LEARNING Node.js

A Hands-On Guide to Building Web Applications in JavaScript®

SECOND EDITION

MARC WANDSCHNEIDER

Learning Node.js

Second Edition

Learning Node.js

Second Edition

Marc Wandschneider

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi
Mexico City • São Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Learning Node.js, Second Edition

ISBN-13: 978-0-134-66370-8

ISBN-10: 0-134-66370-5

Library of Congress Control Number: 2016956520

Printed in the United States of America

First printing: December 2016

Trademarks

Warning and Disclaimer

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Editor
Mark Taber

Project Manager
Dhayanidhi

Copy Editor
Warren Hapke

Indexer
Cheryl Lenser

Technical Reviewer
Gustavo Moreira

Cover Designer
Chuti Prasertsith

❖

Much love to Tina, for simply being there.

❖

Contents at a Glance

Contents

Acknowledgments

I'd like to thank all the Marks at Pearson (it's a common name, it seems) who have helped me make this book and other projects a reality. The copy editors have been brilliant and helpful.

A huge debt of gratitude is due to Gustavo Moreira for his excellent technical and style reviews.

And finally, much love to Tina, for making it all worthwhile.

About the Author

Marc Wandschneider is a software developer who has spent much time working on scalable web applications and responsive mobile apps. A graduate of McGill University's School of Computer Science, he spent five years at Microsoft, developing and managing developers on the Visual Basic, Visual J++, and .NET Windows Forms teams. As a Software Developer/Architect at SourceLabs, he built the SWiK open source Wiki platform and then co-founded Adylitica in Beijing. He currently works for Google in London. He authored *PHP and MySQL LiveLessons and Core Web Application Development with PHP and MySQL*.

Introduction

Welcome to *Learning Node.js*. Node.js is an exciting platform for writing applications of all sorts, ranging from powerful web applications to simple scripts you can run on your local computer. The project has grown from a reasonably small software package managed by one company into a production-quality system governed by a Technical Steering Committee (TSC) and has a sizeable following in the developer community. In this book, I teach you more about it and why it is special, then get you up and writing Node.js programs in short order. You'll soon find that people are rather flexible with the name of Node.js and will refer to it frequently as just Node or even node. I certainly do a lot of that in this book as well.

Why Node.js?

Node.js has arisen for a couple of primary reasons, which I explain next.

The Web

In the past, writing web applications was a pretty standard process. You have one or more servers on your machine that listen on a *port* (for example, *80* for HTTP), and when a request is received, it forks a new process or a thread to begin processing and responding to the query. This work frequently involves communicating with external services, such as a database, memory cache, external computing server, or even just the file system. When all this work is finally finished, the thread or process is returned to the pool of available servers, and more requests can be handled.

It is a reasonably linear process, easy to understand and straightforward to code. There are, however, a couple of disadvantages that continue to plague the model:

1. Each of these threads or processes carries some overhead with it. On some machines, PHP and Apache can take up as much as 10–15MB per process. Even in environments where a large server runs constantly and forks threads to process the requests, each of these carries some overhead to create a new stack and execution environment, and you frequently run into the limits of the server's available memory.
2. In most common usage scenarios where a web server communicates with a database, caching server, external server, or file system, it spends most of its time sitting around doing nothing and waits for these services to finish and return their responses. While it is sitting there doing nothing, this thread is effectively blocked from doing anything else. The resources it consumes and the process or thread in which it runs are entirely frozen waiting for those responses to come back.

Only after the external component has finally sent back its response will that process or thread be free to finish processing, send a response to the client, and then reset to prepare for another incoming request.
So, although it's pretty easy to understand and work with, you do have a model that can be quite inefficient if your scripts spend most of their time waiting for database servers to finish running a query—an extremely common scenario for many modern web applications.

Many solutions to this problem have been developed and are in common use. You can buy ever bigger and more powerful web servers with more memory. You can replace more powerful and feature-rich HTTP servers such as Apache with smaller, lightweight ones such as *lighttpd* or *nginx*. You can build stripped-down or reduced versions of your favorite web programing language such as PHP or Python (indeed, for a time, Facebook took this one step further and built a system that converts PHP to native C++ code for maximal speed and optimal size). Or you can throw more servers at the problem to increase the number of simultaneous connections you can accommodate.

New Technologies

Although the web developers of the world have continued their eternal struggle against server resources and the limits on the number of requests they can process, a few other interesting things have happened in the world.

JavaScript, that old (meaning 1995 or so) language that came to be known mostly (and frequently reviled) for writing client-side scripts in the web browser, has become hugely popular. Modern versions of web browsers are cleaning up their implementations of it and adding new features to make it more powerful and less quirky. With the advent of client libraries for these browsers, such as jQuery, script.aculo.us, or Prototype, programming in JavaScript has become fun and productive. Unwieldy APIs have been cleaned up, and fun, dynamic effects have been added.

At the same time, a new generation of browser competition has erupted, with Google's Chrome, Mozilla's Firefox, Apple's Safari, and Microsoft's Edge all vying for the crown of browser king. As part of this, all these companies are investing heavily in the JavaScript portion of these systems as modern web applications continue to grow ever-more dynamic and script-based. In particular, Google Chrome's V8 JavaScript runtime is particularly fast and also open sourced for use by anybody.

With all these things in place, the opportunity arose for somebody to come along with a new approach to network (web) application development. Thus, the birth of Node.js.

What Exactly Is Node.js?

In 2009, a fellow named Ryan Dahl was working for Joyent, a cloud and virtualization services company in California. He was looking to develop push capabilities for web applications, similar to how Gmail does it, and found most of what he looked at not quite appropriate. He eventually settled on JavaScript because it lacked a robust input/output (I/O) model (meaning he could write his own new one), and had the fast and fully programmable V8 runtime readily available.

Inspired by some similar projects in the Ruby and Python communities, he eventually took the Chrome V8 runtime and an event-processing library called *libev* and came up with the first versions of a new system called *Node.js*. The primary methodology or innovation in Node.js is that it is built entirely around an event-driven, nonblocking model of programming. In short, you never (well, rarely) write code that blocks.

If your web application—in order to process a request and generate a response—needs to run a database query, it runs the request and then tells Node.js what to do when the response returns. In the meantime, your code is free to start processing other incoming requests or, indeed, do any other task it might need, such as cleaning up data or running analyses.

Through this simple change in the way the application handles requests and work, you are able to easily write web servers that can handle hundreds, if not thousands, of requests simultaneously on machines without much processing or memory resources. Node runs in a single process, and your code executes largely in a single thread, so the resource requirements are much lower than for many other platforms.

This speed and capacity come with a few caveats, however, and you need to be fully aware of them so you can start working with Node with your eyes wide open.

First and foremost, the new model is different from what you may have seen before and can sometimes be a bit confusing. Until you've wrapped your brain fully around some of the core concepts, some code you see written in Node.js can seem a bit strange. Much of this book is devoted to discussing the core patterns many programmers use to manage the challenges of the asynchronous, nonblocking way of programming that Node uses and how to develop your own.

Another limitation with this model of programming is that it really is centered around applications that are doing lots of different things with lots of different processes, servers, or services. Node.js truly shines when your web application is juggling connections to databases, caching servers, file systems, application servers, and more. The flip side of this, however, is that it's actually not necessarily an optimal environment for writing compute servers that are doing serious, long-running computations. For these, Node's model of a single thread in a single process can create problems if a given request is taking a ton of time to generate a complicated password digest or processing an image. In situations in which you're doing more computationally intensive work, you need to be careful how your applications use resources or perhaps even consider farming those tasks out to other platforms and run them as a service for your Node.js programs to call.

Node.js's path to adulthood has been a somewhat rocky one—the 0.x series of Node.js lingered for quite a while, releasing often but seemingly not making much progress, and some grew impatient with the governance of the project. This caused a schism in late 2014, with a group of people forking the open sourced code and creating io.js, a new version of node with the goals of being more open and transparent and responsive to the developer community. Fortunately, this break did not last long, and within nine months, Joyent agreed to hand over guidance of Node.js to the Technical Steering Committee (TSC) in autumn 2015.

Today, however, the platform is quite stable and predictable, and has adopted *semantic versioning*, where your versionsversion numbers have the format *major.minor.patchlevel*. In this model you only make breaking API changes with major version number changes, add features in minor version number changes, and can update and fix anything necessary in patch-level changes. Each major version is developed for 18 months and then supported for another 12 months after that, meaning you have 2.5 years of use for each version. After that, you'll need (and definitely want) to migrate to the latest version to be sure you're getting the latest features and most secure version of the software).

To help you keep track of and manage all of these updates, the developers have taken to labeling portions of the system with different degrees of *stability*, ranging from *Unstable* to *Stable* to *Locked*. Changes to *Stable* or *Locked* portions of the runtime are rare and involve much community discussion to determine whether the changes will generate too much pain. As you work your way through this book, we point out which areas are less stable than others and suggest ways you can mitigate the dangers of changing APIs. Newer versions of Node.js have introduced the concept of *Deprecated* APIs. If part of Node.js is becoming too difficult to maintain and is not heavily used, or otherwise doesn't make sense to continue supporting, it will (again, after much community discussion) be marked as *Deprecated* and not included in the next major version update. This gives developers plenty of time to move to alternatives (of which there are always going to be dozens).

The good news is that Node.js already has a large and active user community and a bunch of mailing lists, forums, and user groups devoted to promoting the platform and providing help where needed. A simple Internet search will get you answers to 99 percent of your questions in a matter of seconds, so never be afraid to look!

Who Is This Book For?

I wrote this book under the assumption that you are comfortable programming computers and are familiar with the functionality and syntax of at least one major programming language such as Java, C/C++, PHP, or C#. Although you don't have to be an expert, you've probably moved beyond "Learn X in Y days" level tasks.

If you're like me, you have probably written some HTML/CSS/JavaScript and thus have "worked with" JavaScript, but you might not be intimately familiar with it and have just largely templated heavily off code found on blog posts or mailing lists. Indeed, because of its clunky UI and frustrating browser mismatches, you might even frown slightly at the mere mention of JavaScript. Fear not—by the end of the first section of this book, distasteful memories of the language will be a thing of the past and, I hope, you'll be happily writing your first Node.js programs with ease and a smile on your face!

I also assume that you have a basic understanding of how web applications work: browsers send HTTP requests to a remote server; the server processes each request and sends a response with a code indicating success or failure, and then optionally some data along with that response (such as the HTML for the page to render or perhaps JavaScript Object Notation, or JSON, containing data for that request). You've probably connected to database servers in the past, run queries, and waited for the resulting rows, and so on. When I start to describe concepts beyond these in the samples and programs, I explain and refresh everybody's memory on anything new or uncommon.

How to Use this Book

This book is largely tutorial in nature. I try to balance out explanations with code to demonstrate it as much as possible and avoid long, tedious explanations of everything. For those situations in which I think a better explanation is interesting, I might point you at some resources or other documentation to learn more if you are so inclined (but it is never a necessity).

The book is divided into four major sections:

Part 1. **Learning to Walk**—You start installing and running Node, take another look at the JavaScript language and the extensions used in V8 and Node.js, and then write your first application.

Part 2. **Learning to Run**—You start developing more powerful and interesting application servers in this part of the book, and I start teaching you some of the core concepts and practices used in writing Node.js programs.

Part 3. **Breaking Out the Big Guns**—In this section, you look at some of the powerful tools and modules available to you for writing your web applications, such as help with web servers and communication with database servers.

Part 4. **Getting the Most Out of Node.js**—Finally, I close out the book by looking at a few other advanced topics such as ways in which you can run your applications on production servers, how you can test your code, and how you can use Node.js to write command-line utilities as well!

As you work your way through the book, take the time to fire up your text editor and enter the code, see how it works in your version of Node.js, and otherwise start writing and developing your own code as you go along. You develop your own little photo sharing application as you work through this book, which I hope provides you with some inspiration or ideas for things you can write.

Download the Source Code

Source code for most of the examples and sample projects in this book can be found at *github.com/marcwan/LearningNodeJS*. You are highly encouraged to download it and play along, but don't deny yourself the opportunity to type in some of the code as well and try things out.

The GitHub code has some fully functioning samples and has been tested to work on Mac, Linux, and Windows with the latest versions of Node.js. If new updates of Node require updates to the source code, I will put changes and notes there, so please be sure to pull down new versions every few months. Sadly, my code is not perfect, and I always welcome bug reports and pull requests!

If you have any questions or problems with the code in this book, feel free to go to *github.com/marcwan/LearningNodeJS* and add an issue; they'll be monitored and answered reasonably quickly.

This page intentionally left blank

Part I

Learning to Walk

This page intentionally left blank

3

Asynchronous Programming

Now that you have a refreshed and updated idea of what JavaScript programming is really like, it's time to get into the core concept that makes Node.js what it is: *nonblocking IO and asynchronous programming*. It carries with it some huge advantages and benefits, which you shall soon see, but it also brings some complications and challenges with it.

The Old Way of Doing Things

In the olden days (2008 or so), when you sat down to write an application and needed to load in a file, you would write something like the following (let's assume you're using something vaguely PHP-ish for the purposes of this example):

```
$file = fopen('info.txt', 'r');
// wait until file is open

$contents = fread($file, 100000);
// wait until contents are read

// do something with those contents
```

If you were to analyze the execution of this script, you would find that it spends a vast majority of its time *doing nothing at all*. Indeed, most of the clock time taken by this script is spent waiting for the computer's file system to do its job and return the file contents you requested. Let me generalize things a step further and state that for most IO-based applications—those that frequently connect to databases, communicate with external servers, or read and write files—your scripts will spend a majority of their time sitting around waiting (see Figure 3.1).

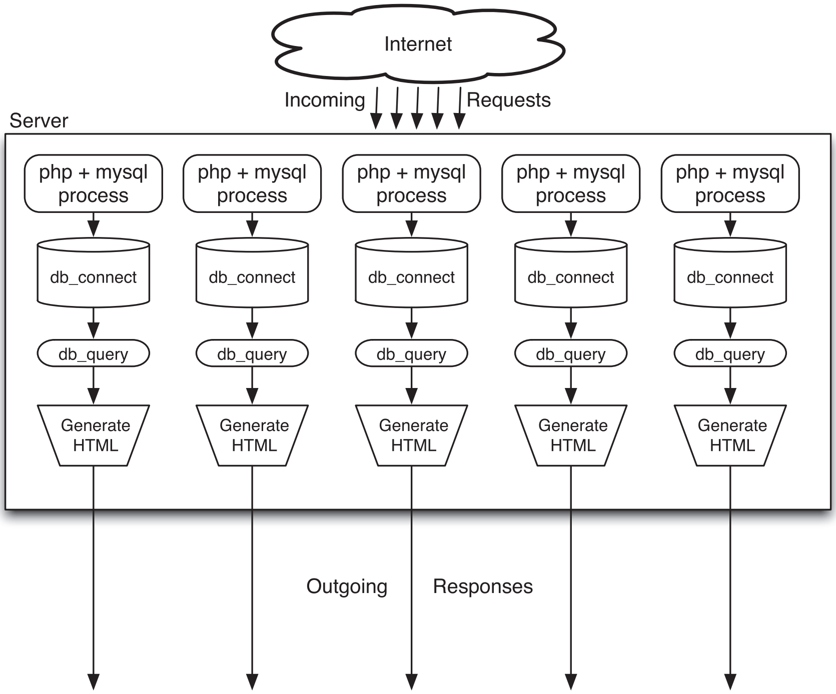

Figure 3.1 Traditional blocking IO web servers

The way your servers process multiple requests at the same time by running many of these scripts in parallel. Modern computer operating systems are great at multitasking, so you can easily switch out processes that are blocked and let other processes have access to the CPU. Some environments take things a step further and use threads instead of processes.

The problem is that for each of these processes or threads, there is some amount of overhead. For heavier implementations using Apache and PHP, I have seen up to 10–15MB of memory overhead per process—never mind the resources and time consumed by the operating system switching that context in and out constantly. That's not even 100 simultaneously executing servers per gigabyte of RAM! Threaded solutions and those using more lightweight HTTP servers do, of course, have better results, but you still end up in a situation in which the computer spends most of its time waiting around for blocked processes to get their results, and you risk running out of capacity to handle incoming requests.

It would be nice if there were some way to make better use of all the available CPU power and available memory so as not to waste so much. This is where Node.js shines.

The Node.js Way of Doing Things

To understand how Node.js changes the method demonstrated in the preceding section into a nonblocking, asynchronous model, first look at the `setTimeout` function in JavaScript. This function takes a function to call and a timeout after which it should be called:

```
setTimeout(() => {
    console.log("I've done my work!");
}, 2000);

console.log("I'm waiting for all my work to finish.");
```

If you run the preceding code, you see the following output:

```
I'm waiting for all my work to finish.
I've done my work!
```

I hope this is not a surprise to you: The program sets the timeout for *2000 ms* (2 seconds), giving it the function to call when it fires, and then continues with execution, which prints out the "I'm waiting…" text. Two seconds later, you see the "I've done…" message, and the program then exits.

Now, look at a world where any time you call a function that needs to wait for some external resource (database server, network request, or file system read/write operation), it has a similar signature. That is, instead of calling `fopen(path, mode)` and waiting, you would instead call `fopen(path, mode, (file_handle) => { ... })`.

Now rewrite the preceding synchronous script using the new asynchronous functions. You can actually enter and run this program with `node` from the command line. Just make sure you also create a file called *info.txt* that can be read.

```
var fs = require('fs');                          // We'll explain this below

var file;
var buf = new Buffer(100000);

fs.open('info.txt', 'r', (err, handle) => {
    file = handle;
});

// fs.read needs the file handle returned by fs.open. But this is broken.
fs.read(file, buf, 0, 100000, null, (err, length) => {
    console.log(buf.toString());
    fs.close(file, () => { /* don't care */ });
});
```

The first line of this code is something you haven't seen just yet: the `require` function is a way to include additional functionality in your Node.js programs. Node comes with a pretty impressive set of *modules*, each of which you can include separately as you need functionality. You will work further with modules frequently from now on; you learn about consuming them and writing your own in Chapter 5, "Modules."

If you run this program as it is, it throws an error and terminates. How come? Because the `fs.open` function runs *asynchronously;* it returns immediately, before the file has been opened and the callback function invoked. The `file` variable is not set until the file has been opened and the handle to it has been passed to the callback specified as the third parameter to the `fs.open` function. Thus, you are trying to access an `undefined` variable when you try to call the `fs.read` function with it immediately afterward.

Fixing this program is easy:

```
var fs = require('fs');

fs.open('info.txt', 'r', (err, handle) => {
    var buf = new Buffer(100000);
    fs.read(handle, buf, 0, 100000, null, (err, length) => {
        console.log(buf.toString('utf8', 0, length));
        fs.close(handle, () => { /* Don't care */ });
    });
});
```

The key way to think of how these asynchronous functions work internally in Node is something along the following lines:

- Check and validate parameters.
- Tell the Node.js core to queue the call to the appropriate function for you (in the preceding example, the operating system `open` or `read` function) and to notify (call) the provided callback function when there is a result.
- Return to the caller.

You might be asking: if the `open` function returns right away, why doesn't the `node` process exit immediately after that function has returned? The answer is that Node operates with an *event queue;* if there are pending events for which you are awaiting a response, it does not exit until your code has finished executing *and* there are no events left on that queue. If you are waiting for a response (either to the `open` or the `read` function calls), it waits. See Figure 3.2 for an idea of how this scenario looks conceptually.

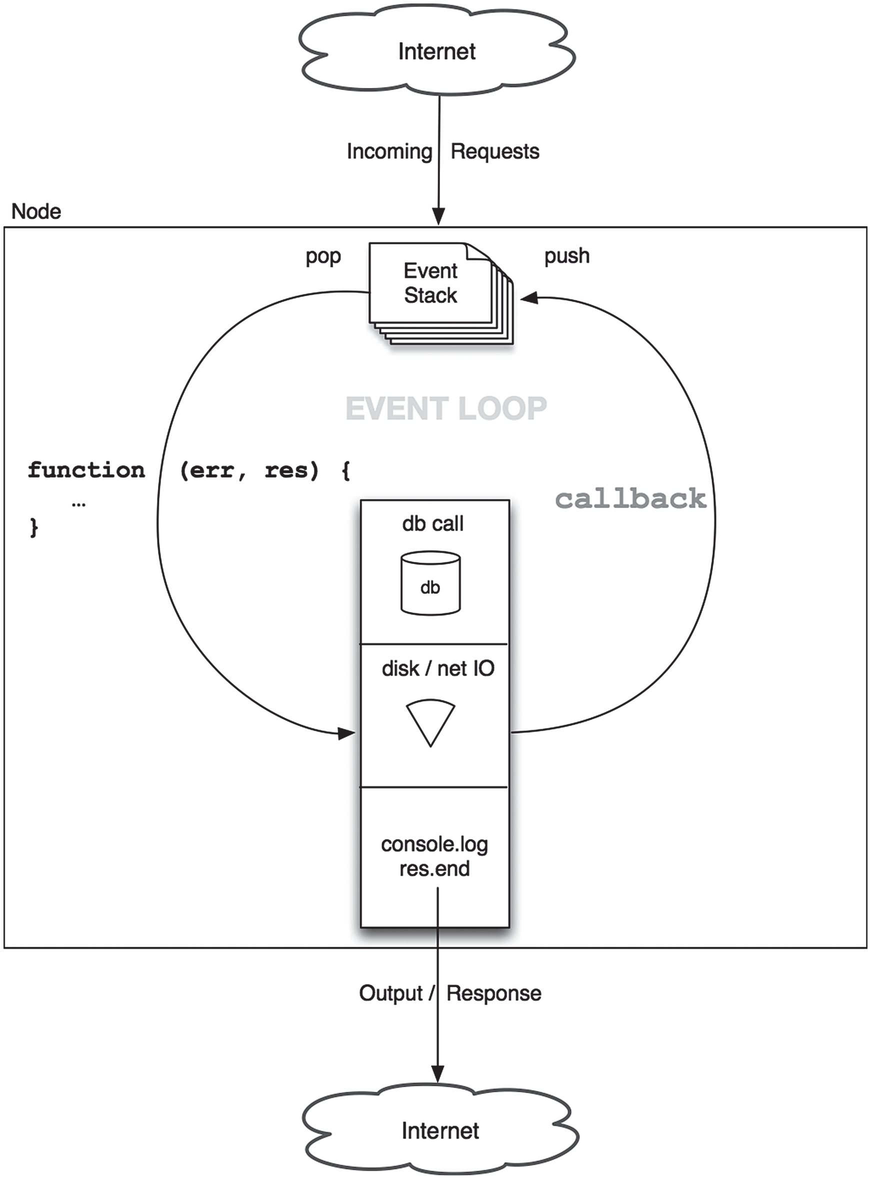

Figure 3.2 As long as there is code executing or somebody is waiting for something, Node runs

Error Handling and Asynchronous Functions

In the preceding chapter, I discussed error handling and events as well as the `try / catch` block in JavaScript. The addition of nonblocking IO and asynchronous function callbacks in this chapter, however, creates a new problem. Consider the following code:

```
try {
    setTimeout(() => {
        throw new Error("Uh oh!");
    }, 2000);
} catch (e) {
    console.log("I caught the error: " + e.message);
}
```

If you run this code, you might very well expect to see the output `"I caught the error: Uh oh!"`. But you do not. You actually see the following:

```
timers.js:103
            if (!process.listeners('uncaughtException').length) throw e;
                                                                      ^

Error: Uh oh, something bad!
    at Object._onTimeout errors_async.js:5:15)
    at Timer.list.ontimeout (timers.js:101:19)
```

What happened? Did I not say that `try / catch` blocks were supposed to catch errors for you? I did, but asynchronous callbacks throw a new little wrench into this situation.

In reality, the call to `setTimeout` *does* execute within the `try / catch` block. If that function were to throw an error, the `catch` block would catch it, and you would see the message that you had hoped to see. However, the `setTimeout` function just adds an event to the Node event queue (instructing it to call the provided function after the specified time interval—2000 ms in this example) and then returns. The provided callback function actually operates within its own entirely new context and scope!

As a result, when you call asynchronous functions for nonblocking IO, very few of them throw errors, but instead use a separate way of telling you that something has gone wrong.

In Node, you use a number of core *patterns* to help you standardize how you write code and avoid errors. These patterns are not enforced syntactically by the language or runtime, but you will see them used frequently and should absolutely use them yourself.

The callback Function and Error Handling

One of the first patterns you will see is the format of the *callback* function you pass to most asynchronous functions. It always has at least one parameter, the success or failure status of the last operation, and very commonly a second parameter with some sort of additional results or information from the last operation (such as a file handle, database connection, rows from a query, and so on); some callbacks are given even more than two:

```
do_something(param1, param2, ..., paramN, function (err, results) { ... });
```

The err parameter is either

- null, indicating the operation was a success, and (if there should be one) there will be a result.
- An instance of the Error object class. You will occasionally notice some inconsistency here, with some people always adding a code field to the Error object and then using the message field to hold a description of what happened, whereas others have chosen other patterns. For all the code you write in this book, you will follow the pattern of always including a code field and using the message field to provide as much information as you can. For all the modules you write, you will use a string value for the code because strings tend to be a bit easier to read. Some libraries provide extra data in the Error object with additional information, but at least the two members should always be there.

This standard prototype methodology enables you to always write predictable code when you are working with nonblocking functions. Throughout this book, I demonstrate two common coding styles for handling errors in callbacks. Here's the first:

```
fs.open('info.txt', 'r', (err, handle) => {
    if (err) {
        console.log("ERROR: " + err.code + " (" + err.message ")");
        return;
    }
    // success!! continue working here
});
```

In this style, you check for errors and return if you see one; otherwise, you continue to process the result. And now here's the other way:

```
fs.open('info.txt', 'r', (err, handle) => {
    if (err) {
        console.log("ERROR: " + err.code + " (" + err.message ")");
    } else {
        // success! continue working here
    }
});
```

In this method, you use an if ... then ... else statement to handle the error.

The difference between these two may seem like splitting hairs, but the former method is a little more prone to bugs and errors for those cases when you forget to use the return statement inside the if statement, whereas the latter results in code that indents itself much more quickly and you end up with lines of code that are quite long and less readable. We'll look at a solution to this second problem in the section titled "Managing Asynchronous Code" in Chapter 5.

A fully updated version of the file loading code with error handling is shown in Listing 3.1.

Listing 3.1 **File Loading with Full Error Handling**

```
var fs = require('fs');

fs.open('info.txt', 'r', (err, handle) => {
    if (err) {
        console.log("ERROR: " + err.code + " (" + err.message + ")");
        return;
    }
    var buf = new Buffer(100000);
    fs.read(handle, buf, 0, 100000, null, (err, length) => {
        if (err) {
            console.log("ERROR: " + err.code
                        + " (" + err.message + ")");
            return;
        }
        console.log(buf.toString('utf8', 0, length));
        fs.close(handle, () => { /* don't care */ });
    });
});
```

Who Am I? Maintaining a Sense of Identity

Now you're ready to write a little class to help you with some common file operations:

```
var fs = require('fs');

function FileObject () {
    this.filename = '';

    this.file_exists = function (callback) {
        console.log("About to open: " + this.filename);
        fs.open(this.filename, 'r', function (err, handle) {
            if (err) {
                console.log("Can't open: " + this.filename);
                callback(err);
                return;
            }
            fs.close(handle, function () { });
            callback(null, true);
        });
    };
}
```

You have currently added one property, `filename`, and a single method, `file_exists`. This method does the following:

- It tries to open the file specified in the `filename` property read-only.
- If the file doesn't exist, it prints a message and calls the callback function with the error info.
- If the file does exist, it calls the callback function indicating success.

Now, run this class with the following code:

```
var fo = new FileObject();
fo.filename = "file_that_does_not_exist";

fo.file_exists((err, results) => {
    if (err) {
        console.log("\nError opening file: " + JSON.stringify(err));
        return;
    }

    console.log("file exists!!!");
});
```

You might expect the following output:

```
About to open: file_that_does_not_exist
Can't open: file_that_does_not_exist
```

But, in fact, you see this:

```
About to open: file_that_does_not_exist
Can't open: undefined
```

What happened? Most of the time, when you have a function nested within another, it inherits the scope of its parent/host function and should have access to all the same variables. So why does the nested callback function not get the correct value for the `filename` property?

The problem lies with the `this` keyword and asynchronous callback functions. Don't forget that when you call a function like `fs.open`, it initializes itself, calls the underlying operating system function (in this case to open a file), and places the provided callback function on the event queue. Execution immediately returns to the `FileObject#file_exists` function, and then you exit. When the `fs.open` function completes its work and Node runs the callback, you no longer have the context of the `FileObject` class any more, and the callback function is given a new `this` pointer representing some other execution context!

The bad news is that you have, indeed, lost your `this` pointer referring to the `FileObject` class. The good news is that the callback function for `fs.open` does still have its function scope. A common solution to this problem is to "save" the disappearing `this` pointer in a variable called `self` or `me` or something similar. Now rewrite the `file_exists` function to take advantage of this:

```
    this.file_exists = function (callback) {
        var self = this;
```

```
        console.log("About to open: " + self.filename);
        fs.open(this.filename, 'r', function (err, handle) {
            if (err) {
                console.log("Can't open: " + self.filename);
                callback(err);
                return;
            }

            fs.close(handle, function () { });
            callback(null, true);
        });
    };
```

Because local function scope *is* preserved via closures, the new `self` variable is maintained for you even when your callback is executed asynchronously later by Node.js. You will make extensive use of this in all your applications. Some people like to use `me` instead of `self` because it is shorter; others still use completely different words. Pick whatever kind you like and stick with it for consistency.

The above scenario is another reason to use *arrow functions*, introduced in the previous chapter. Arrow functions capture the `this` value of the enclosing scope, so your code actually works as expected! Thus, as long as you are using `=>`, you can continue to use the `this` keyword, as follows:

```
var fs = require('fs');

function FileObject () {
    this.filename = '';

    // Always use "function" for member fns, not =>, see below for why
    this.file_exists = function (callback) {
        console.log("About to open: " + this.filename);
        fs.open(this.filename, 'r', (err, handle) => {
            if (err) {
                console.log("Can't open: " + this.filename);
                callback(err);
                return;
            }
            fs.close(handle, () => { });
            callback(null, true);
        });
    };
}
```

One other thing to note is that we *do not* use arrow functions for declaring member functions on objects or prototypes. This is because in those cases, we actually *do* want the `this` variable to update with the context of the currently executing object. Thus, you'll see us using => only when we're using anonymous functions in other contexts.

The key takeaway for this section should be: If you're using an anonymous function that's not a class or prototype method, you should stop and think before using `this`. There's a good chance it won't work the way you want. Use arrow functions as much as possible.

Being Polite—Learning to Give Up Control

Node runs in a single thread with a single event loop that makes calls to external functions and services. It places callback functions on the event queue to wait for the responses and otherwise tries to execute code as quickly as possible. So what happens if you have a function that tries to compute the intersection between two arrays:

```
function compute_intersection(arr1, arr2, callback) {
    var results = [];
    for (var i = 0 ; i < arr1.length; i++) {
        for (var j = 0; j < arr2.length; j++) {
            if (arr2[j] == arr1[i]) {
                results[results.length] = arr2[j];
                break;
            }
        }
    }
    callback(null, results);   // no error, pass in results!
}
```

For arrays of a few thousand elements, this function starts to consume significant amounts of time to do its work, on the order of a second or more. In a single-threaded model, where Node.js can do only one thing at a time, this amount of time can be a problem. Similar functions that compute hashes, digests, or otherwise perform expensive operations are going to cause your applications to temporarily "freeze" while they do their work? What can you do?

In the introduction to this book, I mentioned that there are certain things for which Node.js is not particularly well suited, and one of them is definitely acting as a *compute server*. Node is far better suited to more common network application tasks, such as those with heavy amounts of IO and requests to other services. If you want to write a server that does a lot of expensive computations and calculations, you might want to consider moving these operations to other services that your Node applications can then call remotely.

I am not saying, however, that you should completely shy away from computationally intensive tasks. If you're doing these only some of the time, you can still include them in Node.js and take advantage of a method on the `process` global object called `nextTick`. This method basically says "Give up control of execution, and then when you have a free moment, call the provided function." It tends to be significantly faster than just using the `setTimeout` function.

Listing 3.2 contains an updated version of the `compute_intersection` function that yields every once in a while to let Node process other tasks.

Listing 3.2 **Using Process#nextTick to be Polite**

```
function compute_intersection(arr1, arr2, callback) {
    // let's break up the bigger of the two arrays
    var bigger = arr1.length > arr2.length ? arr1 : arr2;
    var smaller = bigger == arr1 ? arr2 : arr1;
    var biglen = bigger.length;
    var smlen = smaller.length;

    var sidx = 0;           // starting index of any chunk
    var size = 10;          // chunk size, can adjust!
    var results = [];       // intermediate results

    // for each chunk of "size" elements in bigger, search through smaller
    function sub_compute_intersection() {
        for (var i = sidx; i < (sidx + size) && i < biglen; i++) {
            for (var j = 0; j < smlen; j++) {
                if (bigger[i] == smaller[j]) {
                    results.push(smaller[j]);
                    break;
                }
            }
        }

        if (i >= biglen) {
            callback(null, results);   // no error, send back results
        } else {
            sidx += size;
            process.nextTick(sub_compute_intersection);
        }
    }

    sub_compute_intersection();
}
```

In this new version of the function, you basically divide the bigger of the input arrays into chunks of 10 (you can choose whatever number you want), compute the intersection of that many items, and then call `process#nextTick` to allow other events or requests a chance to do their work. Only when there are no events in front of you any longer, will you continue to do the work. Don't forget that passing the callback function `sub_compute_intersection` to `process#nextTick` ensures that the current scope is preserved as a closure, so you can store the intermediate results in the variables in `compute_intersection`.

Listing 3.3 shows the code you use to test this new `compute_intersection` function.

Listing 3.3 **Testing the compute_intersection Function**

```
var a1 = [ 3476, 2457, 7547, 34523, 3, 6, 7,2, 77, 8, 2345,
           7623457, 2347, 23572457, 237457, 234869, 237,
           24572457524] ;
var a2 = [ 3476, 75347547, 2457634563, 56763472, 34574, 2347,
           7, 34652364 , 13461346, 572346, 23723457234, 237,
           234, 24352345, 537, 2345235, 2345675, 34534,
           7582768, 284835, 8553577, 2577257,545634, 457247247,
           2345 ];

compute_intersection(a1, a2, function (err, results) {
    if (err) {
        console.log(err);
    } else {
        console.log(results);
    }
});
```

Although this has made things a bit more complicated than the original version of the function to compute the intersections, the new version plays much better in the single-threaded world of Node event processing and callbacks, and you can use `process.nextTick` in any situation in which you are worried that a complex or slow computation is necessary.

Synchronous Function Calls

Now that I have spent nearly an entire chapter telling you how Node.js is very much asynchronous and about all the tricks and traps of programming nonblocking IO, I must mention that Node actually *does* have synchronous versions of some key APIs, most notably file APIs. You use them for writing command-line tools in Chapter 12, “Command-Line Programming.”

To demonstrate briefly here, you can rewrite the first script of this chapter as follows:

```
var fs = require('fs');

var handle = fs.openSync('info.txt', 'r');
var buf = new Buffer(100000);
var read = fs.readSync(handle, buf, 0, 10000, null);
console.log(buf.toString('utf8', 0, read));
fs.closeSync(handle);
```

As you work your way through this book, I hope you are able to see quite quickly that Node.js isn’t just for network or web applications. You can use it for everything from command-line utilities to prototyping to server management and more!

Summary

Switching from a model of programming where you execute a sequence of synchronous or blocking IO function calls and wait for each of them to complete before moving on to the next call, to a model where you do everything asynchronously and wait for Node to tell you when a given task is done requires a bit of mental gymnastics and experimentation. But I am convinced that when you get the hang of this, you'll never be able imagine going back to the other way of writing your web apps.

Next, you write your first simple JSON application server.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

N

O

P

Q

R

T

U

V

W

Y