Getting MEAN with Mongo, Express, Angular, and Node

Simon Holmes
brief contents

PART 1 SETTING THE BASELINE .................................................1

1 ■ Introducing full-stack development  3
2 ■ Designing a MEAN stack architecture  24

PART 2 BUILDING A NODE WEB APPLICATION...........................51

3 ■ Creating and setting up a MEAN project  53
4 ■ Building a static site with Node and Express  80
5 ■ Building a data model with MongoDB and Mongoose  120
6 ■ Writing a REST API: Exposing the MongoDB database to the application  160
7 ■ Consuming a REST API: Using an API from inside Express  202

PART 3 ADDING A DYNAMIC FRONT END WITH ANGULAR...........241

8 ■ Adding Angular components to an Express application  243
PART 4 MANAGING AUTHENTICATION AND USER SESSIONS ......347

11 ■ Authenticating users, managing sessions, and securing APIs 349
Reintroducing JavaScript

This appendix 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 because 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’s written. Unfortunately, this looseness and low barrier to entry can encourage bad habits and sometimes have some quite unexpected results.

The aim of this appendix isn’t 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 appendix 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 here.
D.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 D.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!

```javascript
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);
```

Listing D.1 Example JavaScript file with intentional bugs
How did you get on with that? There are a couple of intentional “bugs” in listing D.1 that 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
3: 35 United Kingdom
6: Simon Holmes 35 United Kingdom
7: 37 United Kingdom
4: Age: 30
8: 37 United Kingdom
5: Age: 43
```

Among other things, this code snippet shows 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 go over it all.

### D.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’s 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.

#### D.2.1 Variables, functions, and scope

The 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’s accessible by everything, but if you declare a variable inside a function it’s 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:

```javascript
var firstname = 'Simon';
var addSurname = function(){
    var age = 30;
    function getAge()
```

```javascript
    return age;
}
```

```javascript
    console.log(getAge());  // 30
    setTimeout(getAge, 0);  // 43
}
```

```javascript
Age is undefined due to scope clashes and variable hoisting
Country hasn’t changed but age has, due to variable scopes
Runs later due to setTimeout; age is wrong due to “mistake” in for loop
Runs when called, not when defined; uses local variables over global
```

Variable declared in global scope
```
var surname = 'Holmes';
function addSurname() {
    console.log(firstname + ' ' + surname);
}
addSurname();
console.log(firstname + ' ' + surname);
```

This piece of code throws an error because it’s 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. In figure D.1, the outer circle depicts the global scope and the inner circle depicts the local scope.

![Figure D.1](scope-circles.png)

In figure D.1 you 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 you want the global scope to be able to output the full name while keeping the surname private in the local scope, you need a way of pushing the value into the global scope. In terms of the scope circles you’re aiming for what you see in figure D.2. You want a new variable `fullname` that you can use in both global and local scopes.

![Figure D.2](additional-global-variable.png)
**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’s `global`. Sticking with browser examples for now, let’s see how this would look if you update the code to use the `fullname` variable:

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

This approach allows you to add a variable to the global scope from inside a local scope, but it’s not ideal. The problems are two-fold. First, if anything goes wrong with the `addSurname` function and the variable isn’t 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 track of it? How do you test it? How do you explain to someone else what’s 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 you need a global variable, then you should define it in the global scope, like in the following code snippet:

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

Here it’s 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 the code still references the global variable using the fully qualified `window.fullname`. It’s 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, because you can explicitly see which variable is being referenced. So the code should look like this:

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

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’s another reason for this, particularly when 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, if you declare a variable without using `var`, JavaScript will create the variable in the global scope. For example

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

Hopefully you can see how this could be confusing, and is a bad practice. So the takeaway 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’s correct, and the reason is because of *variable hoisting*. Variable hoisting means that JavaScript declares all variables at the top anyway, without telling you! This can lead to some unexpected results.

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

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

You expect this to use global variable

But 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 code snippet, JavaScript sees the following:

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

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 you 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’s less room for error and unexpected problems.

**No such thing as block scope**

In various programming languages there’s also the concept of block scope. This is a scope within, for example, an `if` statement or a `for` loop. JavaScript doesn’t have block scope. The following code snippet is incorrect in JavaScript:

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

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

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

This is a better approach because it’s closer to how JavaScript engines actually interpret the code.
**FUNCTIONS ARE VARIABLES**
You may have noticed throughout the preceding code snippets that the `addSurname` function has been declared as a variable. Again this is a best practice. First, this is how JavaScript sees it anyway, and second, it makes it very clear which scope the function is in.

While you can declare a function in this format:

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

JavaScript interprets this as follows:

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

So it’s 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 in various 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 aren’t “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 the ongoing name example to see how it looks by creating a `nameSetup` object in the global scope and use this to hold everything else. For example

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

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 you could have the following code snippet:

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

Invoking function and sending parameter

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.

D.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. The text assumes that you’re 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 the following code snippet is valid JavaScript:

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

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 isn’t ideal. More importantly, what happens if you want to add some extra lines within the if blocks? Let’s start by giving Sally a middle initial. See the following code snippet for how you might logically try this:

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

What went wrong here is that without the block braces only the first expression is considered part of the block, and anything following that’s outside of the block. So here, if firstname is Sally, initial becomes J, but surname always becomes Panayiotou.
The following code snippet shows the correct way of writing this:

```javascript
var firstname = 'Simon', surname, initial = '', fullname;
if (firstname === 'Simon') {
    surname = 'Holmes';
} else if (firstname === 'Sally') {
    initial = 'J';
    surname = 'Panayiotou';
}
fullname = firstname + ' ' + initial + ' ' + surname;
console.log(fullname);
```

By being prescriptive like this you’re now seeing what the JavaScript interpreter sees, and reducing the risk of unexpected errors. It’s a good aim to always make your code as explicit as possible, and 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 `===` is used to check for a match. This is not only a 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, such as number, string, and 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 code snippet for some interesting cases that could easily trip you up:

```javascript
var number = '';
number == 0;  // True
number === 0; // False
number = 1;
number == '1';  // True
number === '1'; // False
```

There are some situations where this might appear to be useful, but it’s 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:

```javascript
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 `!=`. 
**Good habits or bad habits**

**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.

First, like 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 code snippet shows some valid JavaScript that might not have the results you expect:

```javascript
for (var i = 0; i < 3; i++)
    console.log(i);
console.log(i*5);
// Output in the console
// 0
// 1
// 2
// 15
```

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 preceding snippet should be written like this:

```javascript
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 for declaring variables, you should never really see var inside the for conditional statement. Updating the previous code snippet to meet this best practice gives you the following:

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

As the variable declaration should be at the top of the scope, there could be many lines of code between it and the variable’s first use in a loop. JavaScript interpreters will act as though it has been defined there, so that’s where it should go.

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’s 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:

```
var myArray = ["one","two","three"];
```

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

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

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 because 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’s one thing you can do to improve the performance of this code. Each time the loop goes around 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 code snippet:

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

Now there’s a new variable `arrayLength` that’s 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.
D.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

- Syntactically correct JavaScript
- Ensuring your code functions correctly when minified
- Readability for yourself and/or others on 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’s still not a standard for the placement of tab stops. 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 of the statement starting the block. What? All of the code snippets so far have been written like this. The following code snippet shows the right way and the wrong way of doing it:

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

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?
What’s the 1% of the time when the wrong approach will cause you a problem? Consider the following code snippet using the return statement:

```javascript
return
{
    name : "name"
};
```

If you put your opening bracket onto a different line JavaScript will assume that you’ve missed a semicolon after the return command itself and add one in for you. So JavaScript evaluates it like this:

```javascript
return;
{
    name: "name"
};
```

Due to JavaScript’s semicolon insertion, you wouldn’t be returning the object you intended; instead it will return undefined. Let’s look at semicolon use and JavaScript semicolon insertion in more detail now.

**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 its own semicolons at runtime if it deems it necessary. This isn’t 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 semicolons as not optional, and I’m now 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:

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

But code blocks shouldn’t 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

```javascript
if (firstname === 'Simon') {
    ...
}
function addSurname() {
    ...
}
```
for (i = 0; i < 3; i++) {
    ...
}

The rule isn’t quite so straightforward as “don’t use a semicolon” after curly braces. When assigning a function or object to a variable you do have a semicolon after the curly braces. You saw a couple of examples just above, and we’ve been using them throughout the book so far.

```javascript
var addSurname = function() {
    ...
};
var nameSetup = {
    firstname : 'Simon'
};
```

This can take a little while to get used to, but it’s 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 is to write one variable name per line. This makes it really easy to see at a glance what variables you’ve set up. The classic placement for the comma separating these is at the end of the line, like in the following code snippet:

```javascript
var firstname = 'Simon',
    surname,
    initial = '',
    fullname;
```

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:

```javascript
var firstname = 'Simon'
    , surname
    , initial = ''
    , fullname;
```

This is perfectly valid JavaScript, and when minified down to one line will read exactly the same as the first code 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 that is what is 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. The critical thing is to have a standard and stick to it.
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 whitespace from between a lot of JavaScript operators. Take a look at the following code snippet, showing the same piece of code with and without extra whitespace:

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

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

As humans we read using whitespace as the delimiters for words, and the way we read code is no different. Yes, you can figure out the second part of the code snippet here, as there are many syntactic pointers to act as delimiters, but it’s 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.

TOOLS TO HELP YOU WRITE GOOD JAVASCRIPT
There are a couple of online code quality checkers called 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.

D.3 Getting to know JSON
JavaScript Object Notation (JSON) is a JavaScript-based approach to data exchange. It’s 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 manners of programming languages.

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

D.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’re 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 you’ve 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:

- The key must be a string.
- The string must be wrapped in double quotes if it’s a JavaScript reserved word or an illegal JavaScript name.

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

Listing D.2 An example of a JavaScript object literal

```javascript
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'
    }]
};
```

Here all keys in the object are strings, but the values are a mixture of types: string, number, Boolean, null, function, and array.

Accessing the properties of an object literal

The preferred way of accessing properties is using the dot notation (.) Examples 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 are

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

Again, these references can be used to get or set the values. After that quick recap of object literals, let’s take a look at how JSON is related.
D.3.2 Differences with JSON

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

- All key names and strings must be wrapped in double quotes.
- Functions aren’t 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

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

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

Looking at this list and comparing it to the JavaScript object in listing D.2, if you remove the function property you should be able to convert it to JSON.

Formatting JSON data

Unlike the JavaScript object, we’re not assigning the data to a variable, 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 the following listing.

Listing D.3 An example of correctly formatted JSON

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

With JSON you can send strings

Empty strings

Boolean values

Arrays of other JSON objects
This listing shows some valid JSON. This data can be exchanged between applications and programming languages without issue. It’s 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 it will most likely throw an error when the next item isn’t valid JSON.

The following code snippet shows an example of this. There are two double quotes inside the string, which isn’t 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 the following:

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

This escape character tells JSON interpreters that the following character shouldn’t be considered part of the code; it’s part of the value and can be ignored.

**Shrinking JSON for transporting across the internet**

The spacing and indentation in listing D.3 is purely to aid human readability. Programming languages don’t need it, and you can reduce the amount of information being transmitted if you remove unnecessary whitespace before sending the code.

The following code snippet shows a minimized version of listing D.3, which is more along the lines of what you’d expect to exchange between applications:

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

The content of this is exactly the same as listing D.3, just much more compact.

**D.3.3 Why is JSON so good?**

The popularity of JSON as a data exchange format predates the development of Node 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 you’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 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 there are no data transformations to worry about.

D.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 mouse-clicking on a link, data being written to a database, or just another piece of code finishing 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’s passed directly into the 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!

D.4.1 Use setTimeout to run code later

Most of the time you use callbacks to run code after something has happened. For getting accustomed to the concept you can use a function that’s 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’s as follows:

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

First setTimeout is declared inside a variable so that you can access it again to cancel it should you want to. As we talked about earlier, a callback is typically an unnamed anonymous function. So if you wanted to log your name to the JavaScript console after two seconds you could use this code snippet:

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

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?

```javascript
console.log("Hello, what's your name?");
var waitForIt = setTimeout(function(){
    console.log("My name is Simon Holmes");
}, 2000);
clearTimeout(waitForIt);

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.

D.4.2 Asynchronous code

Before we look at some more code, let’s start by reminding ourselves of the bank teller analogy from chapter 1. Figure D.3 shows how a bank teller can deal with multiple requests by passing on any time-consuming tasks to other people.

The bank teller is able to respond to Sally’s request because she 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.
You can mimic this in JavaScript, using the `setTimeout` function to demonstrate the asynchronous approach. All you 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 code snippet, where it’s assumed that Simon’s request will take three seconds (3,000 milliseconds) and Sally’s will take one second:

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

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

console.log("Free to take another request");  // 3 Ready for another request
```

Figure D.3 Handling multiple requests
This code has three distinct blocks: taking the first request from Simon and sending it away ①; taking the second request from Sally and sending it away ②; and ready to take another request ③. If this were synchronous code like you’d see in PHP or .NET, you’d first deal with Simon’s request in its entirety before even taking Sally’s request three seconds later.

With 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 an HTML page and run it, or enter it directly into the 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 the bank teller 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.

### D.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 you can create a function that uses a callback. 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 code snippet demonstrates this (note that you’ll not be able to run this in a JavaScript console):

```javascript
var setTimeout = function(callback, delay){
  ...
  ...
  callback();
};
var requestB = setTimeout(function(){
  console.log("Sally: Here’s your $100");
}, 1000);
```

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 you’re passing it a simple anonymous
function \( \texttt{c} \) 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**

Something to bear in mind when passing anonymous functions around like this is that the callback doesn’t inherit the scope of the function it’s passed into. The callback function isn’t declared inside the destination function, merely invoked from it. A callback function inherits the scope in which it’s defined.

Let’s return to the idea of scope circles and look at this visually in figure D.4.
Here you can see that the callback has its own local scope inside the global scope, as that’s where requestB is defined. This is all very well and good if your callback will only need access to its inherited scope, but what if you want it to be smarter? What if you want to use data from your asynchronous function in your callback?

Currently the example callback function has a dollar amount hard-coded into it, but what if you want that value to be dynamic, to be a variable? Assuming this value is set in the setTimeout function, how do you get it into the callback? You could save it to the global scope, but as you know by now this would be bad. So you need to pass it as a parameter into the callback function. This should give you something like the scope circles shown in figure D.5.

Doing the same thing in code would look like the following:

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

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

This code snippet will output the same message to the console that you’ve already seen. The big difference now is that the value of dollars is being set in the setTimeout function and being passed to the callback.
It’s important that you understand this approach, as the vast majority of Node code examples on the internet use asynchronous callbacks in this way. But there are a couple of potential problems with this approach, particularly when your codebase gets larger and more complex. Primarily, an overreliance on passing around anonymous callback functions can make the code hard to read and follow, especially when you find you have multiple nested callbacks. It also makes it difficult to run tests on the code, as you can’t actually call any of these functions by name—they are all anonymous. We aren’t covering unit testing in this 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 you can achieve this, using named callbacks.

### D.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 inside a defined function. Then rather than passing the code directly as an anonymous function, you can pass the function name. So rather than passing the code you’re passing a *reference* to the code to run. Sticking with the ongoing example, add a new function `onCompletion` that will be the callback function. Figure D.6 shows how this looks in the scope circles.

This looks just like the previous example, except that the callback scope now has a name. As with an anonymous callback, a named callback can be invoked without any parameters, as is implied in figure D.6.

---

![Figure D.6 Change in scope when using a named callback](image-url)
The following code snippet shows how to declare and invoke a named callback, putting into code what you see in figure D.6:

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

var onCompletion = function(){
  console.log("Sally: Here's your $100");
};

var requestB = setTimeout(
  onCompletion,
  1000);
```

The named function 1 now exists as an entity in its own right, creating its own scope. Notice how there’s no longer an anonymous function, but instead the name of the function 2 is passed as a reference.

**PASSING VARIABLES ACROSS**

The previous code snippet uses a hard-coded dollar value in the console log again. Like with anonymous callbacks, passing a variable from one scope to another is pretty straightforward. You can simply pass the parameters you need into the named function. Figure D.7 shows how this looks in the scope circles.

![Figure D.7 Passing the required parameter into the new function scope](image-url)
So you just need to pass the variable `dollars` from `setTimeout` to the `onCompletion` callback function. This is done without changing anything in your request; take a look at the following code snippet:

```javascript
var setTimeout = function(callback, delay){
  ... dollars = 100;
  callback(dollars);  // Send dollars variable as a parameter to callback
};

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

var requestB = setTimeout(
  onCompletion,
  1000);
```

Here the `setTimeout` function is sending the `dollars` variable to the `onCompletion` function as a parameter. You’ll often have no control over the parameters sent to your callback, because asynchronous functions like `setTimeout` are generally provided as is. But you’ll often want to use variables from other scopes inside your callback, and not just what your asynchronous function provides. Let’s look at how to send the parameters you want to your callback.

**USING VARIABLES FROM A DIFFERENT SCOPE**

Let’s say, for example, that you want the name in the output to come through as a parameter. So the updated function will look like the following:

```javascript
var onCompletion = function(dollars,name){
  console.log(name + ": Here's your $\" + dollars);
};

var requestB = setTimeout(
  function(dollars){
    onCompletion(dollars,name);
  }, 1000);

getMoney('Simon');
```

The problem is that the `setTimeout` function will only pass a single parameter, `dollars`, to the callback. You can address this by using an anonymous function as a callback again, remembering that it will inherit the scope in which it’s defined. To demonstrate this outside of the global scope, wrap the request in a new function, `getMoney`, that accepts a single parameter, `name`:

```javascript
var getMoney = function (name) {
  var requestB = setTimeout(function(dollars){
    onCompletion(dollars,name);
  }, 1000);
};

getMoney('Simon');
```

In the scope circles this looks like figure D.8.
Understanding JavaScript callbacks

And putting all the code together for the sake of completeness gives you the following:

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

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

var getMoney = function(name) {
    var requestB = setTimeout(function(dollars) {
        onCompletion(dollars, name);
    }, 1000);
};;
getMoney('Simon');
```

Figure D.8  The process of sending variables from different scopes to a named callback function
The simple way to think of it is that calling the named function from inside the anonymous callback like this enables you to capture anything you need from the parent scope (getMoney in this case) and explicitly pass it to the named function (onCompletion).

**Seeing the flow in action**

If you want to see this flow in action you can add a debugger statement and run it in your browser and step through the functions and see which variables and values are set where and when. All together it will be something like this:

```javascript
var mySetTimeout = function(callback, delay){
    var dollars = 100;
    callback(dollars);
};

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

var getMoney = function (name) {
    debugger;
    var requestB = mySetTimeout(function(dollars){
        onCompletion(dollars,name);
    }, 1000);
};

getMoney('Simon');

Note that as well as adding in a debugger statement you’ll want to change the name of the setTimeout function so that it doesn’t interfere with the native function.

Remember that you normally won’t have access to the code inside the 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 extra you need to add must be added inside the anonymous 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 think that the flow is harder to understand, by moving the code into its own function. And you could well have a point! But when the code becomes more complex, and you have multiple lines of code inside multiple nested callbacks, you’ll 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 you can test it more easily. For example, you can now say “when the function onCompletion is passed a number of dollars and a name it should output a message to the console including this number and name.” 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 you’ve got a good idea of how callbacks are defined and used, let’s look at Node and see why callbacks are so useful.

D.4.5 Callbacks in Node

In the browser a lot of 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’s generally an individual user who initiates the event, and it’s 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 discussed in chapter 1, there’s 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 gets held up. This isn’t a good thing! And this is why it’s important to understand callbacks, because 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 CALLBACK

Using a callback in Node isn’t really any different than using it in the browser. If you want to save some data, you don’t want the main Node process doing this, just like you didn’t want the bank teller going with the safe manager and waiting for the response. So you want to use an asynchronous function with a callback. All database drivers for Node provide this ability; we’ll get into the specifics about how to create and save data in the book, so for now we’ll just use a simplified example. The following code snippet shows an example of asynchronously saving data mySafe and outputting a confirmation to the console when the database finishes and returns a response:

```javascript
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’s normally a bit more to it than this, but the basic construct is fairly simple.
RUNNING CALLBACKS ONE AFTER ANOTHER

So that’s great. You get the idea of running a callback, but what do you do if you want to run another asynchronous operation when the callback is finished? Returning to the banking metaphor, say, for example, you 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 you’re looking to create a flow like that shown in figure D.9.

Now that it’s clear that this will require two operations, you can see that this is going to need another asynchronous call to the database. You know from what we’ve already discussed that you can’t just put it in the code after the `save` function, like in the following code snippet:

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

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, because it won’t take into account the value being added to the safe. So you need to ensure that the second operation runs when you know the first one has finished. The solution is simple: you 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 put the second function inside the callback from the first, as in the following example:

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

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

This ability is very important. Node is inherently asynchronous, jumping from request to request, from site visitor to site visitor. But sometimes you need to do things in a sequential manner, one thing after another. Nesting callbacks gives you 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 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 callbacks. The problem is so great that it has become known as *callback hell*. This is actually a reason why some people think that Node 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’s easy to end up in callback hell when developing Node, 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’s using named callbacks. So let’s take a look at how named callbacks help 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’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:

```javascript
myData.save().
    .then(myAccounts.findTotal).
    .then(doSomethingElseAsynchronous)
```
USING NAMED CALLBACKS TO AVOID CALLBACK HELL
Named callbacks can help avoid nested callback hell because 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 you need to take the content of a callback function and declare it as a separate function. In the nested callback example there are two callbacks, so you’re going to need two new functions, one for when the mySafe.save operation has completed, and one for when the myAccounts.findTotal operation has completed. If these call onSave and onFindTotal respectively, you can create some code like the following:

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

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

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

Now that each piece of functionality is separated out into a separate function, it’s easier to look at each part in isolation and understand what it’s doing. You can see what parameters it expects and what the outcomes should be. In reality, the outcomes are likely to be 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, 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.

D.5 Writing modular JavaScript

A couple of days ago someone anonymously 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.

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’ll 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 don’t 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.

D.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 code snippet demonstrates how you can send a value to a function and then later retrieve it:

```javascript
var user = {};

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

user.age = setAge(30);
console.log(user.age);
console.log(user.age.getAge());
```

Okay, so as well as sounding a bit weird, it possibly also looks a bit weird! But here’s what’s 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 its own scope. Nothing outside of this function has access to the scope.
So `myAge` is also not a one-off shared variable. You can call the function again—creating a second new function scope—to set (and get) the age of a second user. You could happily run the following code snippet after the one we’ve just looked at, creating a second user and giving them a different age:

```javascript
var usertwo = {};
usertwo.age = setAge(35);
console.log(usertwo.age.getAge());
console.log(user.age.getAge());
```

Each user has a different age that isn’t 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 it has evolved into more useful patterns. We’ll start by looking at the module pattern.

### D.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’s self-contained, and only uses data explicitly passed into it, and only reveals data that’s 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’ve been using in this book up until now have been function declarations, creating functions that we can call upon later in the code. The IIFE creates 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 code snippet).

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

This is the typical use, but not the only one. The IIFE has been assigned to a variable. When you do this, the returned methods from the function become properties of the variable.
This is made possible by using an IIFE (see the previous sidebar for a bit more information on IIFE). Like the basic closure, the module pattern returns functions and variables as properties of the variable it’s assigned to. Unlike the basic closure, the module pattern doesn’t have to be manually initiated; the module immediately calls itself as soon as it has been defined.

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

```javascript
var user = {firstname: "Simon"};
var userAge = (function () {
    var myAge;
    return {
        setAge: function (initAge) {
            myAge = initAge;
        },
        getAge: function () {
            return myAge;
        }
    };
})();

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

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 code snippet you just get and set, but it’s of course possible to modify. You 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 code snippet, with the new parts in bold:

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

Assign module to variable
Define variable in module scope
Define method to be returned that can take parameter and modify module variable
Define method to be returned that can access module variable
Call methods set and get for the module variable
Outputs “30”

New method to increment myAge by 1 and return new value
userAge.setAge(30);
user.age = userAge.getAge();
console.log(user.age);
user.age = userAge.happyBirthday();
console.log(user.age);
user.age = userAge.getAge();
console.log(user.age);

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.

D.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 the module pattern. The aim is to make really obvious what is exposed as public and what remains as private to the module.

Underscore variable naming convention

The first aspect of this approach is really just a convention. There’s 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

```javascript
var _privateVariable;
var publicVariable;
```

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 a break. When you use this approach, the return statement just contains a list of the functions that you’re returning, without any of the actual code. The code is all declared in functions above the return statement, although within the same module, of course. The following code snippet shows an example of this:

```javascript
var userAge = (function () {
    var _myAge;
    var setAge = function (initAge) {
        _myAge = initAge;
    };
    return {
        setAge: setAge
    };
})();
```
return {
  setAge: setAge,
  getAge: getAge,
  happyBirthday: happyBirthday
};
})();

You can’t really see the benefit of this approach in such a small example. We’ll look at a longer example shortly that will get you 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 it 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’s 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 aren’t 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. The following listing shows an example of the revealing module pattern, using the underscore convention, and removing code from the return statement.

```javascript
var user = {};
var userAge = (function () {
  var _myAge;
  var setAge = function (initAge) {
    _myAge = initAge;
  };
  var getAge = function () {
    return _myAge;
  };
  var _addYear = function () {
    _myAge = _myAge + 1;
  };
  var happyBirthday = function () {
    _addYear();
    return _myAge;
  };
  return {
    setAge: setAge,
    getAge: getAge,
    happyBirthday: happyBirthday
  };
})();
```

- Has underscore as it’s never directly exposed outside of module
- Private function that isn’t exposed
- Can be called by a public function that’s exposed
- Return statement acts as reference for exposed methods
Listing D.4 demonstrates a few interesting things. First, notice that the variable `_myAge` 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 noted with `_addYear` in the listing. Private functions can easily be called by public methods, the underscore again denoting that it’s not itself exposed.

The `return` statement 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’re 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’re doing you could also create a nested module, or even a separate module with a 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._

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

### D.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’s wrong. This isn’t really acceptable for good code, so it’s 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’s global scope and function scope. There are no other types of scope in JavaScript. You really want to try to 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 it can be difficult to maintain if you’re not careful.

JSON is born of JavaScript but isn’t JavaScript; it’s 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 isn’t required to interpret JSON.
Callbacks are vital to running successful Node applications, because they allow the central process to effectively delegate tasks that could hold it up. To put it another way, callbacks enable you to use sequential synchronous operations in an asynchronous environment. But callbacks aren’t without their problems. It’s 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, you can use named callbacks to address this problem on all levels, so long as you remember that named callbacks don’t inherit scope like their inline anonymous counterparts.

Closures and module patterns provide ways to write code that’s 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 toward the revealing module pattern, which is convention-driven to draw specific lines between what’s private and what’s 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.
Traditional web dev stacks use a different programming language in every layer, resulting in a complex mashup of code and frameworks. Together, the MongoDB database, the Express and AngularJS frameworks, and Node.js constitute the MEAN stack—a powerful platform that uses only one language, top to bottom: JavaScript. Developers and businesses love it because it’s scalable and cost-effective. End users love it because the apps created with it are fast and responsive. It’s a win-win-win!

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

What’s Inside

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

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

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

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

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

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

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

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