console.log(firstname + ' ' + surname);   #D
```

#A Variable declared in the global scope
#B Variable declared in the local scope
#C Outputs 'Simon Holmes'
#D Throws an error because surname isn't defined

This piece of code throws an error, because we are trying to use the variable `surname` in the global scope, but it was defined in the local scope of the function `addSurname`. A good way to visualize the concept of scope is to draw out some nested circles. The outer circle depicts the global scope and the inner circle depicts the local scope. See Figure 2.1.

Figure 2.1 Scope circles depicting global scope versus local scope, and scope inheritance.

In Figure 2.1 we can see that the global scope only has access to the variable `firstname`, and that the local scope of the function `addSurname` has access to the global variable `firstname` and the local variable `surname`.

If we want the global scope to be able to output the full name whilst keeping the surname private in the local scope we need a way of pushing the value into the global scope. In terms of the scope circles we are aiming for what we see in Figure 2.2. We want a new variable `fullname` that we can use in both global and local scopes.

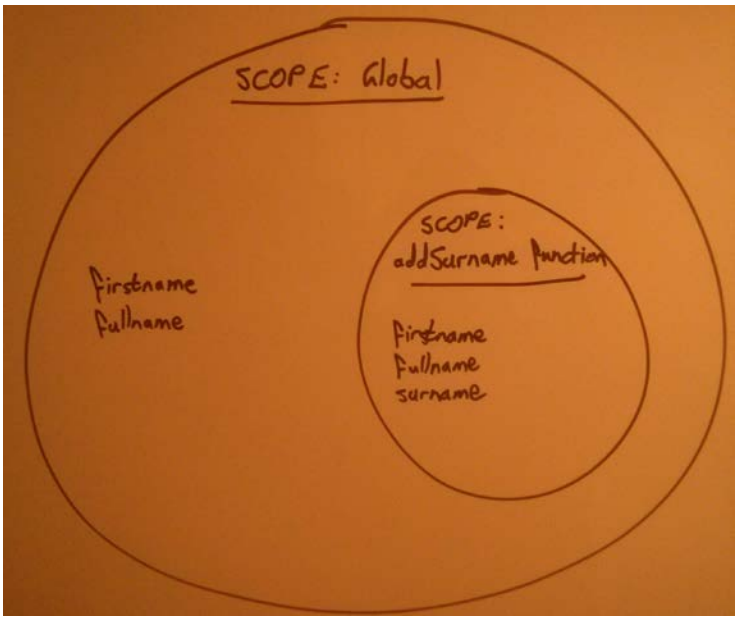


Figure 2.2 Using an additional global variable to return data from local scope

PUSHING FROM LOCAL TO GLOBAL SCOPE: THE WRONG WAY

One way you could do it – and I'll warn you now that this is bad practice – is to define a variable against the global scope from inside the local scope. In the browser, the global scope is the object `window`, in Node.js it is `global`. Sticking with browser examples for now, let's see how this would look if we update the code to use the `fullname` variable.

```
var firstname = 'Simon';
var addSurname = function(){
  var surname = 'Holmes';
  window.fullname = firstname + ' ' + surname;    #A
  console.log(fullname);
};
addSurname();
console.log(fullname);                            #B
```

#A The fullname variable is defined in the window object
#B The global scope can now output the full name

This approach allows us to add a variable to the global scope from inside a local scope, but it is not ideal. The problems are two-fold. Firstly, if anything goes wrong with the addSurname function and the variable is not defined, then when the global scope tries to use it you'll get an error thrown. The second problem with this really becomes obvious when your code gets large. Say you have dozens of functions all adding things to different scopes – how

do you keep a track of it? How do you test it? How do you explain to someone else what is going on. The answer to all of those questions is *with great difficulty*.

PUSHING FROM LOCAL TO GLOBAL SCOPE: THE RIGHT WAY

So if declaring the global variable in the local scope is wrong, what's the right way? The rule of thumb is **always declare variables in the scope in which they belong**. So if we need a global variable, then we should define it in the global scope, like in the following snippet.

```
var firstname = 'Simon',
    fullname;                       #A
var addSurname = function(){
  var surname = 'Holmes';
  window.fullname = firstname + ' ' + surname;
  console.log(fullname);
};
addSurname();
console.log(fullname);
```

#A Variable declared in the global scope, even if there isn't a value assigned to it yet

Here we have made it obvious that the global scope now contains the variable `fullname`. This makes the code easier to read when you come back to it.

REFERENCING GLOBAL VARIABLES FROM LOCAL SCOPE

You may have noticed that from within the function we still referenced the global variable using the fully qualified `window.fullname`. It is best practice to do this whenever referencing a global variable from a local scope. Again, this makes your code easier to come back to and debug, as you can explicitly see which variable is being referenced. So the code should look like this.

```
var firstname = 'Simon',
    fullname;
var addSurname = function(){
  var surname = 'Holmes';
  window.fullname = window.firstname + ' ' + surname;     #A
  console.log(window.fullname);                           #A
};
addSurname();
console.log(fullname);
```

#A When using global variables in a local scope always use the fully qualified reference

Using this approach might add a few more characters to your code, but it makes it really obvious which variable you're referencing and where it has come from. There is another reason for this, particularly assigning a value to a variable.

IMPLIED GLOBAL SCOPE

JavaScript lets you declare a variable without using `var`. This is a very bad thing indeed! Even worse, is that if you declare a variable without using `var`, JavaScript will create the variable in the global scope.

```
var firstname = 'Simon';
var addSurname = function(){
  surname = 'Holmes';                    #A
  fullname = firstname + ' ' + surname;  #A
  console.log(fullname);
};
addSurname();
console.log(firstname + surname);        #B
console.log(fullname);                   #B
```

#A surname and fullname are both defined in the global scope, by implication
#B They can be used in the global scope

Hopefully you can see how this could be confusing, and is a bad practice. So the take away here is **always declare variables in the scope in which they belong using the var statement.**

THE PROBLEM OF VARIABLE HOISTING

You've probably heard that with JavaScript you should always declare your variables at the top. That is correct, and the reason is because of **variable hoisting**. Variable hoisting means that JavaScript declare all variables at the top anyway, without telling you! This can lead to some unexpected results.

The following snippet shows an example of how this might show itself. In the `addSurname` function we want to use the global value of `firstname`, and then later declare a local scope value.

```
var firstname = 'Simon';
var addSurname = function(){
  var surname = 'Holmes';
  var fullname = firstname + ' ' + surname;      #A
  var firstname = 'David';
  console.log(fullname);                         #B
};
addSurname();
```

#A You expect this to use the global variable
#B But the output is actually "undefined Holmes"

So why is the output of this wrong? JavaScript 'hoists' all variable declarations to the top of their scope. So while you see the previous snippet JavaScript sees the following snippet.

```
var firstname = 'Simon';
var addSurname = function(){
  var firstname;                                 #A
  var surname = 'Holmes';
  var fullname = firstname + ' ' + surname;      #B
  firstname = 'David';
  console.log(fullname);
};
addSurname();
```

#A JavaScript has moved the declaration to the top
#B No value is assigned before it is used, so is undefined

When you see what JavaScript is actually doing, it makes the 'bug' a little more obvious. JavaScript has declared the variable `firstname` at the top of the scope, but it doesn't have a value to assign to it. This leaves it undefined when we first try to use it.

You should bear this in mind when writing your code. This is what JavaScript sees, so it should be what you see too. If you can see things from the same perspective, there is less room for error and unexpected problems.

NO SUCH THING AS BLOCK SCOPE

In various programming languages there is also the concept of block scope. This is a scope within – for example – an `if` statement or a `for` loop. JavaScript does not have block scope. The following snippet is incorrect in JavaScript.

```
var addSurname = function () {
  if (firstname === 'Simon') {
    var surname = 'Holmes';                    #A
  } else if (firstname === 'Sally') {
    var surname = 'Panayiotou';                #A
  }
};
```

#A Declaring variables inside blocks is wrong in JavaScript

JavaScript will let you do this, but it is technically incorrect and a bad practice. At the risk of sounding like a broken record, you should declare you variable at the top of the scope in which it belongs. The following snippet shows the best practice for doing this.

```
var addSurname = function () {
  var surname;                             #A
  if (firstname === 'Simon') {
    surname = 'Holmes';                    #B
  } else if (firstname === 'Sally') {
    surname = 'Panayiotou';                #B
  }
};
```

#A Variable is declared at the top of the scope
#B and then referenced within the two 'if' statements

FUNCTIONS ARE VARIABLES

You may have noticed throughout the preceding snippets that the `addSurname` function has been declared as a variable. Again this is a best practice. Firstly this is how JavaScript sees it anyway, and secondly it makes it very clear which scope the function is in.

While you can declare a function in this format:

```
function addSurname() {}
```

JavaScript interprets this as:

```
var addSurname = function() {}
```

So it is best practice to define functions as variables.

LIMITING USE OF THE GLOBAL SCOPE

We've talked a lot about using the global scope here, but in reality you should really try to limit your use of global variables. Your aim should be to keep the global scope as clean as possible. This really becomes important as applications grow. The chances are that you'll add it third-party libraries and modules. If these all use the same variable names in the global scope your application will go into meltdown.

Global variables are not 'evil' as some would have you believe, but you must be careful when using them. When you truly need global variables, a good approach is to create a container object in the global scope and put everything in there. Let's do this with our ongoing name example to see how it looks. We'll create a nameSetup object in the global scope and use this to hold everything else.

```
var nameSetup = {                                                    #A
  firstname : 'Simon',
  fullname : '',
  addSurname : function(){
    var surname = 'Holmes';                                          #B
    nameSetup.fullname = nameSetup.firstname + ' ' + surname;        #C
    console.log(nameSetup.fullname);                                 #C
  }
};
nameSetup.addSurname();                                              #C
console.log(nameSetup.fullname);                                     #C
```

#A Declare a global variable as an object
#B Local variables are still okay inside functions
#C Always access the values of the object using the fully qualified reference

When you code like this all of your variables are held together as properties of an object, keeping the global space nice and neat. Working like this also minimizes the risk of having conflicting global variables. You can of course add more properties to this object after declaration, even adding new functions. Adding to the previous code sample we could have the following snippet.

```
nameSetup.addInitial = function (initial) {                          #A
  nameSetup.fullname = nameSetup.fullname.replace(" ", " " + initial + "
");
};
nameSetup.addInitial('D');                                           #B
console.log(nameSetup.fullname);                                     #C
```

#A Defining a new function inside the global object
#B Invoking the function and sending a parameter
#C The output is "Simon D Holmes"

Working in this way gives you control over your JavaScript, and reduces the chances that your code will give you unpleasant surprises. So remember to declare variables in the appropriate scope and at the correct time, and group them together into objects wherever possible.

2.2.2 Logic flow and loops

Now we're going to take a quick look at best practices for the commonly used patterns of `if` statements and `for` loops. We're going to assume that you are familiar with these to some extent.

CONDITIONAL STATEMENTS – WORKING WITH 'IF'

JavaScript is very helpful with if statements. If you only have one expression within an `if` block you don't have to wrap it in curly braces `{}`. You can even follow it with an `else`. So this snippet is valid JavaScript.

```
var firstname = 'Simon', surname, fullname;
if (firstname === 'Simon')
  surname = 'Holmes';                          #A
else if (firstname === 'Sally')
  surname = 'Panayiotou';                      #A
fullname = firstname + ' ' + surname;
console.log(fullname);
```

#A BAD PRACTICE! Omitting {} around single expression 'if' blocks

Yes, you can do this in JavaScript, but no you shouldn't! Doing this relies on the layout of the code to be readable, which is not ideal. More importantly, what happens if we want to add some extra lines within our `if` blocks? Let's start by giving Sally a middle initial. See the following snippet for how you might logically try this.

```
var firstname = 'Simon', surname, initial = '', fullname;
if (firstname === 'Simon')
  surname = 'Holmes';
else if (firstname === 'Sally')
  initial = 'J';                                          #A
  surname = 'Panayiotou';
fullname = firstname + ' ' + initial + ' ' + surname;
console.log(fullname);                                    #B
```

#A Adding a line into the 'if' block
#B Output is 'Simon Panayiotou'

What went wrong here is that without the block braces only the first expression is considered part of the block, anything following that is outside of the block. So here, if `firstname` is Sally `initial` becomes `J`, but `surname` always becomes Panayiotou.

The following snippet shows the correct way of writing this.

```
var firstname = 'Simon', surname, initial = '', fullname;
if (firstname === 'Simon') {                      #A
```

```
  surname = 'Holmes';
} else if (firstname === 'Sally') {              #A
  initial = 'J';
  surname = 'Panayiotou';
}                                                #A
fullname = firstname + ' ' + initial + ' ' + surname;
console.log(fullname);
```

#A BEST PRACTICE: always use {} to define if statement blocks

By being prescriptive like this we are now seeing what the JavaScript interpreter sees, and reducing the risk of unexpected errors. It is a good aim to always make your code as explicit as possible, don't leave anything open to interpretation. This will help both the quality of your code, and your ability to understand it again when you come back to it after a year of working on other things.

How many '=' symbols to use

In the code snippets here, you'll notice that in each of the `if` statements we use `===` to check for a match. This is a not only best practice but also a great habit to get into.

The `===` operator is much stricter than `==`. `===` will only ever provide a positive match when the two operands are of the same type, e.g. number, string, Boolean. `==` will attempt type coercion to see if the values are similar but just a different type. This can lead to some interesting and unexpected results.

Look at the following snippet for some interesting cases that could easily trip you up.

```
var number = '';
number == 0;                                     #A
number === 0;                                    #B
number = 1;
number == '1';                                   #C
number === '1';                                  #D
```

#A True
#B False
#C True
#D False

There are some situations where this might appear to be useful, but it is far better to be clear and specific about what you consider a positive match, as opposed to what JavaScript interprets as a positive match. If it doesn't matter to your code whether number is a string or a number type, you can simple match one or the other, for example:

```
number === 0 || number === '';
```

The key is to **always use the *exact* operator** `===`. The same goes for the *not equals* operators: you should always use the exact `!==` instead of the loose `!=`.

RUNNING LOOPS – WORKING WITH 'FOR'

The most common method of looping through a number of items is the `for` loop. JavaScript handles this pretty well, but there are still a couple of pitfalls and best practices to be aware of.

Firstly, like we saw with the `if` statement, JavaScript allows you to omit the curly braces `{}` around the block if you only have one expression in it. Hopefully you should know by now that this is a bad idea here, just as it was for the `if` statements. The following snippet shows some valid JavaScript that might not have the results you expect.

```
for (var i = 0; i < 3; i++)
  console.log(i);
  console.log(i*5);               #A

// Output in the console
// 0
// 1
// 2
// 15                                    #A
```

#A Second statement is outside of the loop so only fires once

From the way this is written and laid out you might expect both `console.log` statements to run on each iteration of the loop. For clarity the above snippet should be written like this:

```
for (var i = 0; i < 3; i++) {
  console.log(i);
}
console.log(i*5);
```

I know I keep going on about this, but making sure that your code reads in the same way that JavaScript interprets it really helps you out! Bearing this in mind, and the best practice we learned for declaring variables, we should never really see `var` inside the `for` conditional statement. Updating the previous snippet to meet this best practice gives us the following snippet.

```
var i;                            #A
for (i = 0; i < 3; i++) {
  console.log(i);
}
console.log(i*5);
```

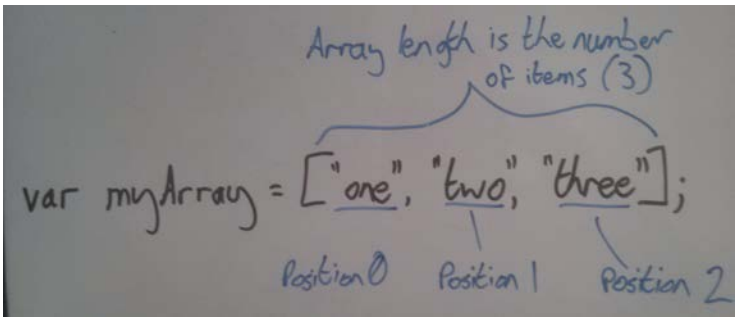
#A Variables should be declared outside of the 'for' statement

A common use for the for loop is to iterate through the contents of an array, so now we'll take a quick look at the best practices and issues to look out for.

USING FOR LOOPS WITH ARRAYS

The key to using for loops with arrays is remembering the arrays are zero-indexed. That is to say, the first object in an array is in position 0. The knock-on effect is that the position of the

last item in the array is one less than the length. This sounds more complicated than it is. A simple array breaks down like this.



The typical code you might see for declaring and array like this and then looping through it is in the following snippet.

```
var i, myArray;
myArray = ["one","two","three"];
for (i = 0; i < myArray.length; i++) {    #A
  console.log(myArray[i]);
}
```

#A Start counting at 0, loop through while the count is less than the length

This works well and will loop through the array correctly, starting at position 0 and going through to the final position 2. Some people prefer to rule out the use of `i++` to auto-increment in their code as it can make code difficult to fathom. Personally I think that for loops are the exception to this rule, and in fact make the code easier to read, rather than adding a manual increment inside the loop itself.

There is one thing we can do to improve the performance of this code. Each time the loop goes round JavaScript is checking the length of `myArray`. This would be quicker if it was just checking against a variable, so a better practice is to declare a variable to hold the length of the array. You can see this in action in the following snippet.

```
var i, myArray, arrayLength;                                    #A
myArray = ["one","two","three"];
for (i = 0, arrayLength = myArray.length; i < arrayCount; i++) { #B
  console.log(myArray[i]);
}
```

#A Declare the arrayLength variable with the other variables
#B Assign the length of the array to arrayLength when setting up the loop

Now we have a new variable `arrayLength` that is given the length of the array to be looped through when the loop is initiated. This means that the script only needs to check the length of the array once, not on every loop.

2.2.3 Formatting practices

The code samples in this book use my personal preference for laying out code. Some of these are necessary for best practice, others because I find they increase readability. If you have different preferences, so long as the code remains correct that's absolutely fine, the important thing is to be consistent.

The main reasons for being concerned with formatting are:

1. Syntactically correct JavaScript
2. Ensuring your code functions correctly when minified
3. Readability for yourself and/or others in your team

Let's start with an easy formatting practice – indentation.

INDENTING CODE

The only real reason for indenting your code is to make it considerably easier for us mere humans to read. JavaScript interpreters don't care about it, and will happily run code without any indentation or line breaks.

Best practice for indentation is to use spaces not tabs, as there is still not a standard for the placement of tabstops. How many spaces you choose is up to you. I personally prefer two spaces. I find that using one space can make it difficult to follow at a glance as the difference isn't all that big. Four spaces can make your code unnecessarily wide, again, in my opinion. I like to balance the readability gains of indentation against the benefits of maximizing the amount of code you can see on screen at one time. Well, there's that and a dislike of horizontal scrolling.

POSITION OF BRACES FOR FUNCTIONS AND BLOCKS

A best practice you should get into is placing the opening bracket of a code block at the end the statement starting the block. What? All of the snippets so far have been written like this so far. The following snippet shows the right way and the wrong way of doing it.

```
var firstname = "Simon", surname;
if (firstname === "Simon") {                    #A
  surname = "Holmes";
  console.log(firstname + " " + surname);
}
if (firstname === "Simon")
{                                               #B
  surname = "Holmes";
  console.log(firstname + " " + surname);
}
```

#A The RIGHT way – opening bracket on the same line as the statement
#B The WRONG way – opening bracket on its own line

99% of the time the second approach won't cause you a problem. The first approach won't cause you a problem 100% of the time. I'll take that over wasting time debugging, how about you?

DON'T BE AFRAID OF WHITESPACE

Adding a bit of whitespace between sets of braces can help readability and won't cause any problems for JavaScript. Again, you've seen this approach in all of the code snippets so far. You can also add or remove white space from between a lot of JavaScript operators. Take a look at the following snippet, showing the same piece of code with and without extra whitespace.

```
var firstname = "Simon", surname;          #A
if (firstname === "Simon") {               #A
  surname = "Holmes";                      #A
  console.log(firstname + " " + surname); #A
}                                          #A
var firstname="Simon",surname;             #B
if(firstname==="Simon"){                   #B
  surname="Holmes";                        #B
  console.log(firstname+" "+surname);      #B
}                                          #B
```

#A JavaScript snippet using whitespace for readability
#B The same snippet with whitespace removed (excluding indentation)

As humans we read using whitespace as the delimiters for words, the way we read code is no different. Yes, you can figure out the second part of the snippet there, as there are many syntactic pointers to act as delimiters, but it is quicker and easier to read and understand the first part. JavaScript interpreters don't notice the whitespace in these places, and if you're concerned about increasing the file size for browser-based code you can always minimize it before pushing it live.

USING THE SEMICOLON CORRECTLY

JavaScript uses the semicolon character to denote the end of statements. It tries to be helpful by making this optional and will inject it's own semicolons at runtime if it deems it necessary. This is not a good thing at all.

When using semicolons to delimit statements you should return to the goal of seeing in the code what the JavaScript interpreter sees, and not let it make any assumptions. I treat semi-colons as not optional, and I'm not at a point where code looks wrong to me if they're not there.

So most lines of your JavaScript will have a semicolon at the end, but not all – that would be too easy! All of the following types of statements should end with a semicolon.

```
var firstname = "Simon", surname;                        #A
surname = "Holmes";                                      #A
console.log(firstname + " " + surname);                  #A
var addSurname = function() {};                          #A
alert("Hello");                                          #A
var nameSetup = { firstname : 'Simon', fullname : ''};   #A
```

#A Use a semicolon at the end of most statements, including these examples

But code blocks should not end with a semicolon. We're talking about blocks of code associated with if, switch, for, while, try, catch and function (when not being assigned to a variable. For example:

```
if (firstname === 'Simon') {
  …
}                                   #A
function addSurname() {
  …
}                                   #A
for (i = 0; i < 3; i++) {
  …
}                                   #A
```

#A No semicolon used at the end of a code block

The rule isn't quite so straightforward as "don't use a semicolon" after curly braces. When assigned a function or object to a variable you *do* have a semicolon after the curly braces. We saw a couple of examples just above, and have been using them throughout the book so far.

```
var addSurname = function() {
  …
};                                  #A
var nameSetup = {
  firstname : 'Simon'
};                                  #A
```

#A Semicolons after curly braces when assigning to a variable

This can take a little while to get used to, but it is worth the effort and it ends up becoming second nature.

WHERE TO PLACE THE COMMAS IN A LIST

When you're defining a long list of variables at the top of a scope, the most common approach to write one variable name per line. This makes it really easy to see at a glance what variables you have set up. The classic placement for the comma separating these is at the end of the line, like in the following snippet.

```
var firstname = 'Simon',    #A
  surname,                  #A
  initial = '',             #A
  fullname;
```

#A Using a comma at the end of each line separating it from the next variable declaration

Personally, this is my preferred approach as I've been using it for about 15 years. Other people prefer to put the comma at the front of each line, giving you the following snippet.

```
var firstname = 'Simon'
  , surname                #A
  , initial = ''           #A
  , fullname;              #A
```

#A Using a comma at the start of each line separating it from the next variable declaration

This is perfectly valid JavaScript, and when minified down to one line will read exactly the same as the first snippet. I've tried to get used to it but I can't. It just looks wrong to me!

There are arguments for and against both approaches, and in the end it comes down to personal preference. My preference is for commas at the end of lines and is what will be used in this book. But I won't hold it against you if you – or your company's coding standards – prefer commas at the start of lines.

TOOLS TO HELP YOU WRITE GOOD JAVASCRIPT

There are a couple of online code quality checkers JSLint and JSHint. These both check the quality and consistency of your code. Even better, most IDEs and good text editors have plugins or extensions for one or the other, so your code can be quality checked as you go. This is very useful for spotting the occasional missed semicolon or comma in the wrong place.

2.3 Getting to know JSON

JSON – JavaScript Object Notation – is a JavaScript-based approach to data exchange. It is much smaller than XML, more flexible, and easier to read. JSON is based on the structure of JavaScript objects, but is actually language independent and can be used to transfer data between all manner of programming languages.

We have used objects already in our sample code, and seeing as JSON is based on JavaScript objects let's start by taking a quick refresher of them.

2.3.1 A recap on JavaScript object literals

In JavaScript everything other than the most simple data types – string, number, Boolean, null and undefined – is an object. This includes arrays and functions; they are actually objects. Object literals are what most people think of as JavaScript objects. These are typically used to store data, but can also contain functions as we have already seen.

LOOKING AT THE CONTENTS OF A JAVASCRIPT OBJECT

A JavaScript object is essentially a collection of key-value pairs, which are the properties of the object. Each key must have a value.

The rules for a key are quite simple:

1. The key must be a string
2. The string must be wrapped in double quotes if it is a JavaScript reserved word or an illegal JavaScript name

The value can be any JavaScript value, including functions, arrays and nested objects. Listing 2.3.1 shows a valid JavaScript object literal based on these rules.

Listing 2.3.1 An example of a JavaScript object literal

```
var nameSetup = {
  firstname: 'Simon',
  fullname: '',
  age: 37,
  married: true,
  "clean-shaven": null,
  addSurname: function(){
    var surname = 'Holmes';
    nameSetup.fullname = nameSetup.firstname + ' ' + surname;
  },
  children: [{
    firstname: 'Erica'
  }, {
    firstname: 'Isobel'
  }]
};
```

ACCESSING THE PROPERTIES OF AN OBJECT LITERAL

The preferred way of accessing properties is using the `.` notation – this is referred to a *dot notation*. Examples of this are:

```
nameSetup.firstname
nameSetup.fullname
```

This can be used to either get or set property values. If a property doesn't exist when you try to *get* it, JavaScript will return undefined. If a property doesn't exist when you try to *set* it, JavaScript will add it to the object and create it for you.

The dot notation can't be used when the key name is a reserved word or an illegal JavaScript name. To access these properties you need to wrap the key string in square braces `[]`. A couple of examples of this are:

```
nameSetup["clean-shaven"]
nameSetup["var"]
```

Again, these references can be used to get or set the values. After that quick recap over object literals, let's take a look at how JSON is related.

2.3.2 The differences with JSON

JSON is based on the notation of JavaScript object literals, but because it is designed to be language independent there are a couple of important differences.

1. All key names and strings must wrapped in double quotes
2. Functions are not a supported data type

These two differences are largely because you don't know what will be interpreting it. Other programming languages will not be able to process JavaScript functions, and will probably have different sets of reserved names and restrictions on names. If you send all names as strings you can bypass this issue.

ALLOWABLE DATA TYPES IN JSON

We can't send functions with JSON, but as it's a data exchange format that's not such a bad thing. The types of data you can send are:

- Strings
- Numbers
- Objects
- Arrays
- Booleans
- The value `null`

Looking at this list and comparing it to the JavaScript object we had in Listing 2.1, if we remove the function property we should be able to convert it to JSON.

FORMATTING JSON DATA

Unlike the JavaScript object we're not assigning the data to a variable, and nor do we need a trailing semicolon. So by wrapping all key names and strings in double quotes – and they do have to be double quotes – we can generate Listing 2.3.2.

Listing 2.3.2 An example of correctly formatted JSON

```
{
  "firstname": "Simon",     #A
  "fullname": "",           #B
  "age": 37,                #C
  "married": true,          #D
  "has-own-hair": null,     #E
  "children": [{            #F
    "firstname": "Erica"    #F
  }, {                      #F
    "firstname": "Isobel"   #F
  }]                        #F
}
```

#A With JSON we can send strings…
#B empty strings…
#C numbers…
#D Boolean values…
#E null…
#F and arrays of other JSON objects

In this listing we have seen some valid JSON. This data can be exchanged be applications and programming languages without issue. It is also easy for the human eye to read and understand pretty quickly.

Sending strings containing double quotes

JSON specifies that all strings must be wrapped in double quotes. So what if your string contains double quotes? The first double quote that an interpreter comes across will be seen as the end delimiter for the string, so will most likely throw an error when the next item is not valid JSON.

The following snippet shows an example of this. There are two double quotes inside the string, which is not valid JSON and will cause errors.

```
"line": "So she said "Hello Simon""
```

The answer to this problem is to escape nested double quotes with the backslash character \. Applying this technique gives us the following snippet.

```
"line": "So she said \"Hello Simon\""
```

This escape character tells JSON interpreters that the following character should not be considered part of the code, it is part of the value and can be ignored.

SHRINKING JSON FOR TRANSPORTING ACROSS THE INTERNET

The spacing and indentation in Listing 2.3.2 is purely to aid human readability. Programming languages don't need it, and we can reduce the amount of information being transmitted if we remove unnecessary whitespace before sending.

The following snippet shows a minimized version of Listing 2.3.2, which is more along the lines of what you'd expect to exchange between applications.

```
{"firstname":"Simon","fullname":"","age":37,"married":true,"has-own-
hair":null,"children":[{"firstname":"Erica"},{"firstname":"Isobel"}]}
```

The content of this is exactly the same as Listing 2.3.2, just much more compact.

2.3.3 Why is JSON so good?

The popularity of JSON as a data exchange format pre-dates the development of Node.js by quite some time. JSON really began to flourish as the ability of browsers to run complex JavaScript increased. Having a data format that was (almost) natively supported was extremely helpful, and made life considerably easier for front-end developers.

The previous preferred data exchange format was XML. In comparison to JSON, XML is harder to read at a glance, much more rigid, and considerably larger to send across

networks. As we've just seen in the JSON examples, JSON doesn't waste much space on syntax. JSON uses the minimum amount of characters required to accurately hold and structure the data, and not a lot more.

When it comes to the MEAN stack, JSON is the ideal format for passing data through the layers of the stack. MongoDB actually stores data as binary JSON (BSON), Node.js and Express can interpret this natively and also push it out to AngularJS, which also uses JSON natively. So every part of the MEAN stack – including the database – uses the same data format, meaning we have no data transformations to worry about.

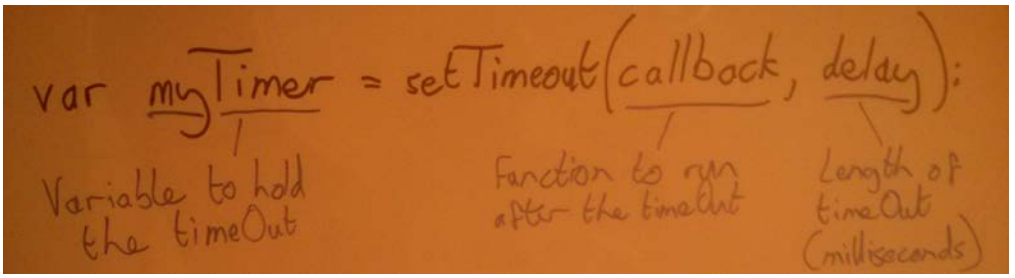
2.4 Understanding JavaScript callbacks

The next aspect of JavaScript programming that we're going to look at is callbacks. These often seem quite confusing or complicated at first, but if we take a look under the hood we'll find that they're fairly straightforward. The chances are that you've already used them.

Callbacks are typically used to run a piece of code after a certain event has happened. Whether this event is a mouse clicking on a link, or data being written to a database or just another piece of code to finish executing isn't important, as it could be just about anything. A callback function itself is typically an *anonymous function* – a function declared without a name – that is passed directly into receiving function as a parameter. Don't worry if this just seems like jargon right now, we'll look at code examples soon and you'll see how easy it actually is!

2.4.1 Use setTimeout to run code later

Most of the time we use callbacks to run code after something has happened. For getting to grips with the concept we can use a function that is built in to JavaScript – `setTimeout`. You may have already used it. In a nutshell `setTimeout` will run a callback function after the number of milliseconds that you declare. The basic construct for using it is this:



Cancelling a setTimeout

If a setTimeout declaration has been assigned to a variable, we can use that variable to clear the time out, and stop it from completing. Assuming that is has not already completed. This is done by using the `clearTimeout()` function, and works like so:

```
var waitForIt = setTimeout(function(){
  console.log("My name is Simon Holmes");
}, 2000);
clearTimeout(waitForIt);
```

This snippet wouldn't output anything to the log, as the waitForIt timer has been cleared before it has had chance to complete.

First of all, setTimeout is declared inside a variable so that we can access it again to cancel it should we want to. As we talked about at the opening of the chapter, a callback is typically an un-named anonymous function. So if we wanted to log our name to the JavaScript console after 2 seconds we could use this snippet:

```
var waitForIt = setTimeout(function(){
  console.log("My name is Simon");
}, 2000);
```

NOTE callbacks are asynchronous. This means that they will run when required, not necessarily in the order in which they appear in your code.

So keeping in mind this asynchronous nature, what would you expect the output of the following code snippet to be?

```
console.log("Hello, what's your name?");
var waitForIt = setTimeout(function(){
  console.log("My name is Simon");
}, 2000);
console.log("Nice to meet you Simon");
```

Reading the code from top to bottom, the console log statements look to make sense. But because the setTimeout callback is asynchronous it doesn't hold up the processing of code, so you actually end up with this.

```
Hello, what's your name?
Nice to meet you Simon
My name is Simon
```

As a conversation this clearly doesn't flow properly. In your code having the correct flow is essential, otherwise your application will quickly fall apart.

Seeing as this asynchronous approach is so fundamental to working with Node, let's look into it a little deeper.

2.4.2 Asynchronous code

Before we look at some more code, let's start by reminding ourselves of the bank teller analogy we saw in Chapter 1. Figure 3.1 shows how a bank teller can deal with multiple requests by passing on any time-consuming tasks to other people.

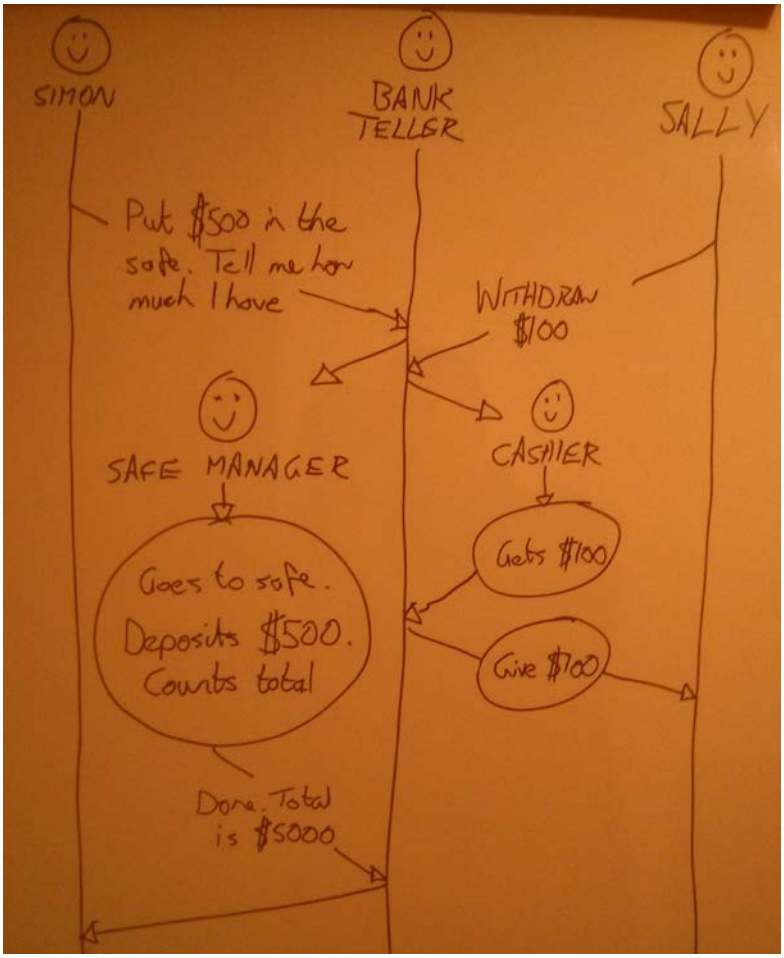


Figure 3.1 Handling multiple requests

The bank teller is able to respond to Sally's request because it has passed responsibility for Simon's request to the Safe Manager. The Teller isn't interested in how the Safe Manager does what he does, or how long it takes. This is an asynchronous approach.

We can mimic this in JavaScript, using the setTimeout function to demonstrate the asynchronous approach. All we need are some console.log statements to demonstrate the bank teller's activity, and a couple of timeouts to represent the delegated tasks. You can see this in the following snippet, where we assume that Simon's request will take three seconds (3000 milliseconds) and Sally's will take 1 second.

```
console.log("Taking Simon's request");                          #1
var requestA = setTimeout(function(){
  console.log("Simon: money's in the safe, you have $5000");
}, 3000);

console.log("Taking Sally's request");                          #2
var requestB = setTimeout(function(){
  console.log("Sally: Here's your $100");
}, 1000);

console.log("Free to take another request");                    #3

// ** console.log responses, in order **
// Taking Simon's request
// Taking Sally's request
// Free to take another request
// Sally: Here's your $100                                      #A
// Simon: money's in the safe, you have $5000                   #B
```

#1 Take the first request
#2 Take the second request
#3 Ready for another request
#A Sally's response appears after 1 second
#B Simon's response appears after another 2 seconds

This code has three distinct blocks: #1 Taking the first request from Simon and sending it away; #2 taking the second request from Sally and sending it way; #3 ready to take another request. If this were synchronous code like you'd see in PHP or .NET, we would first deal with Simon's request in entirety before even taking Sally's request three seconds later.

As we are using an asynchronous approach the code doesn't have to wait for one of the requests to complete before taking another one. You can run this code snippet in your browser to see how it works – either put it into a HTML page and run it, or enter it directly into JavaScript console.

You should hopefully see how this mimics the scenario we talked through as we kicked off this section. Simon's request was first in, but as it took some time to complete, the response didn't come back immediately. While somebody was dealing with Simon's request, Sally's request was taken. While Sally's request was being dealt with we became open again to take another request. As Sally's request took less time to complete she got her response first, while Simon had to wait a bit longer for his response. Neither Sally nor Simon were held up by each other.

Now let's go one step further by taking a look at what might be happening *inside* the `setTimeout` function.

2.4.3 Running a callback function

We're not actually going to look at the source code of `setTimeout` here, but rather a skeleton of how we can create a function that uses a callback. We will simply declare a new function called `setTimeout` that accepts the parameters `callback` and `delay`; the names aren't important, they can be anything you want. The following snippet demonstrates this (note that you will not be able to run this in a JavaScript console).

```
var setTimeout = function(callback, delay){
  …                                                    #A
  …
  callback();                                          #1
};

var requestB = setTimeout(function(){                  #2
  console.log("Sally: Here's your $100");              #2
}, 1000);                                              #2
```

#A Delay processing for specified number of milliseconds
#1 Run the callback function
#2 Send the anonymous function & delay

The `callback` parameter is expected to be a function, which can be invoked at a specific point in the `setTimeout` function `#1`. In this case we are passing it a simple anonymous function `#2` that will write a message to the console log. So when the `setTimeout` function deems it appropriate, it will invoke the callback and the message will be logged to the console. That's not so difficult, is it?

If JavaScript is your first programming language you'll have no idea how weird this concept of passing anonymous functions around looks to those coming in from different backgrounds. But the ability to operate like this is one of JavaScript's great strengths.

Typically you won't generally look inside the functions running the callbacks, whether it's `setTimeout`, jQuery's `ready` or Node's `createServer`. The documentation for all of these will tell you what the expected parameters are, and also what parameters it may return.

Why setTimeout is unusual

The setTimeout function is unusual in that you specify a delay after which the callback will fire. In a more typical use case, the function itself will decide when the callback should be triggered. In jQuery's `ready` method this is when jQuery says the DOM has loaded; in a save operation in Node this is when the data is saved to the database and a confirmation returned.

CALLBACK SCOPE

A particularly interesting aspect of passing anonymous functions around like this, is that the callback inherits the scope of the function it is passed into. As far as setTimeout is concerned it is declaring and running the callback function inside itself. In terms of code it is akin to doing this:

```
var setTimeout = function(callback, delay){
  var myCallback = function() {                                    #A
    console.log("Sally: Here's your $100");
  }();                                                             #B
};
```

#A Define a function inside setTimeout
#B Use the () at the end of the function to immediately invoke it

If you remember from when we looked at functions and scope in Chapter 2, a nested function – `myCallback` in this instance – has it's own local scope, but also inherits the scope of the parent function, `setTimeout`.

Let's return to the idea of 'scope circles' and look at this visually in Figure 3.2.

Figure 3.2 Callback scope nested inside the function that runs it

Here we can see that as the callback runs inside the scope of `setTimeout` it gains access to `setTimeout`'s scope, and isn't limited to the global scope in which it was actually written.

We can further demonstrate this and embellish on the same example by setting a variable inside the `setTimeout` scope and accessing it from the callback. Let's take the dollar amount we've been sending to the console log and set that as a variable inside the `setTimeout` function. In our scope circles that should give us something like figure 3.3.

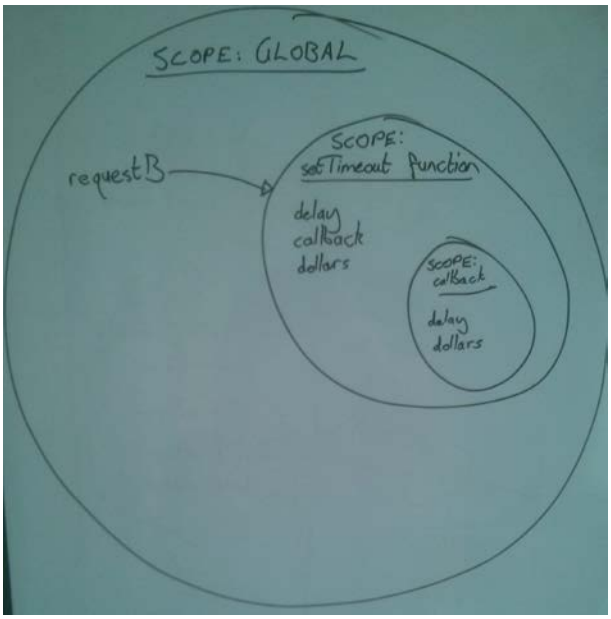


Figure 3.3 Setting a variable specifically for the callback to use

Doing the same thing in code would look like the following snippet.

```
var setTimeout = function(callback, delay){
  var dollars = 100;                                          #A
  …
  callback();
};

var requestB = setTimeout(function(){
  console.log("Sally: Here's your $" + dollars);              #B
}, 1000);
```

#A Declare a variable in the function scope
#B Use the variable in the callback

This snippet will output the same message to the console that we have already seen. The approach works because even though the callback function is declared outside the

`setTimeout` function – and so in a different scope – the callback function itself is passed into the `setTimeout` function where it is actually run, inheriting the scope.

It is important that you understand this approach, as the vast majority of Node.js code examples on the Internet use asynchronous callbacks in this way. However, there are a couple of potential problems with this approach, particularly when your codebase gets larger and more complex. Firstly, passing around functions and having implied scope inheritance can make the code hard to read and follow. Secondly, it makes it difficult to run tests on the code. We will be looking at unit testing our code later in the book, but in a nutshell the idea is that every piece of code can be tested separately with repeatable and expected results.

Let's look at a way that can help us achieve this, using named callbacks.

2.4.4 Named callbacks

Named callbacks differ from inline callbacks in one fundamental way. Instead of putting the code you want to run directly into the callback, you put the code you want to run inside a defined function, and then call this function *from* the callback. So rather than passing the code to run you are passing a *reference* to the code to run. Sticking with our ongoing example, let's add a new function `onCompletion` that we'll invoke from the callback. Figure 3.4 shows how this looks in our scope circles.

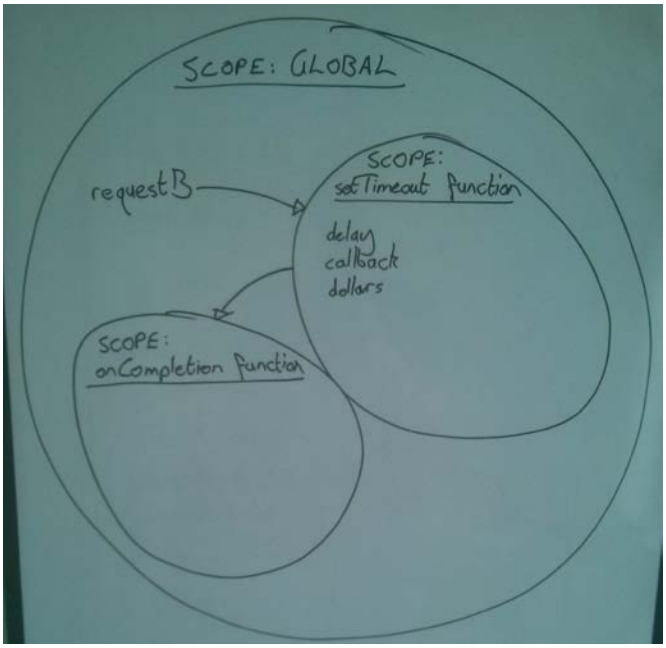


Figure 3.4 The change in scope when using a named callback

You may have noticed from the figure, that moving the code out into a named callback has an important knock-on effect. The named function is declared and defined in a different scope, the global scope in our example. This means that we lose the benefits of scope inheritance, so our new function `onCompletion` has no idea what the variable `dollars` should be – it will be `undefined`. The following snippet shows how to declare and invoke a named callback, putting into code what we see in Figure 3.4.

```
var setTimeout = function(callback, delay){
  var dollars = 100;
  …
  callback();
};

var onCompletion = function(){                          #1
  console.log("Sally: Here's your $" + dollars);
};

var requestB = setTimeout(function(){
  onCompletion();                                       #2
}, 1000);
```

#1 Declaring a named function in a distinct scope
#2 Calling the named function from inside the callback

The named function #1 now exists as an entity in it's own right, creating it's own scope. Notice how we use the same callback function construct, but now we send a call to the named function #2 instead of passing the code through. Wrapping the named function inside an anonymous function may seem a little odd, but we'll get to the reason for this in just a moment.

SOLVING THE SCOPE PROBLEM

The previous snippet reiterated the `undefined` variable problem we have with the named callback. Fortunately the way to combat this is quite straightforward. All we need to do is send a variable from one scope to another, so we can simply pass the parameters we need into the named function. Figure 3.5 shows how this looks in our scope circles.

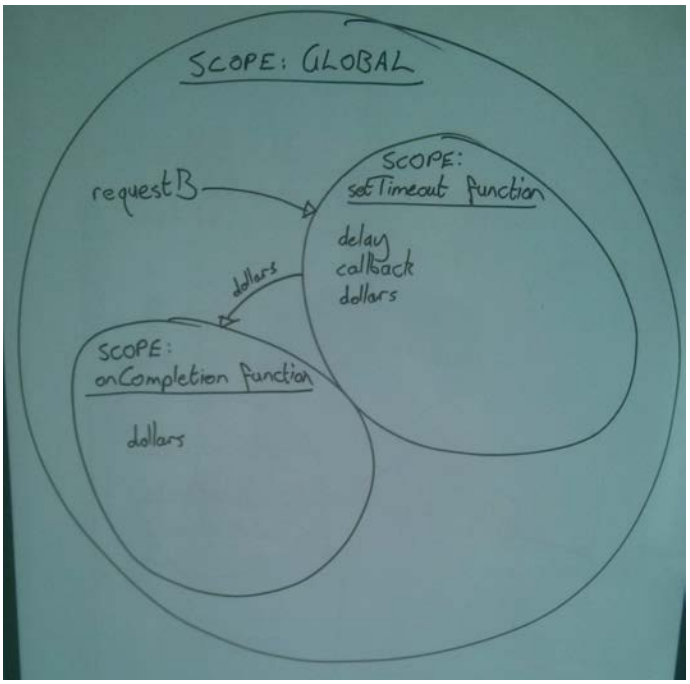


Figure 3.5 Passing the required parameter into the new function scope

So we just need to pass the variable `dollars` from the callback to the new function. This is done in exactly the way you might expect; take a look at the following snippet:

```
var setTimeout = function(callback, delay){
  var dollars = 100;
  …
  callback();
};

var onCompletion = function(dollars){                          #A
  console.log("Sally: Here's your $" + dollars);
};

var requestB = setTimeout(function(){
  onCompletion(dollars);                                       #B
}, 1000);
```

#A Named function accepting the parameter
#B Calling the named function, passing the parameter

Passing the named function through inside the anonymous callback construct like this enables us to capture anything we need from the `setTimeout` scope and explicitly pass it to

the `onCompletion` function. Remember that you normally won't have access to the code inside the parent function that invokes the callback, and that the callback will often be invoked with a fixed set of parameters (or none at all, like with `setTimeout`). So anything we need to add must be added inside the callback.

BETTER FOR READING AND TESTING

By defining a named function in this way, it makes the scope and code of the function easier to comprehend at a glance, especially if you name your functions well. With a small, simple example like this, you could well think that we've made the flow harder to understand, by moving the code into it's own function. And you could well have a point! However, when the code becomes more complex, and you have multiple lines of code inside multiple nested callbacks you will most definitely see the advantage of doing it this way.

Another advantage of being able to easily see what the `onCompletion` function should do, and what parameters it expects and requires to work, is that we can test it more easily. For example, we can now say "when the function `onCompletion` is passed a number of dollars it should output a message to the console including this number". This is a pretty simple case, but hopefully you can see the value of this.

That brings us to the end of 'understanding callbacks' from a code perspective. Now that we've got a good idea of how callbacks are defined and used, let's look at Node.js and see why they're so useful.

2.4.5 Callbacks in Node.js

In the browser a lot of the events are based around user interaction, waiting for things to happen outside of what the code can control. The concept of waiting for external things to happen is similar on the server side. The difference on the server side is that the events are more focused around other things happening on the server, or indeed a different server. Whereas in the browser the code waits for events such as a mouse click or form submit, the server-side code waits for events such as reading a file from the file system or saving data to a database.

The big difference is this. In the browser, it is generally an individual user who initiates the event, and it is only that user who is waiting for a response. On the server side, the central code generally initiates the event and waits for a response. As we saw in Chapter 1, there is only a single thread running in Node, meaning that if the central code has to stop and wait for a response then every visitor to the site get's held up. This is not a good thing! And this is why it is important to understand callbacks, as Node uses callbacks to delegate the waiting to other processes, making it asynchronous.

Let's take a look at an example of using callbacks in Node.

A NODE.JS CALLBACK

Using a callback in Node isn't really any different to using it in the browser. If we want to save some data, we don't want the main Node process doing this, just like we didn't want the Bank Teller going with the Safe Manager and waiting for the response. So we want to use

an asynchronous function with a callback. All database drivers for Node provide this ability; we will get into the specifics about how to create and save data later in the book, so for now we'll just us a simplified example. The following snippet shows an example of asynchronously saving data `mySafe` and outputting a confirmation to the console when the database finishes and returns a response.

```
mySafe.save(
  function (err, savedData) {
    console.log("Data saved: " + savedData);
  }
);
```

Here the save function expects a callback function that can accept two parameters, an error object `err` and the data returned from the database following the save `savedData`. There is normally a bit more to it than this, but the basic construct is fairly simple.

RUNNING CALLBACKS ONE AFTER ANOTHER

So that's great. We get the idea of running a callback, but what do we do if we want to run another asynchronous operation when the callback is finished? Returning to our banking metaphor, say for example we want to get a total value from all of Simon's accounts after the deposit is made to the safe. Simon doesn't need to know that there are multiple steps and multiple people involved, and the Bank Teller doesn't need to know until everything is complete. So we're looking to create a flow like that in Figure 3.6.

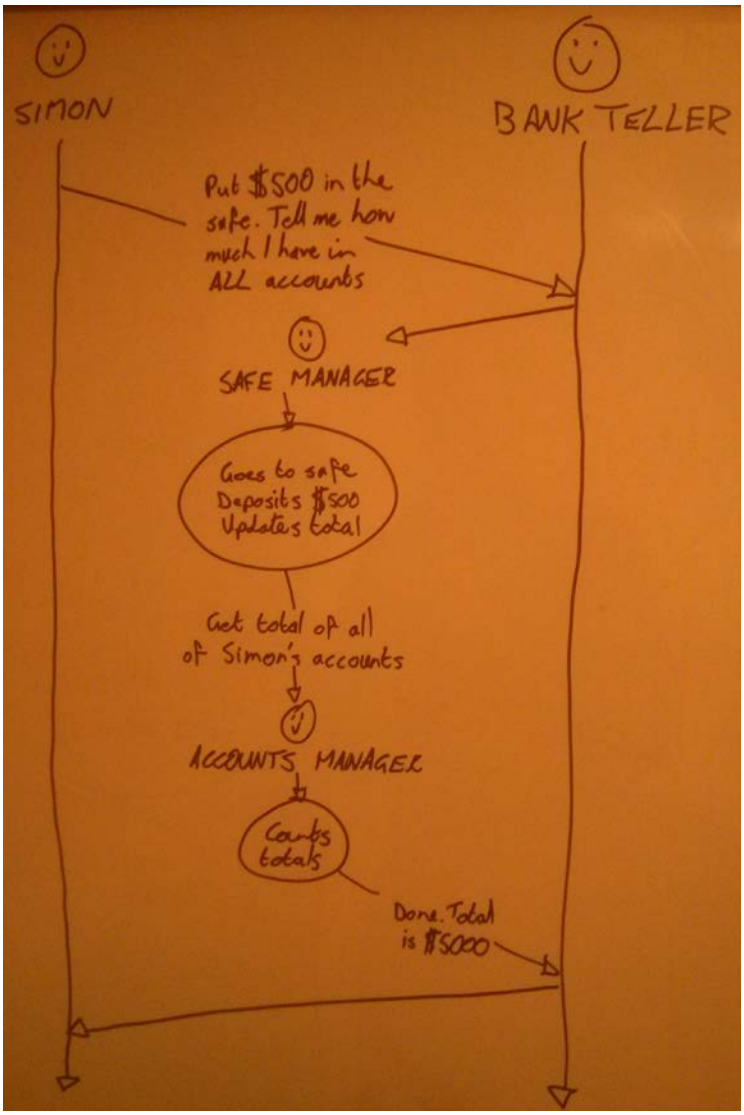


Figure 3.6 The required flow when using two asynchronous operations, one after another.

Now that it's clear in our mind that this will require two operations, we can see that this is going to need another asynchronous call to the database. We know from hat we've already learned that we can't just put it in the code after the save function, like in the following snippet.

```
mySafe.save(
  function (err, savedData) {
    console.log("Data saved: " + savedData);
  }
);
myAccounts.findTotal(                                    #A
  function (err, accountsData) {
    console.log("Your total: " + accountsData);
  }
);
// ** console.log responses, in probable order **
// Your total: 4500
// Data saved: {dataObject}
```

#A This second function will fire before the save function has finished, so the returned accountsData will likely be incorrect

That's not going to work because the `myAccounts.findTotal` function will run immediately, rather than when the `mySafe.save` function has finished. This would mean that the return value is likely to be incorrect, as it won't take into account the value being added to the safe. So we need to ensure that the second operation runs when we know the first one has finished. The solution is simple: we need to invoke the second function from inside the first callback. This is known as nesting the callbacks.

Nested callbacks are used to run asynchronous functions one after another. So let's put the second function inside the callback from the first to see an example.

```
mySafe.save(
  function (err, savedData) {
    console.log("Data saved: " + savedData);
    myAccounts.findTotal(                                #A
      function (err, accountsData) {
        console.log("Your total: " + accountsData.total);
      }
    );
  }
);
// ** console.log responses, in order **
// Data saved: {dataObject}
// Your total: 5000
```

#A The second asynchronous operation 'nested' inside the callback of the first

Now we can be sure that the `myAccounts.findTotal` function will run at the appropriate time, which in turns means that we can predict the response.

This ability is very important. Node.js is inherently asynchronous, jumping from request to request, from site visitor to site visitor. But sometimes we need to do things in a sequential manner, one thing after another – nesting callbacks gives us a good way of doing this, using native JavaScript.

The downside of nested callbacks is the complexity. You can probably already see that with just one level of nesting that the code is already a bit harder to read, and following the

sequential flow takes a bit more mental effort. This problem is multiplied when the code gets more complex and you end up with multiple levels of nested callback. The problem is so great that it has become known as **callback hell**. This is actually a reason why some people think that Node.js is particularly hard to learn and difficult to maintain, and use it as an argument against the technology. In fairness, many code samples you can find online do suffer from this problem, which doesn't do much to combat this opinion. It is easy to end up in callback hell when developing Node.js, but it's also easy to avoid if you start off in the right way.

We have actually already looked at the solution to callback hell, and that is using named callbacks. So let's take a look at how named callbacks helps us here.

Promises: an alternative approach

An alternative to using nested callbacks or named callbacks is to use promises. Promises address head on the conflict of Node.js's asynchronous nature against the sequential requirements of most applications.

There's a bit more to it than this, but using promises can simplify your control flow down to something along these lines:

```
myData.save()
  .then(myAccounts.findTotal)
  .then(doSomethingElseAsynchronous)
```

However, promises require the use of third party libraries, which we're not going to explore in this book. These libraries are great, it's not a question of quality. Callbacks used one way or another are still the most common and are what you'll find in most public repositories, so it's important that we focus on solidifying our understanding of them.

When you're comfortable with the callback approach feel free to take a look at some of the promises modules available for Node.js.

USING NAMED CALLBACKS TO AVOID "CALLBACK HELL"

Named callbacks can help avoid nested callback hell as they can be used to separate out each step into a distinct piece of code or functionality. We humans tend to find this easier to read and understand.

Thinking back, to use a named callback we need to take the content of a callback function and declare it as a separate function. In our nested callback example we had two callbacks, so we're going to need to new functions, one for when the `mySafe.save` operation has completed, and one for when the `myAccounts.findTotal` operation has completed. If call these `onSave` and `onFindTotal` respectively we can create some code like the following snippet.

```
mySafe.save(
  function (err, savedData) {
    onSave(err, savedData);                                    #A
  }
);

var onSave = function (err, savedData) {
  console.log("Data saved: " + savedData);
  myAccounts.findTotal(                                        #B
    function (err, accountsData) {
      onFindTotal(err, accountsData);                          #C
    }
  );
};

var onFindTotal = function (err, accountsData) {
  console.log("Your total: " + accountsData.total);
};
```

#A Invoke the first named function from the mySafe.save operation
#B Start the second asynchronous operation from within the first named callback
#C Invoke the second named function

Now that each piece of functionality is separated out into a separate function, it is easier to look at each part in isolation and understand what it is doing. You can see what parameters it expects and what the outcomes should be. In reality the outcomes are likely to more complex than simple console.log statements, but you get the idea! You can also follow the flow relatively easily, and see the scope of each function.

So by using named callbacks you can reduce the perceived complexity of Node.js, and also make your code easier to read and maintain. A very important second advantage is that individual functions are much better suited to unit testing – each part has defined inputs and outputs, with expected and repeatable behavior.

2.5 Writing modular JavaScript

A couple of days ago someone tweeted a great quote:

> The secret to writing large apps in JavaScript is not to write large apps. Write many small apps that can talk to each other.
>
> Source: anon

This quote makes great sense in a number of ways. Many applications share several common features, such as user login and management, comments, reviews and so on. The easier it is for you to take a feature from one application you've written and drop it into another the more efficient you will be. Particularly as you will already have – hopefully – tested the feature in isolation so you know that it works.

This is where modular JavaScript comes in. JavaScript applications do not have to be in one never-ending file with functions, logic and global variables flying loose all over the place. You can contain functionality within enclosed modules.

2.5.1 Closures

A closure essentially gives you access to the variables set in a function after the function has completed and returned. This then offers you a way of avoiding pushing variables into the global scope. It also offers a degree of protection to the variable and its value, as you cannot simply overwrite it, like you could do with a global variable.

Sound a bit weird? Okay, let's take a look at an example. The following snippet demonstrates how you can send a value to a function and then later retrieve it.

```
var user = {};

var setAge = function (myAge) {
  return {                          #A
    getAge: function () {           #A
      return myAge;                 #A
    }                               #A
  };
};

user.age = setAge(30);              #B
console.log(user.age);              #C
console.log(user.age.getAge());     #D
```

#A Return a function that returns the parameter
#B Invoke the function and assign to the 'age' property of 'user'
#C Output - Object {getAge: function}
#D Retrieve the value using the getAge() method. Output - 30

Okay, so as well as sounding a bit weird, it possibly also looks a bit weird! But here's what is happening. The `getAge` function is returned as a method of the `setAge` function. The `getAge` method has access to the scope in which it was created. So `getAge`, and `getAge` alone has access to the `myAge` parameter. As we saw earlier in the chapter, when a function is created it also creates it's own scope. Nothing outside of this function has access to the scope.

So `myAge` is also not a one-off shared variable. We can call the function again – creating a second new function scope – to set (and get) the age of a second user. We could happily run the following snippet after the one we've just looked at, creating a second user and giving them a different age.

```
var usertwo = {};
usertwo.age = setAge(35);                #A
console.log(usertwo.age.getAge());       #B
console.log(user.age.getAge());          #C
```

#A Assign the setAge function to a new user with a different age
#B Outputs usertwo's age: 35
#C Output the original user's age: 30

Each user has a different age that is not aware of – nor impacted by – the other. The closure protects the value from outside interference. The important takeaway point here is: **the returned method has access to the scope in which it was created**.

This closure approach is a great start, but has evolved into more useful patterns. We'll start by looking at the module pattern.

2.5.2 Module pattern

The module pattern takes the closure concept and extends it, typically wrapping up a collection of code, functions and functionality into a *module*. The idea behind it is that it is self-contained, and only uses data explicitly passed into it, and only reveals data that is directly asked for.

Immediately-Invoked Function Expression (IIFE)

The module pattern makes use of what is known as the Immediately-Invoked Function Expression, or IIFE. The functions we have been using up until now have been function declarations, creating functions that we can call upon later in the code. The IIFE create a function expression and immediately invokes it, typically returning some values and/or methods.

The syntax for an IIFE is to wrap the function in parentheses, and immediately invoke it using another pair of parentheses – see the bold sections of this snippet.

```
var myFunc = (function () {          #1
  return {
    myString: "a string"
  };
})();
console.log(myFunc.myString);        #2
```

#1 Assigning the IIFE to a variable
#2 Access the returned methods as properties of the variable

This is the typical use, but not the only one. The IIFE has been assigned to a variable #1. When you do this, the returned methods from the function become properties of the variable #2.

This is made possible by using an Immediately-Invoked Function Expression – see the sidebar for a bit more info on these. Like the basic closure, the module pattern returns functions and variables as properties of the variable it is assigned to. Unlike the basic closure, the module pattern does not have to be manually initiated; the module immediately calls itself as soon as it has been defined.

The following snippet shows a small but usable example of the module pattern.

```
var user = {firstname: "Simon"};
var userAge = (function () {       #A
  var myAge;                       #B
  return {
    setAge: function (initAge) {   #C
      myAge = initAge;             #C
    },                             #C
    getAge: function () {          #D
      return myAge;                #D
    }                              #D
  };
})();

userAge.setAge(30);                #E
user.age = userAge.getAge();       #E
console.log(user.age);             #F
```

#A Assign the module to a variable
#B Define a variable in the module scope
#C Define a method to be returned that can take a parameter and modify the module variable
#D Define a method to be returned that access the module variable
#E Call the methods set and get the module variable
#F Output is 30

In this example the `myAge` variable exists within the scope of the module and is never directly exposed to the outside. You can only interact with the `myAge` variable in the ways defined by the exposed methods. In the previous snippet we just get and set, but it is of course possible to modify. We can add a `happyBirthday` method to the `userAge` module that will increase the value of `myAge` by one, and return the new value. See the following snippet, with the new parts in bold.

```
var user = {firstname: "Simon"};
var userAge = (function () {
  var myAge;
  return {
    setAge: function (initAge) {
      myAge = initAge;
    },
    getAge: function () {
      return myAge;
    },
    happyBirthday: function () {          #A
      myAge = myAge + 1;                  #A
      return myAge;                       #A
    }                                     #A
  };
})();

userAge.setAge(30);
user.age = userAge.getAge();
console.log(user.age);
user.age = userAge.happyBirthday();       #B
console.log(user.age);                    #C
```

```
user.age = userAge.getAge();
console.log(user.age);                    #C
```

#A The new method to increment 'myAge' by 1 and return the new value
#B Call the new method and assign it to 'user.age'
#C Output is 31

The new `happyBirthday` method increments the `myAge` value by one and returns the new value. This is possible because the `myAge` variable exists in the scope of the module function, as does the returned `happyBirthday` function. So the new value of `myAge` continues to persist inside the module scope.

2.5.3 Revealing module pattern

What we've just been looking at in the module pattern is heading very close to the revealing module pattern. The revealing module pattern is essentially just some syntax sugarcoating on the module pattern. The aim is to make it really obvious what is exposed as public, and what remains as private to the module.

THE UNDERSCORE VARIABLE NAMING CONVENTION

The first aspect of this approach is really just a convention, there is no strict need to do this, but it makes it easier for you and others to maintain the code. Inside the module, variables that are to be kept private should have an underscore prefix, for example:

```
var _privateVariable;      #A
var publicVariable;        #B
```

#A This variable shouldn't be exposed outside the module
#B This variable will be exposed outside the module

All the underscore does is signify to you when reading the code whether or not a variable will be directly exposed through the return statement. You can ignore this convention as it isn't strictly required, but following it will make your life easier.

TAKE DECLARATIONS OUT OF THE RETURN STATEMENT

This is also a stylistic convention, but is again one that helps you and others understand your code when you come back to it after break. When you use this approach, the `return` statement just contains a list of the functions that you are returning, without any of the actual code. The code is all declared in functions above the `return` statement, although of course within the same module. The following snippet shows an example of this.

```
var userAge = (function () {
  var _myAge;
  var setAge = function (initAge) {       #A
      _myAge = initAge;                   #A
  };                                      #A
  return {
    setAge: setAge                        #B
  };
})();
```

#A The setAge function has been moved outside of the return statement
#B The return statement now references the setAge function and contains no code

You can't really see the benefit of this approach in such a small example. We will look at a longer example shortly which will get us part of the way there, but you'll really see the benefits when you have a module running into several hundred lines of code. Just like gathering all of the variables together at the top of the scope makes it really obvious which variables are being used, taking the code out of the `return` statement makes is really obvious at a glance which functions are being exposed. If you had a dozen or so functions being returned, each with a dozen or more lines of code, the chances are you're not going to be able to see the entire `return` statement on one screen of code without scrolling.

What is important in the `return` statement, and what you'll be looking for, is which methods are being exposed. In the context of the `return` statement you are not really interested in the inner workings of each method. So separating your code out like this makes sense, and sets you up for having really great maintainable – and understandable – code.

A FULL EXAMPLE OF THE PATTERN

So now let's take a look at a larger example of the pattern, using the `userAge` module that we've been working with. Listing 2.5.1 shows an example of the revealing module pattern, using the underscore convention, and removing code from the `return` statement.

Listing 2.5.1 Revealing module pattern example

```
var user = {};
var userAge = (function () {
  var _myAge;                              #1
  var setAge = function (initAge) {
    _myAge = initAge;
  };

  var getAge = function () {
    return _myAge;
  };

  var _addYear = function () {             #2
    _myAge = _myAge + 1;                   #2
  };                                       #3

  var happyBirthday = function () {
    _addYear();                            #3
    return _myAge;
  };

  return {
    setAge: setAge,                        #4
    getAge: getAge,                        #4
    happyBirthday: happyBirthday           #4
  };
})();
```

```
userAge.setAge(30);
user.age = userAge.getAge();
user.age = userAge.happyBirthday();       #A
```

#1 Has an underscore as it is never directly exposed outside of the module
#2 A private function that isn't exposed …
#3 … can be called by a public function that is exposed
#4 The return statement acts as a reference for the exposed methods
#A user.age and _myAge are now 31

Listing 2.5.1 demonstrates a few interesting things. Firstly notice that the variable `_myAge` #1 now has an underscore. This is because the variable itself is never exposed outside of the module. The value of the variable is returned by various methods, but the variable itself remains private to the module.

As well as private variables you can also have private functions. These are also denoted by a leading underscore, as we can see with `_addYear` #2 in the listing. Private functions can easily be called by public methods #3, the underscore again denoting that it is not itself exposed.

The return statement #4 is kept nice and simple, and is now just an at-a-glance reference to the methods being exposed by this module.

Strictly speaking, the order of the functions inside the module isn't important, so long as they are above the return statement. Anything below the return statement will never run. When writing large modules you may find it easier to group together related functions. If it suits what you are doing you could also create a nested module, or even a separate module with public method exposed to the first module so that they can talk to each other.

Remember the quote from the beginning of this section:

> The secret to writing large apps in JavaScript is not to write large apps.
> Write many small apps that can talk to each other.
>
> Source: anon

This applies not only to large-scale apps, but also to modules and functions. If you can keep your modules and functions small and to the point you are on your way to writing great code.

2.6 Summary

JavaScript is a very forgiving language, which makes it very easy to get started, but also very easy to pick up bad habits. If you make a little mistake in your code, JavaScript will sometimes think "well I think you meant to do this, so that's what I'll go with". Sometimes it's right, and sometime it is wrong. This is not really acceptable for good code, so it is important to be specific about what your code should do, and that you try to write your code in the way the JavaScript interpreter sees the code.

A key to understanding the power of JavaScript is to understand scope. There is the global scope, and there is function scope. There are no other types of scope in JavaScript.

You really want to try and avoid using the global scope as much as possible, and when you do use it, try to do it in a clean and contained way. Scope inheritance cascades down from the global scope, so can be difficult to maintain if you're not careful.

JSON is born of JavaScript but is not JavaScript; it is a language-independent data exchange format. JSON contains no JavaScript code, and can quite happily be passed between a PHP server and a .NET server – JavaScript is not required to interpret JSON.

Callbacks are vital to running successful Node.js applications, as they allow the central process to effectively delegate tasks that could hold it up. To put it another way, callbacks enable us to use sequential synchronous operations in an asynchronous environment. But callbacks are not without their problems. It is very easy to end up in "callback hell", having multiple nested callbacks with overlapping inherited scopes making your code very hard to read, test, debug and maintain. Fortunately we can use named callbacks to address this problem on all levels, so long as we remember that named callbacks do not inherit scope like their inline anonymous counterparts.

Closures and module patterns provide ways to write code that is self-contained and reusable between projects. A closure enables you to define a set of functions and variables within its own distinct scope, which you can come back to and interact with through the exposed methods. This leads us down towards the revealing module pattern, which is convention driven to draw specific lines between what is private and what is public. Modules are perfect for writing self-contained pieces of code that can interact well with other code, not tripping up over any scope clashes.

3

Creating and setting up a MEAN project

This chapter covers

- Creating and configuring Express projects
- Some best practices for app development
- Setting up an MVC environment
- Adding Twitter Bootstrap for layout
- Publishing to a live URL, using Git and Heroku

Now we're really ready to get going, and in this chapter we're going to get stuck into building our application. You remember from Chapter 1 that through this book we're going to build an application called Loc8r. This is going to be a location-aware web application that will display listings near to you, and invite people to login and leave reviews.

In the MEAN stack Express is the Node.js web application framework. Together Node.js and Express underpin the entire stack, so let's start here.

Before you can really do anything we'll make sure that you have everything you need installed on your machine. When that's all done we'll look at creating new Express projects from the command line, and the various options you can specify at this point. During this step we'll also cover how you manage dependencies and versions within an Express application.

Express is great, but you can make it better – and get to know it better – by tinkering a little and changing some things around. I'm a fan of having a goal in mind before setting off, so we'll take a brief pause to look at some best practices of development and how we're going to approach putting the application together. This will lead us into a discussion of the

MVC architecture. Here is where you will get under the hood of Express a little, and see what it's doing by modifying it to have a very clear MVC setup.

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

3.1 Installing the pieces

Before getting started with this chapter, you'll need to have five key things installed on your development machine:

- Node.js and npm
- Express (installed globally)
- Git
- Heroku
- New Command Line Interface or Terminal

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

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

Depending on your operating system you may need to install a new Command Line Interface or Terminal. See Appendix B to find out if this applies to you or not.

> **NOTE:** Throughout this book I will often refer to the Command Line Interface as *terminal*. So when I say "run this command in terminal" simply run it in whichever Command Line Interface you are using. When terminal commands are included as code snippets throughout this book, they will start with a $. You shouldn't type this into the terminal; it is simply there to denote that this is a command line statement. For example, using the echo command: `$ echo 'Welcome to Getting MEAN'`

As soon as you are all set up, let's get started by creating a new Express project.

3.2 Create an Express project

All journeys must have a starting point, and the starting point for building a MEAN application is to create a new Express project.

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

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

Each of these commands should output a version number to the terminal. If one of them fails, head back to Appendix A to install again.

Assuming all is good, start by creating a new folder on your machine called *Loc8r*. This can be on your desktop, your documents, in a Dropbox folder – it doesn't really matter, so long as you have full read and write access rights to the folder. I personally do a lot of my MEAN development in Dropbox folders so that it immediately backed up and accessible on any of my machines. If you're in a corporate environment this may well not be suitable for you, so create the folder wherever you think best.

3.2.1 Configuring an Express installation

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

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

```
$ express
```

This would install the framework with default settings into your current folder. But that's not what we want to do. We want to take a look at some configuration options first.

CONFIGURATION OPTIONS WHEN CREATING AN EXPRESS PROJECT

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

- Which HTML template engine to use
- Which CSS pre-processor to use
- Whether to add support for sessions

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

Table 3.1 The command line configuration options when creating a new Express project

Configuration command	Effect
`--sessions`	Adds support for sessions to your project
`--css less|stylus`	Adds a CSS pre-processor to your project, either Less or Stylus depending on which you type in the command
`--ejs`	Changes the HTML template engine from Jade to EJS
`--jshtml`	Changes the HTML template engine from Jade to JsHtml
`--hogan`	Changes the HTML template engine from Jade to Hogan

For example – and this isn't what we're going to do – if you want to create a project with session support, using the Less CSS pre-processor and the Hogan template engine you would run the following command in terminal:

```
$ express --sessions -- css less --hogan
```

In our project we will need to use sessions, but to keep things simple, we don't need CSS pre-processing using Less or Stylus so we can stick with the default of plain CSS. But we do need to use a template engine, so let's take a quick look at the options.

THE DIFFERENT TEMPLATE ENGINES

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

All of the engines have their own merits and quirks, and if you already have a preferred one then that's fine. In this book we're going to be using Jade. Jade is very powerful and provides all of the functionality we're going to need. As it is the default template engine in Express you'll also find that most examples and projects online use it, so it's very helpful to be familiar with it. Finally, Jade's minimal style makes it ideal for code samples in a book!

A QUICK LOOK AT JADE

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

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

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

#A A Jade template contains no HTML tags
#B The compiled output is recognizable HTML

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

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

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

3.2.2 Create the Express project and try it out

So we know the basic command for creating an Express project, and which configuration options we want, so let's go ahead and create a new project. At the start of this chapter you should have created a new folder called *Loc8r*. Navigate to this folder in terminal, and run the following command.

```
$ express --sessions
```

This will create a bunch of folders and files inside your Loc8r folder that will form the basis of your application. But we're not quite ready yet. Next you'll need to install the dependencies.

NODE.JS APPLICATION DEPENDENCIES

Running the express command has created the Node.js application, but this won't run properly until we install the dependencies. In a Node.js application, a file in the root folder of the application called **package.json** defines the dependencies. This package.json file can contain various meta-data about your project, but at this point we're interested in the dependencies section. Open up the package.json file in your Loc8r folder, and you should see something like listing 3.2.1.

Listing 3.2.1 Example package.json file in a new Express project

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {                #A
```

```
    "express": "3.4.0",           #A
    "jade": "*"                   #A
  }                               #A
}
```

#A The dependencies in this package.json are Express version 3.4.0 and the latest version of Jade

This is the file in its entirety, so as you can see it's not particularly complex. There is various meta data at the top of the file, followed by the dependencies section. In this default installation of an Express project there are just two dependencies, Express and Jade.

WORKING WITH DEPENDENCY VERSIONS IN PACKAGE.JSON

Alongside the name of each dependency is the version number that your application is going to use. Notice that Jade has a "*". This says that the application should use the latest version of Jade. This wildcard is not normally recommended, but can be used in certain cases where the dependency is intended to always be fully backwards compatible. Jade is one of these few cases.

The dependency definition for Express 3.4.0 is much more typical. It specifies a particular version at each of the three levels of major version (3), minor version (4) and patch version (0). It is possible to throw a wildcard in to replace any of the numbers, in which case your application will use the latest version available at that level. The best practice is to put a wildcard in for the patch version. Patches should only contain fixes that shouldn't have any impact on your application, but different major and minor versions could well include changes that cause problems with your application. The wildcard character for minor versions and patch versions is x.

So you can change your package.json file to define Express version 3.4.x, and while you're at it, you can set the name of your application, like you can see in listing 3.1.2.

Listing 3.2.2 The updated package.json file

```
{
  "name": "Loc8r",                #A
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.4.x",           #B
    "jade": "*"
  }
}
```

#A Set the name of your application
#B Add a wildcard to specify the latest patch version of Express

When you've done that and saved it you are ready to install the dependencies for your application.

INSTALLING NODE.JS DEPENDENCIES

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

In terminal make sure you are in the Loc8r folder, where the package.json file is saved. Simply run the following command:

```
$ npm install
```

This tells npm to install all of the dependencies listed in your package.json file. As soon as you run it you'll see your terminal window light up with all of the things it is downloading. Once it has finished, your application is ready for a test drive.

3.2.3 Try it out

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

In terminal, in your Loc8r folder, run the following:

```
$ node app.js
```

And you should see following confirmation:

```
Express server listening on port 3000
```

This means that your Express application is running! You can see it in action by opening a browser and heading over to http://localhost:3000. Hopefully you'll see something like the screenshot in Figure 3.1.

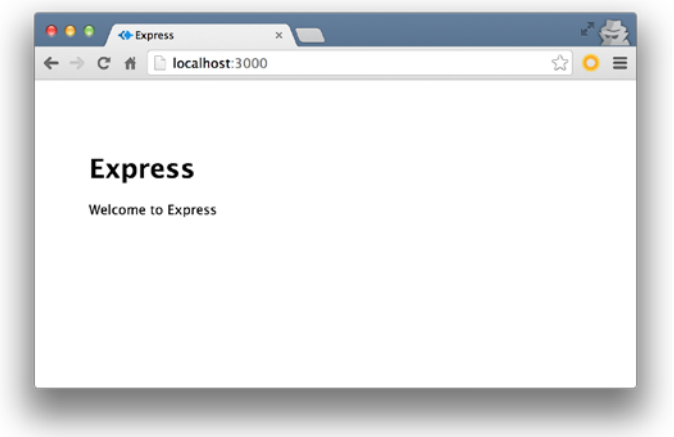


Figure 3.1 The landing page for a bare-bones Express project

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

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

How Express handles the requests

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

About Express middleware

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

Take `app.use(express.bodyParser());` for example. This will take an incoming request, parse out any submitted form fields and then attach them to the request in a way that makes it easy to reference them in your controller code.

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

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

Figure 3.2 illustrates this flow, using the example of the default Express homepage that you've just run.

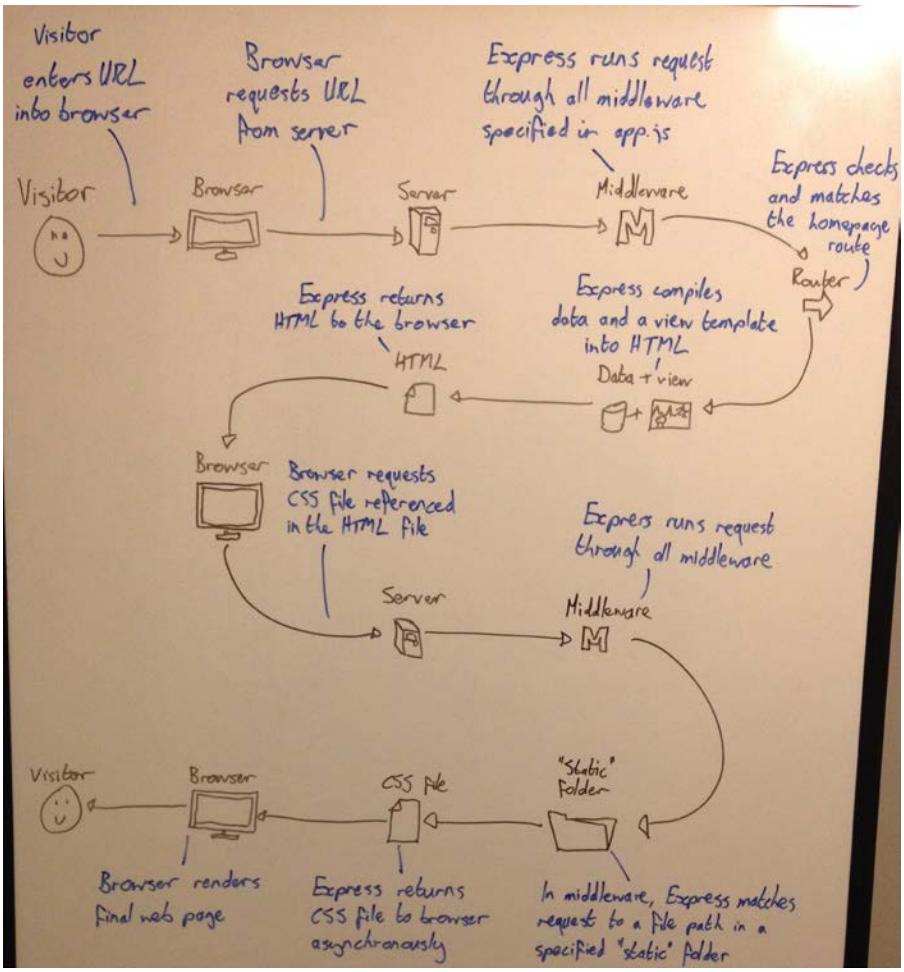


Figure 3.2 The key interactions and processes that Express goes through when responding to the request for the default landing page. The HTML page is processed by Node to compile data and a view template, and the CSS file is served asynchronously from a static folder.

The flow in Figure 3.2 shows the separate requests made and how Express handles them differently. Both requests run through the middleware as a first action, but the outcomes are very different.

3.2.4 Restarting your application

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

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

Restarting the Node.js process is a two-step procedure. First you have to stop the running process. You do this in terminal by pressing Ctrl+C. Then you have to start the process again in terminal using the same command as before – `node app.js`.

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

AUTOMATICALLY RESTARTING YOUR APPLICATION WITH NODEMON

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

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

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

When the installation has finished, you will be able to use nodemon wherever you wish. Using it is really simple. Instead of typing node to start your application, you type nodemon. So, making sure you are in your Loc8r folder in terminal – and that you have stopped the Node.js process if it is still running – enter the following command:

```
$ nodemon app.js
```

You should see that a few extra lines are output to the terminal confirming that nodemon is running, and then the familiar log `Express server listening on port 3000`. If you head back over to your browser and refresh, you should see that you application is still there.

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

3.3 Best practice for app development

Before we dive in and start coding stuff, it's important to pause for a moment and think about what we want to build, and plan how we approach it.

3.3.1 A phased approach to development

Back in Chapter 1 we talked about how we will build an application in stages, focusing on one layer at a time. To recap, the stages that we talked about are:

- Stage 1: Building a static site
- Stage 2: Create the database and design the data model

- Stage 3: Hook the database into the application
- Stage 4: Enable users to login
- Stage 5: Turbo-charge the front-end with AngularJS

Approaching application development like this helps you to separate your concerns, and keeps the distinct parts decoupled from each other as much as possible. The order that you approach the layers is not necessarily important, and will depend on a number of factors, such as whether you have a design and UX team providing a look-and-feel for the front-end, or if you have an existing database and data model that you need to work with.

In this book we are going to follow a rapid prototyping approach, making an idea into a product as quickly as possible, as efficiently as we can. Figure 3.3 illustrates the stages involved, and how we really do build up the application layer by layer.

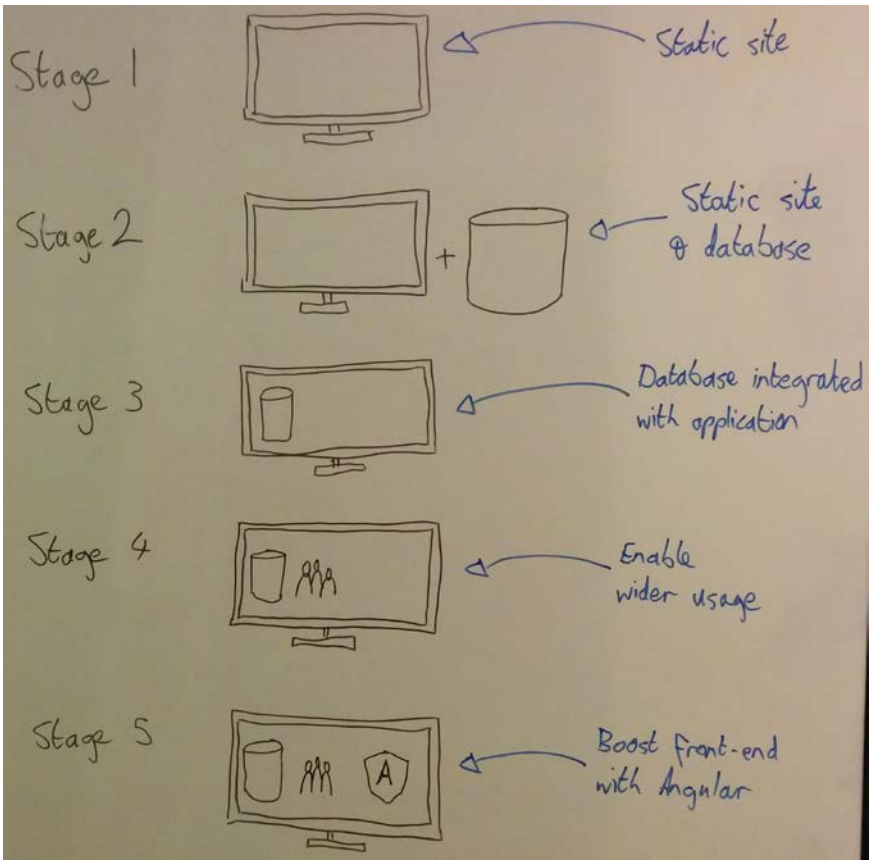


Figure 3.3 The five stages of rapid prototyping used in this book to build an application from an idea using all of the MEAN stack

In Stage 1 we focus on what is on each page, and on how you navigate through and use the application. This helps solidify what we need in our data model, so we work on that in Stage 2, and Stage 3 sees us tie the front-end and the database together with controllers and application logic. To finalize the working application, in Stage 4 we enable users to login to provide a personalized experience. After this, once we have a basic application, we can really let the front-end take over by using Angular JS in Stage 5.

3.3.2 Build an internal API for your data layer

When you get to the stage of creating your data layer, how you approach this can make your future life really easy, or really difficult. I personally favor easy. The temptation though can be to focus on a "quick win" and only think about what you are doing right now. This would probably lead you down the path of talking to the database directly from your Node application code. With a short-term view this is the easy way. But with a long-term view this becomes the difficult way, as it will tightly couple your data to your application code in a way that nothing else could use it.

The other option is to build your own API, which can talk to the database directly, and output the data you need. Your Node application can then talk with this API instead of directly with the database. Figure 3.4 shows a comparison of the two setups.

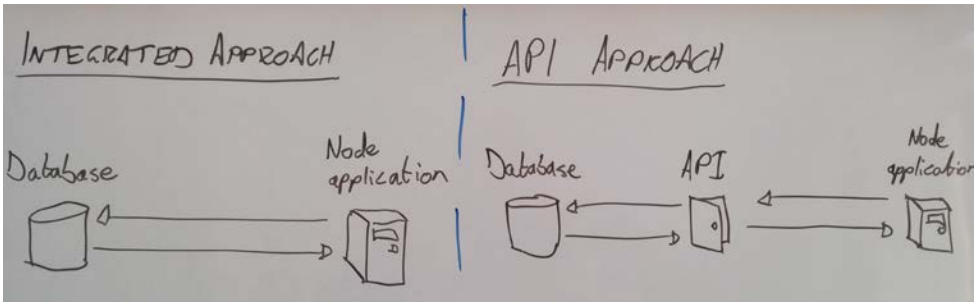


Figure 3.4 The short-term view of data integration into your Node.js application. You can set up your Node.js application can talk directly to your database, or you can create an API that interacts with the database, and have your Node application talk only with the API.

Looking at Figure 3.4 you could well be wondering why you'd want to go to the effort of creating an API just to sit in between your application and your database. Isn't it creating more work? At this stage, yes, it is creating more work—but we're looking further down the road here. What if you want to use your data in a native mobile application a little later? Or – for example – in an AngularJS front-end?

You certainly don't want to find yourself in the position where you have to write separate but similar interfaces for each. If you have built your own API up front, which outputs the data you need, you can avoid all of this. If you an API in place, when you want to integrate the data layer into your application you can simply make it reference your API. Now it

doesn't matter if your application is Node.js, AngularJS or iOS. It doesn't have to be a public API that anyone can use, so long as you can access it. Figure 3.5 shows a comparison of the two approaches when you have Node, Angular and iOS applications all using the same data source.

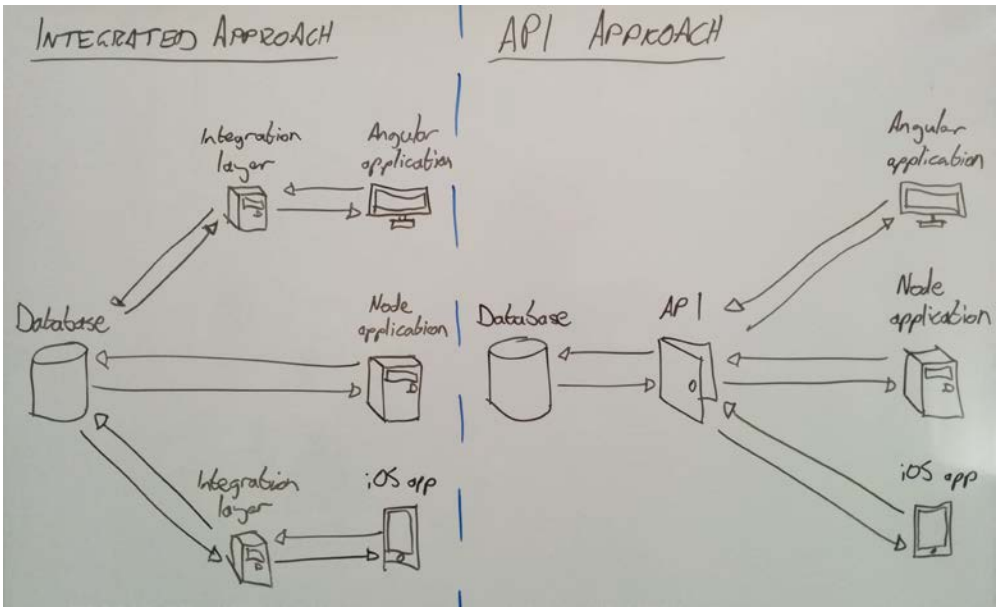


Figure 3.5 The long-term view of data integration into your Node application, and additional Angular and iOS applications. The integrated approach has now become fragmented whereas the API approach is simple and maintainable.

The previously simple 'integrated approach' is now becoming fragmented and complex. You'll have three data integrations to manage and maintain, so any changes will have to be made in multiple places in order to retain consistency. If you have a single API you don't have any of these worries. So with a little bit of extra work at the beginning, you can make life much easier for yourself further down the road. We'll look at creating the internal API for Loc8r in Chapter 6, once we have a static site and data model.

This phased approach sounds great, but how do you turn this theory into practice, and bring it to life in code? The answer is by using the MVC architecture.

3.4 Modifying Express for MVC

First off, what is MVC architecture? MVC stands for model-view-controller, and is an architecture that aims to separate out the data (model), the display (view) and the

application logic (controller). Sounds a bit like stages 1, 2, and 3 of our rapid prototype approach doesn't it?

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

3.4.1 A bird's eye view of MVC

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

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

In reality, depending on your setup, the controller may actually compile the view before sending the response back to the visitor. The effect is the same though, so let's keep this simple flow in mind as a visual for what will happen in our application. See Figure 3.6 for an illustration of this loop.

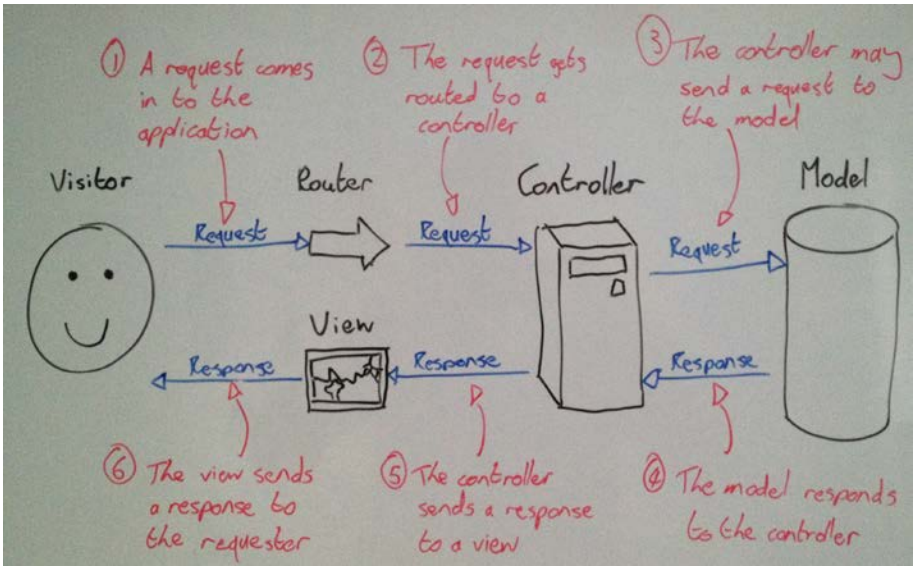


Figure 3.6 The request-response flow of a basic MVC architecture.

Figure 3.6 highlights the individual parts of the MVC architecture, and how they link together. It also illustrates the need for a routing mechanism along with the Model, View and Controller components. So now that you have seen how you want the basic flow of your application to work, it's time to modify the Express setup to make this happen.

3.4.2 Changing the folder structure

If you look inside your newly created Express project in your Loc8r folder, you should see a file structure including a views folder, and even a routes folder, but no mention of models or controllers. Rather than just going ahead and cluttering up the root level of the application with some new folders, let's keep things tidy with one new folder for all of our MVC architecture. Three quick steps here:

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

Figure 3.7 illustrates these changes and shows the folder structures before and after the modifications.

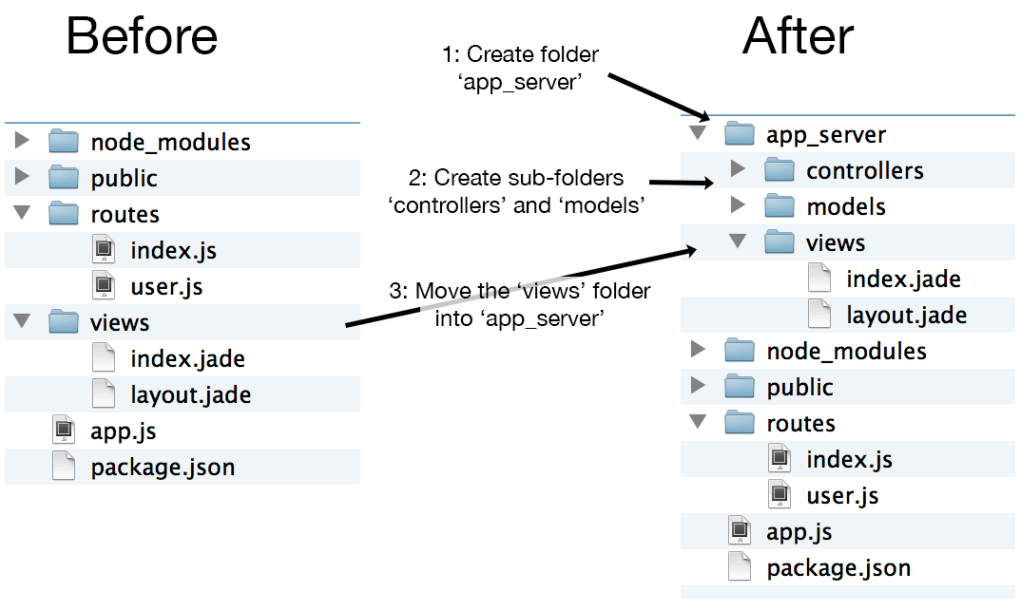


Figure 3.7 Changing the folder structure of an Express project into an MVC architecture

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

3.4.3 Use your new views folder

The first thing we need to do is tell Express that we've moved the views folder. Express will be looking in `/views` but we've just moved it to `/app_server/views`. Changing it is really simple. In app.js find the line:

```
app.set('views', __dirname + '/views');
```

… and change it to the following (changes in bold):

```
app.set('views', __dirname + '/app_server/views');
```

If you save that and run your application again, you'll find that you've fixed it and your application works once more!

3.4.4 Splitting controllers from routes

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

Start converting the setup by moving the controller for the default Express homepage into the correct place. In `/app_server/controllers` create a new file **`main.js`**. In this file paste the contents of `/routes/index.js`, which is the following snippet.

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

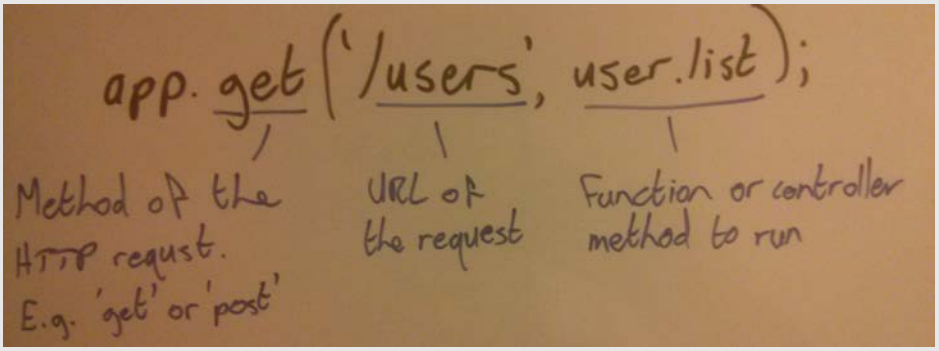
That's the controller ready but not connected to anything, now it's time to set up the routing to make use of the controller.

3.4.5 Sort out the routing

A lot of examples out there will just set up the routing inside app.js, which is where Express puts them by default. This is fine for a quick example, but app.js should really be used for the setup of the application, and not be concerned with the inner workings.

Anatomy of an Express route definition

Routes in Express are designed to map an incoming HTTP request to a function or controller. This is the general construct for a route:



In this example when a GET request comes into your URL path '/users', Express will use the controller user.list. If someone tried to make a POST request to the same URL path it would fail, as the route is only set up for GET requests.

The following two lines in app.js define the default routes that come with an Express installation.

```
app.get('/', routes.index);
app.get('/users', user.list);
```

We're not interested in the second one, as we only have a single index page at the moment. So go ahead and delete the `'/users'` route. Now we want to move the remaining route to somewhere more appropriate.

MOVING THE ROUTES OUT OF THE MAIN FILE

This bit of best practice is to move the routes out of app.js and into their own file. In Node.js, to reference code in an external file you have to create a new module and then `require` it – see the *Creating and using Node.js modules* sidebar for some overarching principle behind this process.

Creating and using Node.js modules

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

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

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

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

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

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

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

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

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

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

We currently have an empty file in routes/index.js as we moved the contents of it in a controller. We can make use of this to do three things:

1. Define the route for the index page
2. Expose the route to the application
3. Reference the controller so that Express knows what to do when a request comes in

In routes/index.js you can do this by adding the few simple lines of code in the following snippet.

```
var ctrl = require('../app_server/controllers/main');     #A

module.exports = function (app) {                         #B
  app.get('/', ctrl.index);                               #C
};
```

#A Require the controllers files
#B Create the exports function to expose the routes
#C Define the route for the homepage, using the 'index' controller

This small snippet links your route to the new controller, and exposes the route to the application. Notice that the `module.exports` function accepts the single parameter app, enabling the routes inside to reference the `get` and `post` methods.

All that remains is to make sure that this file is being called in properly from the main app.js file. Firstly, at the top of app.js delete the following two lines, as they are no longer needed.

```
var routes = require('./routes');
var user = require('./routes/user');
```

Secondly, towards the bottom of the file, we want to replace the existing homepage route, with a reference to the new file. So simple delete this line:

```
app.get('/', routes.index);
```

And replace it with this line:

```
require('./routes')(app);
```

We still need to require the routes file, but as we're not accessing any of the individual methods from inside app.js we don't need to assign it to a variable. Also notice how app is passed through as a parameter to the `module.exports` function.

If you test this out now in your browser, you should see that the default Express homepage displays correctly once again. But there is one more piece of best practice that needs looking at.

SPLITTING THE ROUTES INTO SEPARATE FILES

The setup we have now works perfectly, but all of the routes are going to be contained in a single file. This not ideal when your application grows as you end up with a large number of routes, which can give you a bit of a maintenance headache. So the approach here is to separate your routes into logical collections. A typical example would be 'user', as you might well have URLs for user/profile, user/edit, user/update and so on. We'll get into the building of the routes in more detail when we start building Loc8r.

Alongside these collections you'll also probably need a 'catch all' collection for single pages that don't belong anywhere else. That's where we'll put the routing for the homepage of the default Express app, as it doesn't belong anywhere else.

Matching the naming of the controller setup, make a copy of `routes/index.js` and rename it `main.js`.

The `routes/index.js` file now simply needs to reference `main.js`, and pass through whatever `main.js` is exporting back up to `app.js`. So delete the existing code, and use `module.exports` again, this time requiring the new `main.js` file, like in the following snippet.

```
module.exports = function(app){
  require('./main')(app);
};
```

So that's pretty simple. Now, when the application grows and you need a new collection of routes, you can simply create a new file and require it in this function. This approach gives your codebase clarity and makes future maintenance and expansion a much easier task.

3.4.6 Confirming the routing and controller approach

We've now got everything setup for a strong base to build an application on. But let's take a moment to look at what we've got and how we're going to use it going forward.

For taking a URL request, app.js includes routes/index.js, which in turn includes a number of files containing routes. These files are logical collections of routes, and each file references a controller file. Thinking forward a little bit, Loc8r will probably need collections of location and user routes, as well as the main catch-all collection. The architecture for this is shown in Figure 3.8.

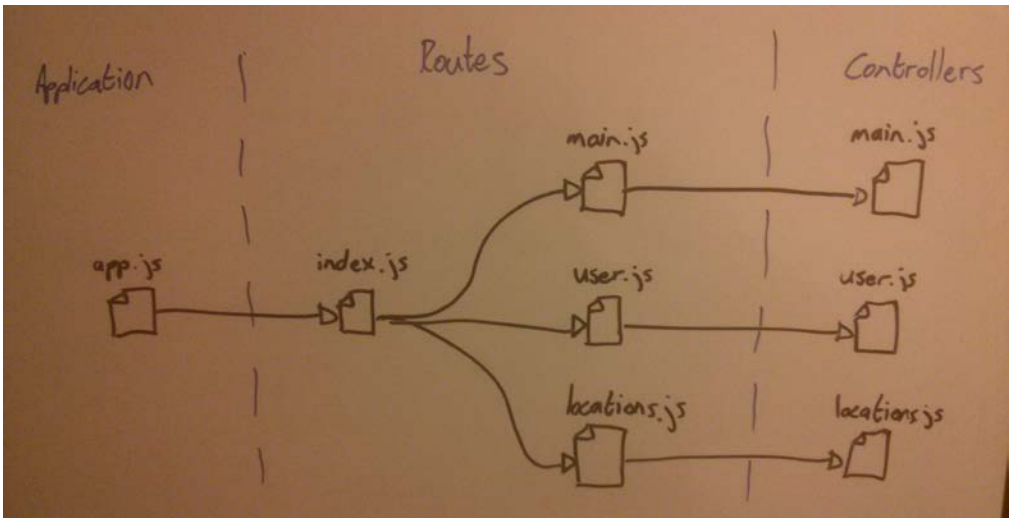


Figure 3.8 The routing architecture of the modified setup, from the core application file through the routes to the controllers

At this point we only have the main route and controller, so you could feel that the approach is a bit over the top and gets in the way of getting started. Hopefully Figure 3.8 goes some way towards showing why this is useful in the long term. Of course, you only have to do this once, as you can keep this setup as a template, either locally or up on GitHub.

If you have an application that continues to grow, then even further down the line you might very well want to split the separate controllers out more, so that one of your routes

files requires multiple controllers. If your application gets really big and your controllers are really complex this can be a good way to help keep your code maintainable.

3.4.7 Checking app.js

There have been a few changes made to app.js in this chapter so far, so let's take a brief moment to take a look at the end result and see what's what. Listing 3.4.1 shows the end result, and notes the changes we've made to get there.

Listing 3.4.1 The final app.js, and what we did to get here

```
var express = require('express');                                    #1
var http = require('http');                                          #1
var path = require('path');                                          #1

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/app_server/views');                   #2
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(express.cookieParser('your secret here'));
app.use(express.session());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

require('./routes')(app);                                            #3

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

#1 Removed references to the routes from the dependencies section
#2 Updated the views to use the new MVC folder structure
#3 Required the new routing architecture, passing app through as a parameter

Right at the top of the file we removed the references to routing from the dependencies section #1. Further down we also removed the inline route definitions and replaced them with a reference to the new routing architecture #3. In the middle of the code we changed the location of the views folder #2, telling Express that the Jade templates are now in our newly created MVC app_server folder.

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

3.5 Import Bootstrap for quick responsive layouts

As we discussed in Chapter 1, Loc8r will make use of Twitter's Bootstrap framework to speed up the development of a responsive design. We'll also make the application stand out by using a theme.

3.5.1 Download bootstrap and add to the application

Instructions for downloading Bootstrap, getting a custom theme and adding the files into your project folder are all found in Appendix B. A key point here is that the Bootstrap files are all static files to be sent directly to the browser; they don't need any processing by the Node engine. Your Express application will already have a folder for this purpose – the *public* folder. When you have it ready, your public folder should look something like Figure 3.9.

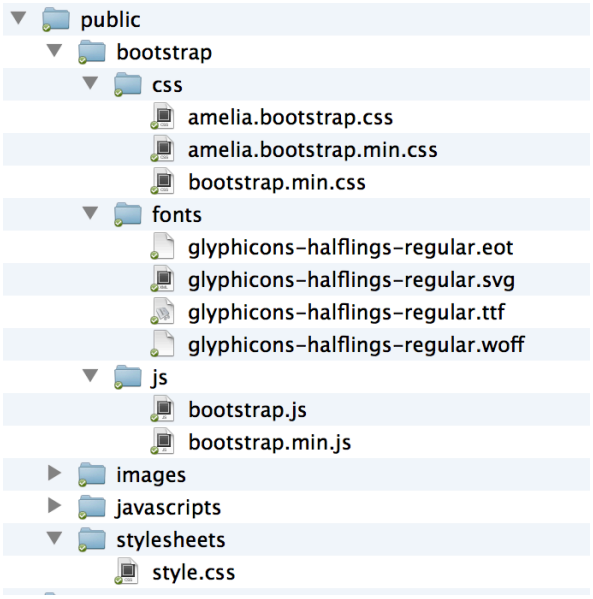


Figure 3.9 The structure of the 'public' folder in your Express application after adding Bootstrap

3.5.2 Using Bootstrap in the application

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

WORKING WITH JADE TEMPLATES

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

This is how Jade appears in a default Express installation. If you look in the views folder in your application you'll see two files, `layout.jade` and `index.jade`. The `index.jade` file is controlling the content for the index page of your application. Open it up, and there's not much in there – the entire contents are shown in Listing 3.5.1.

Listing 3.5.1 The complete index.jade file

```
extends layout              #1

block content               #2
  h1= title                 #3
  p Welcome to #{title}     #3
```

#1 Informs you that this file is extending the layout file
#2 Informs you that the following section goes into an area of the layout file called content
#3 Outputs a h1 and p tag to the content area

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

There are no references to <head> or <body> tags here, nor any stylesheet references. These are all handled in the layout file, so that's more likely where you want to go to add in global scripts and stylesheets to your application. Open up layout.jade and you should see something similar to Listing 3.5.2.

Listing 3.5.2 Default layout.jade file

```
doctype 5
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content                                              #A
```

#A Empty named block can be used by other templates

Listing 3.5.2 shows the layout file being used for the basic index page in the default Express installation. You'll see that there's a head section and a body section and within the body section there is a `block content` line with nothing inside it. This named `block` can be

referenced by other Jade templates, such as the `index.jade` file in Listing 3.4.1. The `block content` from the index file gets pushed into the `block content` area of the layout file when the views are compiled.

ADDING BOOTSTRAP TO THE ENTIRE APPLICATION

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

1. Reference the Bootstrap CSS file
2. Reference the Bootstrap JavaScript file
3. Reference jQuery, which Bootstrap requires
4. Add viewport meta data, so that the page scales nicely on mobile devices

The CSS file and the viewport meta data should both be in the head of the document, and the two script files should be at the end of the body section. Listing 3.5.3 shows all of this in place in `layout.jade`, with the new lines highlighted in bold.

Listing 3.5.3 Updated layout.jade including Bootstrap references

```
doctype 5
html
  head
    meta(name='viewport', content='width=device-width, initial-scale=1.0')
#A
    title= title
    link(rel='stylesheet', href='/bootstrap/css/amelia.bootstrap.css')  #B
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content


script(src='//ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js')
#C
    script(src='/bootstrap/js/bootstrap.min.js')  #D
```

#A Set the viewport meta data for better display on mobile devices
#B Include the themed bootstrap CSS
#C Bring in jQuery as it is needed by Bootstrap
#D Bring in the Bootstrap JavaScript file

With that done, any new template that you create will automatically have Bootstrap included and will scale on mobile devices. So long as your new templates extend the layout template of course.

Finally, before testing it all out, delete the contents of the `style.css` file in `/public/stylesheets/`. This will prevent the default Express styles from overriding the Bootstrap files. We'll want to add our own styles in somewhere a little later down the line, so there's no need to delete the file.

VERIFY THAT IT WORKS

If your application isn't already running with nodemon, start it up and view it in your browser. The content hasn't changed, but the appearance should have. You should now have something looking like Figure 3.10.

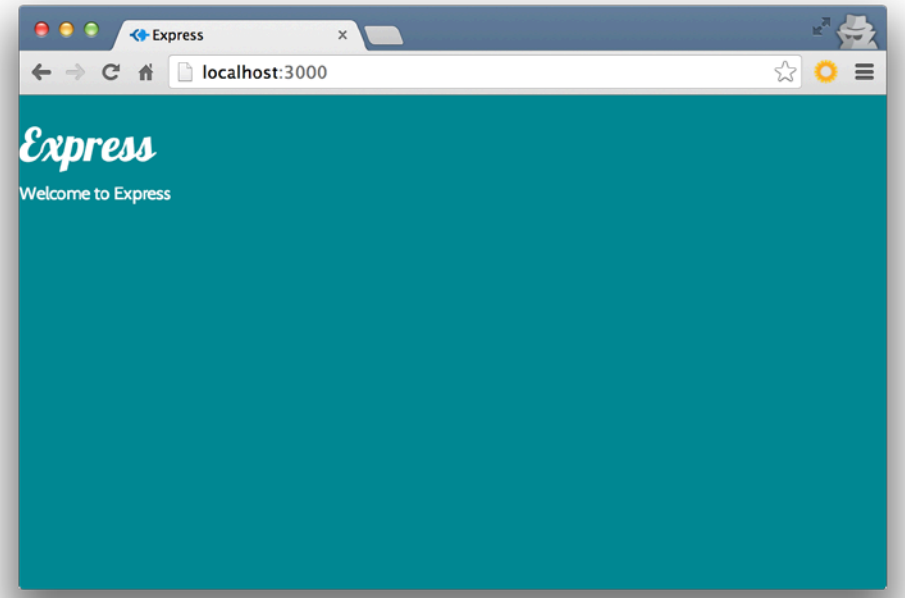


Figure 3.10 The Bootstrap theme having an effect on the default Express index page

3.6 Make it live on Heroku

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

3.6.1 Getting Heroku set up (supported by Appendix B)

Before you can use Heroku, you will need to sign up for a free account and install the Heroku Toolbelt on your development machine. Appendix B has more detailed information on how to do this. You'll also need a bash-compatible terminal – the default command line interface for

Windows users won't do, you'll need to download something like the GitHub Terminal, which comes as part of the GitHub desktop application.

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

UPDATE PACKAGE.JSON

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

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

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

At the time of writing, these commands return `v0.10.13` and `1.3.2` respectively. Using a wildcard for a patch version as we've seen previously, you need to add these to a new `engines` section in your `package.json` file. The complete updated `package.json` file is shown in Listing 3.6.1, with the added section in bold.

Listing 3.6.1 Adding an engines section to package.json

```
{
  "name": "Loc8r",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "engines": {                    #A
    "node": "0.10.x",             #A
    "npm": "1.3.x"                #A
  },                              #A
  "dependencies": {
    "express": "3.4.x",
    "jade": "*"
  }
}
```

#A Add an "engines" section to package.json to tell Heroku which platform the application is on, and which version to use

When pushed up to Heroku, this will tell Heroku that our application uses the latest patch version of Node.js v0.10 and the latest patch version of npm 1.3.

CREATE A PROCFILE

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

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

```
web: node app.js
```

When pushed up to Heroku, this file will simply tell Heroku that the application needs a web process and that is should run node `app.js`.

TEST IT LOCALLY WITH FOREMAN

The Heroku Toolbelt comes with a utility called Foreman. You can use Foreman to verify your setup and run your application locally, before pushing the application up to Heroku. If your application is currently running, stop it by pressing Ctrl+C in the terminal window running the process. Then in the terminal window enter:

```
$ foreman start
```

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

```
06:55:38 web.1  | started with pid 89869
06:55:38 web.1  | Express server listening on port 5000
```

If you fire up a browser and head over to localhost:5000 you should be able to see your application up and running once again.

Now that you know your setup is working it is time to push your application up to Heroku.

3.6.2 Push the site live using Git

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

STORE THE APPLICATION IN GIT

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

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

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

```
$ git init                          #A
$ git add .                         #B
$ git commit -m "First commit"      #C
```

#A Initializes the folder as a local Git repository
#B Adds everything in the folder to the repository
#C Commits the changes to the repository with a message

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

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

CREATE THE HEROKU APP

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

```
$ heroku create
```

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

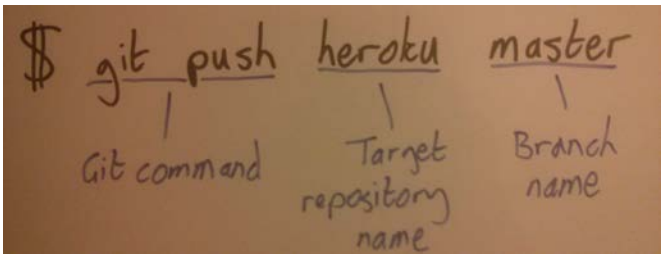
```
http://shrouded-tor-1673.herokuapp.com/ | git@heroku.com:shrouded-tor-
1673.git
Git remote heroku added
```

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

DEPLOYING YOUR APPLICATION TO HEROKU

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

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



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

When you run this, terminal will display a load of log messages as it goes through the process, eventually ending up with a confirmation that your app has been deployed to Heroku. This will be something like the following, except you'll have a different URL of course.

```
http://shrouded-tor-1673.herokuapp.com deployed to Heroku
```

What are Git branches?

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

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

START A WEB DYNO ON HEROKU

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

Heroku also has a great free tier, which is perfect for application prototyping and building a proof-of-concept. You get one web dyno for free, which is more than adequate for our purposes here. Before you can view your application online you need to add a single web dyno. This is easily done with a simple terminal command.

```
$ heroku ps:scale web=1
```

When you've run this, terminal will display a confirmation.

```
Scaling web dynos... done, now running 1
```

VIEW YOUR APPLICATION ON A LIVE URL

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

```
$ heroku open
```

This will launch your application in your default browser. You should see something like Figure 3.11.

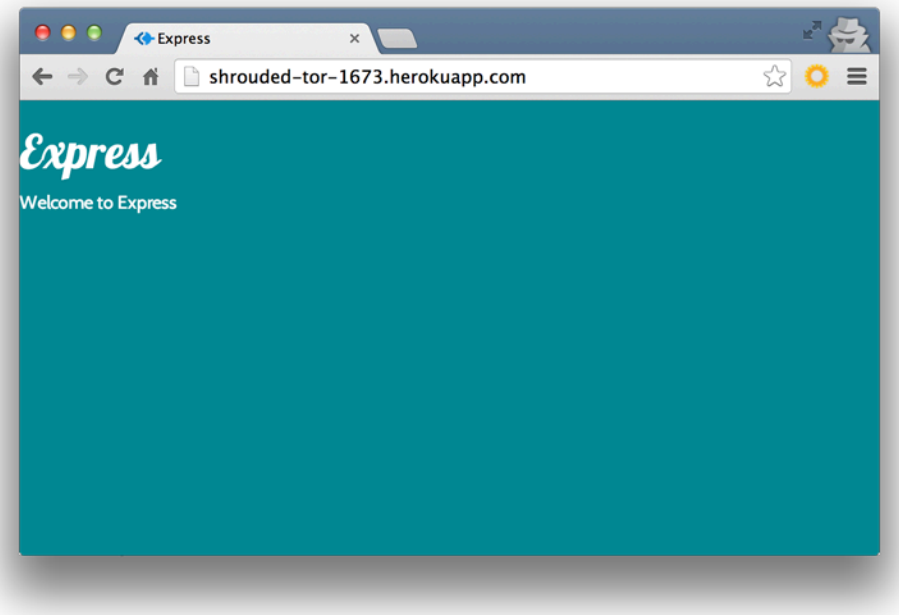


Figure 3.11 The MVC Express application running on a live URL

Your URL will be different of course, and within Heroku you can change it to use your domain name instead of the address it has given you. In the application settings on the Heroku website you can also change it to use a more meaningful sub-domain of herokuapp.com. For example, I have just changed the URL of my Loc8r application to be http://getting-mean-loc8r.herokuapp.com/. All future screenshots of the externally accessible site in this book will have this URL. You can also visit it to see the final application in action.

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

THE SIMPLE UPDATE PROCESS

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

```
$ git add .                                #A
$ git commit -m "Commit message here"      #B
$ git push heroku master                   #C
```

#A Add all changes to your local Git repository
#B Commit the changes to your local repository, with a useful message
#C Push the changes to the Heroku repository

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

3.7 Summary

The easiest way to create a new Express application requires Express to have already been installed globally using npm. You also need Node.js and npm installed globally on your system. A simple terminal command can then create a new skeleton Express application in a folder of your choice. You can using this terminal command to define some of the features to use, such as whether to add session support, which template engine to use and which CSS pre-processor to use (if any).

A default Express installation is set up for getting up and running quickly, but not really for long-term scalability. It is not heavily opinionated, so you can change the setup to meet your needs. Express allows an MVC style approach of application architecture, but is not by default explicitly set up that way. Modifying an Express setup to be MVC is relatively straightforward, by the MVC components into a separate 'app_server' folder. You can also make the route management more scalable by taking the route definitions out of the main app.js file and moving them into separate logical collections.

The MVC structure encourages a separation of concerns when developing, and encourages a good development best practice – build your application one layer at a time. Another best practice is to develop an internal API, so that your data layer is not tightly locked into your application code.

Heroku is great for getting a Node.js / Express application hosted quickly and easily. Most of the setup is in creating your account and preparing your machine. Once that is done it uses Git to push the code in your local repository to the Heroku repository. When your application is on Heroku – and you have a web dyno running – you can see it live on a real URL.

Appendix A

Installing the stack

This appendix covers

- Installing Node.js and npm
- Installing Express globally
- Installing MongoDB

Before you can build anything on the MEAN stack you'll need to install the software to run it. This is really easy to do on Windows, OSX and the more popular Linux distributions such as Ubuntu.

As Node.js underpins the stack, that's the best place to start. Node.js now also ships with npm included, which will be very useful for installing some of the other software.

A.1 Install Node.js and npm

The best option for Windows and Mac OSX users is to simply download installers from the Node.js website. This location always has the latest version as maintained by the Node.js core team. Linux users can also download binaries if you're comfortable working with them. Downloads for all of these OS's are available here http://nodejs.org/download/

OSX and Linux users can also install Node.js from package managers. Package managers do not always have the latest version, so be aware of that. A particularly out-of-date one is the popular *apt* system on Ubuntu. There are instructions for using a variety of package managers – including Homebrew for OSX and a fix for apt on Ubuntu – on Joyent's Node.js wiki on GitHub https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager

A.1.1 Verify installation by checking version

Once you have Node.js and npm installed you can check the versions you have with a couple of terminal commands.

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

These will output the versions of Node.js and npm that you have on your machine.

A.2 Install Express globally

In order to be able to create new Express applications on the fly from the command line, you need to install it globally so that is accessible anywhere on your machine. You can do this from the command line, using npm. In terminal you simply run the following command:

```
$ npm install -g express
```

When this has finished installing you can verify it by checking the version number from terminal.

```
$ express –version
```

If you run into any problems with this, the documentation for Express is available on its website http://expressjs.com/

A.3 Install MongoDB

MongoDB is also available for Windows, Mac and Linux. There are some direct downloads available for Windows, depending on which version you are running. The easiest way to install MongoDB for Mac is to use the Homebrew package manager, and there are also packages available for a few Linux distributions. On the Mac, if you prefer, you can also choose to install MongoDB manually. The same goes if you are running a version of Linux that doesn't have MongoDB available in a package.

Detailed instructions about all of the above options are available in the MongoDB online documentation http://docs.mongodb.org/manual/installation/

A.3.1 Running MongoDB as a service

Once you have MongoDB installed, you'll probably want to run it as a service, so that it automatically restarts whenever you reboot. Again, there are instructions for doing this in the MongoDB installation documentation.

Appendix B

Installing & preparing the supporting cast

This appendix covers

- Adding Twitter Bootstrap and a custom theme
- Installing Git
- Installing a suitable command line interface
- Signing up for Heroku
- Installing Heroku toolbelt

There are several technologies that can help with developing on the MEAN stack, from front-end layouts to source control and deployment tools. This Appendix covers the installation and setup of the supporting technologies used through Getting MEAN.

B.1 Twitter Bootstrap

Bootstrap is not really installed as such, but rather added to your application. This is as simple as downloading the library files, unzipping and placing them into the application.

The first step of course is to download Bootstrap. You can get this from getbootstrap.com. Make sure that you download the distribution zip and not the source. At the time of writing Bootstrap is on version 3.0.2 and the distribution zip contains three folders: css, fonts and js.

Once you have it downloaded an unzipped, the files need to be moved into the *public* folder in your Express application. To keep the files together, and the top level clean create a new folder called `bootstrap` in the public folder and copy the unzipped files into there. The public folder in your application should now look like Figure B.1.

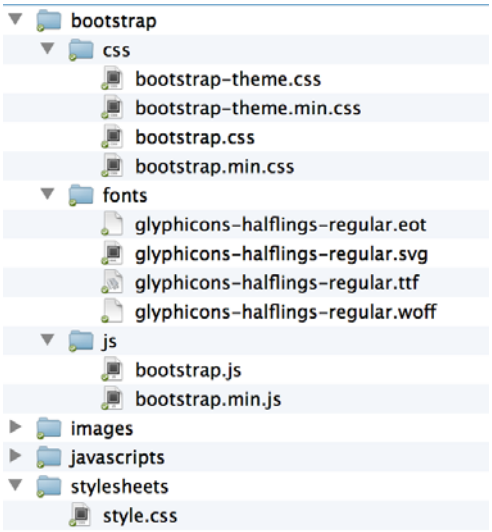


Figure B.1 The structure and contents of the public folder after Boostrap has been added

That will give you access to the default look and feel of Bootstrap, but you probably want your application to stand out from the crowd a bit. You can do this by adding in a theme.

B.1.1 Getting the 'Amelia' theme

The Loc8r application in Getting MEAN uses a bootstrap theme called Amelia from Bootswatch. You can download both the original CSS file and the minified version from http://bootswatch.com/amelia/. Once they are downloaded, so that you don't overwrite the original Bootstrap files rename them to `amelia.bootstrap.css` and `amelia.bootstrap.min.css`.

You can now copy these into the `/public/bootstrap/css` folder in your application.

B.1.2 Tidying up the folders

If you wish you can tidy up your bootstrap folder by removing some of the duplicates. You'll notice that there are readable and minified versions of the CSS and JavaScript files, and also a pair of default bootstrap-theme CSS files. Unless you're going to hack around in the files you only really need the minified versions.

B.2 Install Git

The source code for this book is managed using Git, so the easiest way to access it is with Git. Also, Heroku relies on Git for managing the deployment process and pushing code from your development machine into a live environment. So you need to install Git if you don't already have it.

You can verify if you have it with a simple terminal command:

```
$git --version
```

If this responds with a version number then you already have it installed and can move onto the next section. If not, then you'll need to install Git.

A good starting point for Mac and Windows who are new to Git is to download and install the GitHub user interface https://help.github.com/articles/set-up-git

You don't need a GUI though, and you can install just Git by itself using the instructions found on the main Git website http://git-scm.com/downloads

B.3 Install a suitable command line interface

You can get the most out of Git by using a command line interface, even if you have downloaded and installed a GUI. Some are better than others, and you can't actually use the native Windows command prompt, so if you're on Windows then you'll definitely need to run something else. Here's what I use in a few different environments:

- Mac OSX Mavericks and later: native terminal
- Mac OSX pre-Mavericks (10.8.5 and earlier): iTerm
- Windows: GitHub shell (this comes installed with the GitHub GUI)
- Ubuntu: native terminal

If you have other preferences – and the Git commands work – then by all means use what you already have and you're used to.

B.4 Set up Heroku

Getting MEAN uses Heroku for hosting the Loc8r application in a live production environment. You can to this too – for free – so long as you sign up, install the toolbelt and log in through terminal.

B.4.1 Sign up for Heroku

In order to use Heroku you will need to sign up for an account of course. For the purposes of the application you'll be building through the book a free account will be fine. Simply head over to www.heroku.com and follow the instructions to sign up.

B.4.2 Install Heroku toolbelt

Heroku toolbelt contains the Heroku command line shell and a utility called Foreman. The shell is what you'll use from terminal to manage your Heroku deployment, and Foreman is very useful for making sure what you've built on your machine is setup to run properly on Heroku. You can download the toolbelt for Mac, Windows and Linux from toolbelt.heroku.com

B.4.3 Login to Heroku in terminal

Once you have signed up for an account and installed the toolbelt on your machine, the last step is to login to your account from terminal. Enter the following command:

```
$ heroku login
```

This will prompt you for your username and password, and will most likely generate a new SSH public key and upload it for you. Now you're all setup and ready to go with Heroku.