Alex Young
Marc Harter

Foreword by
Ben Noordhuis

Node.js in Practice

Includes 115 Techniques

MANNING

Node.js in Practice

ALEX YOUNG
MARC HARTER

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Cynthia Kane
Technical development editor: Jose Maria Alvarez Rodriguez
Copyeditor: Benjamin Berg
Proofreader: Katie Tennant
Typesetter: Gordan Salinovic
Cover designer: Marija Tudor

ISBN 9781617290930
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – EBM – 19 18 17 16 15 14

brief contents

contents

foreword

You have in your hands a book that will take you on an in-depth tour of Node.js. In the pages to come, Alex Young and Marc Harter will help you grasp Node's core in a deep way: from modules to real, networked applications.

Networked applications are, of course, an area where Node.js shines. You, dear reader, are likely well aware of that; I daresay it is your main reason for purchasing this tome! For the few of you who actually read the foreword, let me tell you the story of how it all began.

In the beginning, there was the C10K problem. And the C10K problem raised this question: if you want to handle 10,000 concurrent network connections on contemporary hardware, how do you go about that?

You see, for the longest time operating systems were terrible at dealing with large numbers of network connections. The hardware was terrible in many ways, the software was terrible in other ways, and when it came to the interaction between hardware and software … linguists had a field day coming up with proper neologisms; plain *terrible* doesn't do it justice. Fortunately, technology is a story of progress; hardware gets better, software saner. Operating systems improved at managing large numbers of network connections, as did user software.

We conquered the C10K problem a long time ago, moved the goal posts, and now we've set our sights on the C100K, C500K, and C1M problems. Once we've comfortably crossed those frontiers, I fully expect that the C10M problem will be next.

Node.js is part of this story of ever-increasing concurrency, and its future is bright: we live in an increasingly connected world and that world needs a power tool to connect everything. I believe Node.js is that power tool, and I hope that, after reading this book, you will feel the same way.

Ben Noordhuis
Cofounder, StrongLoop, Inc.

preface

When Node.js arrived in 2009, we knew something was different. JavaScript on the server wasn't anything new. In fact, server-side JavaScript has existed almost as long as client-side JavaScript. With Node, the speed of the JavaScript runtimes, coupled with the event-based parallelism that many JavaScript programmers were already familiar with, were indeed compelling. And not just for client-side JavaScript developers, which was our background—Node attracted developers from the systems level to various server-side backgrounds, PHP to Ruby to Java. We all found ourselves inside this movement.

At that time, Node was changing a lot, but we stuck with it and learned a whole lot in the process. From the start, Node focused on making a small, low-level core library that would provide enough functionality for a large, diverse user space to grow. Thankfully, this large and diverse user space exists today because of these design decisions early on. Node is a lot more stable now and used in production for numerous startups as well as established enterprises.

When Manning approached us about writing an intermediate-level book on Node, we looked at the lessons we had learned as well as common pitfalls and struggles we saw in the Node community. Although we loved the huge number of truly excellent third-party modules available to developers, we noticed many developers were getting less and less education on the core foundations of Node. So we set out to write *Node in Practice* to journey into the roots and foundations of Node in a deep and thorough manner, as well as tackle many issues we personally have faced and have seen others wrestle with.

acknowledgments

We have many people to thank, without whose help and support this book would not have been possible.

Thanks to the Manning Early Access Program (MEAP) readers who posted comments and corrections in the Author Online forum.

Thanks to the technical reviewers who provided invaluable feedback on the manuscript at various stages of its development: Alex Garrett, Brian Falk, Chris Joakim, Christoph Walcher, Daniel Bretoi, Dominic Pettifer, Dylan Scott, Fernando Monteiro Kobayashi, Gavin Whyte, Gregor Zurowski, Haytham Samad, JT Marshall, Kevin Baister, Luis Gutierrez, Michael Piscatello, Philippe Charrière, Rock Lee, Shiju Varghese, and Todd Williams.

Thanks to the entire Manning team for helping us every step of the way, especially our development editor Cynthia Kane, our copyeditor Benjamin Berg, our proofreader Katie Tennant, and everyone else who worked behind the scenes.

Special thanks to Ben Noordhuis for writing the foreword to our book, and to Valentin Crettaz and Michael Levin for their careful technical proofread of the book shortly before it went into production.

Alex Young

I couldn't have written this book without the encouragement and support of the DailyJS community. Thanks to everyone who has shared modules and libraries with me over the last few years: keeping up to date with the Node.js community would have been impossible without you. Thank you also to my colleagues at Papers who have allowed me to

use my Node.js skills in production. Finally, thanks to Yuka for making me believe I can do crazy things like start companies and write books.

Marc Harter

I would like thank Ben Noordhuis, Isaac Schlueter, and Timothy Fontaine for all the IRC talks over Node; you know the underlying systems that support Node in such a deep way that learning from you makes Node even richer. Also, I want to thank my coauthor Alex; it seems rare to have such a similar approach to writing a book as I did with Alex, plus it was fun for a Midwestern US guy to talk shop with an English chap. Ultimately my heart goes out to my wife, who really made this whole thing possible, if I'm honest. Hannah, you are loved; thank you.

about this book

Node.js in Practice exists to provide readers a deeper understanding of Node's core modules and packaging system. We believe this is foundational to being a productive and confident Node developer. Unfortunately, this small core is easily missed for the huge and vibrant third-party ecosystem with modules prebuilt for almost any task. In this book we go beyond regurgitating the official Node documentation in order to get practical and thorough. We want the reader to be able to dissect the inner workings of the third-party modules they include as well as the projects they write.

This book is not an entry-level Node book. For that, we recommend reading Manning's *Node.js In Action*. This book is targeted at readers who already have experience working with Node and are looking to take it up a notch. Intermediate knowledge of JavaScript is recommended. Familiarity with the Windows, OS X, or Linux command line is also recommended.

In addition, we're aware that many Node developers have come from a client-side JavaScript background. For that reason, we spend some time explaining less-familiar concepts such as working with binary data, how underlying networking and file systems work, and interacting with the host operating system—all using Node as a teaching guide.

Chapter roadmap

This book is organized into three parts.

Part 1 covers Node's core fundamentals, where we focus our attention on what's possible using only Node's core modules (no third-party modules). Chapter 1 recaps

Node.js's purpose and function. Then chapters 2 through 8 each cover in depth a different core aspect of Node from buffers to streams, networking to child processes.

Part 2 focuses on real-world development recipes. Chapters 9 through 12 will help you master four highly applicable skills—testing, web development, debugging, and running Node in production. In addition to Node core modules, these sections include the use of various third-party modules.

Part 3 guides you through creating your own Node modules in a straightforward manner that ties in all kinds of ways to use npm commands for packaging, running, testing, benchmarking, and sharing modules. It also includes helpful tips on versioning projects effectively.

There are 115 techniques in the book, each module covering a specific Node.js topic or task, and each divided into practical Problem/Solution/Discussion sections.

Code conventions and downloads

All source code in the book is in a `fixed-width font like this`, which sets it off from the surrounding text. In many listings, the code is annotated to point out the key concepts, and numbered bullets are sometimes used in the text to provide additional information about the code.

This book's coding style is based on the Google JavaScript Style Guide.[1] That means we've put `var` statements on their own lines, used `camelCase` to format function and variable names, and we always use semicolons. Our style is a composite of the various JavaScript styles used in the Node community.

Most of the code shown in the book can be found in various forms in the sample source code that accompanies it. The sample code can be downloaded free of charge from the Manning website at www.manning.com/Node.jsinPractice, as well as from GitHub at the following link: https://github.com/alexyoung/nodeinpractice.

Author Online forum

Purchase of *Node.js in Practice* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/Node.jsinPractice. This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

You can also contact the authors at the following Google Group URL: https://groups.google.com/forum/#!forum/nodejsinpractice.

[1] https://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml

about the cover illustration

The caption for the illustration on the cover of *Node.js in Practice* is "Young Man from Ayvalik," a town in Turkey on the Aegean Coast. The illustration is taken from a collection of costumes of the Ottoman Empire published on January 1, 1802, by William Miller of Old Bond Street, London. The title page is missing from the collection and we have been unable to track it down to date. The book's table of contents identifies the figures in both English and French, and each illustration bears the names of two artists who worked on it, both of whom would no doubt be surprised to find their art gracing the front cover of a computer programming book ... two hundred years later.

The collection was purchased by a Manning editor at an antiquarian flea market in the "Garage" on West 26th Street in Manhattan. The seller was an American based in Ankara, Turkey, and the transaction took place just as he was packing up his stand for the day. The Manning editor didn't have on his person the substantial amount of cash that was required for the purchase, and a credit card and check were both politely turned down. With the seller flying back to Ankara that evening, the situation was getting hopeless. What was the solution? It turned out to be nothing more than an old-fashioned verbal agreement sealed with a handshake. The seller simply proposed that the money be transferred to him by wire, and the editor walked out with the bank information on a piece of paper and the portfolio of images under his arm. Needless to say, we transferred the funds the next day, and we remain grateful and impressed by this unknown person's trust in one of us. It recalls something that might have happened a long time ago.

We at Manning celebrate the inventiveness, the initiative, and, yes, the fun of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by the pictures from this collection.

Part 1

Node fundamentals

Node has an extremely small standard library intended to provide the lowest-level API for module developers to build on. Even though it's relatively easy to find third-party modules, many tasks can be accomplished without them. In the chapters to follow, we'll take a deep dive into a number of core modules and explore how to put them to practical use.

By strengthening your understanding of these modules, you'll in turn become a more well-rounded Node programmer. You'll also be able to dissect third-party modules with more confidence and understanding.

1 Getting started

This chapter covers

- Why Node?
- Node's main features
- Building a Node application

Node has quickly become established as a viable and indeed efficient web development platform. Before Node, not only was JavaScript on the server a novelty, but non-blocking I/O was something that required special libraries for other scripting languages. With Node, this has all changed.

The combination of non-blocking I/O and JavaScript is immensely powerful: we can handle reading and writing files, network sockets, and more, all asynchronously *in the same process*, with the natural and expressive features of JavaScript callbacks.

This book is geared toward intermediate Node developers, so this chapter is a quick refresher. If you want a thorough treatment of Node's basics, then see our companion book, *Node.js in Action* (by Mike Cantelon, Marc Harter, TJ Holowaychuk, and Nathan Rajlich; Manning Publications, 2013).

In this chapter we'll introduce Node, what it is, how it works, and why it's something you can't live without. In chapter 2 you'll get to try out some techniques by looking at Node's globals—the objects and methods available to every Node process.

Preflight check

Node In Practice is a recipe-style book, aimed at intermediate and advanced Node developers. Although this chapter covers some introductory material, later chapters advance quickly. For a beginner's introduction to Node, see our companion book, *Node.js in Action*.

1.1 Getting to know Node

Node is a *platform* for developing network applications. It's built on V8, Google's JavaScript runtime engine. Node isn't just V8, though. An important part of the Node platform is its core library. This encompasses everything from TCP servers to asynchronous and synchronous file management. This book will teach you how to use these modules properly.

But first: why use Node, and when should you use it? Let's look into that question by seeing what kinds of scenarios Node excels at.

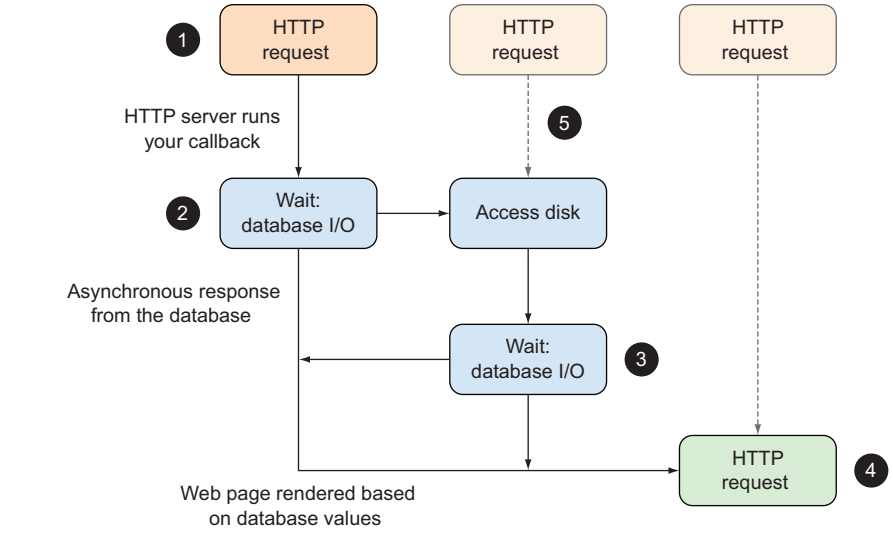
1.1.1 Why Node?

Let's say you're building an advertising server and distributing millions of adverts per minute. Node's non-blocking I/O would be an extremely cost-effective solution for this, because the server could make the best use of available I/O without you needing to write special low-level code. Also, if you already have a web team that can write JavaScript, then they should be able to contribute to the Node project. A typical, heavier web platform wouldn't have these advantages, which is why companies like Microsoft are contributing to Node despite having excellent technology stacks like .NET. Visual Studio users can install Node-specific tools[1] that add support for IntelliSense, profiling, and even npm. Microsoft also developed WebMatrix (http://www.microsoft.com/web/webmatrix/), which directly supports Node and can also be used to deploy Node projects.

Node embraces non-blocking I/O as a way to improve performance in certain types of applications. JavaScript's traditional event-based implementation means it has a relatively convenient and well-understood syntax that suits asynchronous programming. In a typical programming language, an I/O operation blocks execution until it completes. Node's asynchronous file and network APIs mean processing can still occur while these relatively slow I/O operations finish. Figure 1.1 illustrates how different tasks can be performed using asynchronous network and file system APIs.

In figure 1.1, a new HTTP request has been received and parsed by Node's `http` module ❶. The ad server's application code then makes a database query, using an asynchronous API—a callback passed to a database read function ❷. While Node waits for this to finish, the ad server is able to read a template file from the disk ❸.

[1] See https://nodejstools.codeplex.com/.



1. An HTTP request is received from a browser.
2. After Node parses the request, your code executes a database query.
3. While the query callback waits to run, some of your other code reads from an HTML template file.
4. The web page is then rendered based on the template and database values.
5. Meanwhile, other requests can be handled as well.

Figure 1.1 An advertising server built with Node

This template will be used to display a suitable web page. Once the database request has finished, the template and database results are used to render the response 4.

While this is happening, other requests could also be hitting the ad server, and they'll be handled based on the available resources 5. Without having to think about threads when developing the ad server, you're able to push Node to use the server's I/O resources very efficiently, just by using standard JavaScript programming techniques.

Other scenarios where Node excels are web APIs and web scraping. If you're downloading and extracting content from web pages, then Node is perfect because it can be coaxed into simulating the DOM and running client-side JavaScript. Again, Node has a performance benefit here, because scrapers and web spiders are costly in terms of network and file I/O.

If you're producing or consuming JSON APIs, Node is an excellent choice because it makes working with JavaScript objects easy. Node's web frameworks (like Express, http://expressjs.com) make creating JSON APIs fast and friendly. We have full details on this in chapter 9.

Node isn't limited to web development. You can create any kind of TCP/IP server that you like. For example, a network game server that broadcasts the game's state to

When to use Node

To get you thinking like a true Nodeist, the table below has examples of applications where Node is a good fit.

Node's strengths

Scenario	Node's strengths
Advertising distribution	■ Efficiently distributes small pieces of information ■ Handles potentially slow network connections ■ Easily scales up to multiple processors or servers
Game server	■ Uses the accessible language of JavaScript to model business logic ■ Programs a server catering to specific networking requirements without using C
Content management system, blog	■ Good for a team with client-side JavaScript experience ■ Easy to make RESTful JSON APIs ■ Lightweight server, complex browser JavaScript

various players over TCP/IP sockets can perform background tasks, perhaps maintaining the game world, while it sends data to the players. Chapter 7 explores Node's networking APIs.

1.1.2 Node's main features

Node's main features are its standard library, module system, and npm. Of course, there's more to it than that, but in this book we'll focus on teaching you how to use these parts of Node. We'll use third-party libraries where it's considered best practice, but you'll see a lot of Node's built-in features.

In fact, Node's strongest and most powerful feature is its standard library. This is really two parts: a set of binary libraries and the core modules. The binary libraries include `libuv`, which provides a fast run loop and non-blocking I/O for networking and the file system. It also has an HTTP library, so you can be sure your HTTP clients and servers are fast.

Figure 1.2 is a high-level overview of Node's internals that shows how everything fits into place.

Node's core modules are mostly written in JavaScript. That means if there's anything you either don't understand or want to understand in more detail, then you can read Node's source code. This includes features like networking, high-level file system operations, the module system, and streams. It also includes Node-specific features like running multiple Node processes at once with the cluster module, and wrapping sections of code in event-based error handlers, known as *domains*.

The next few sections focus on each core module in more detail, starting with the `events` API.

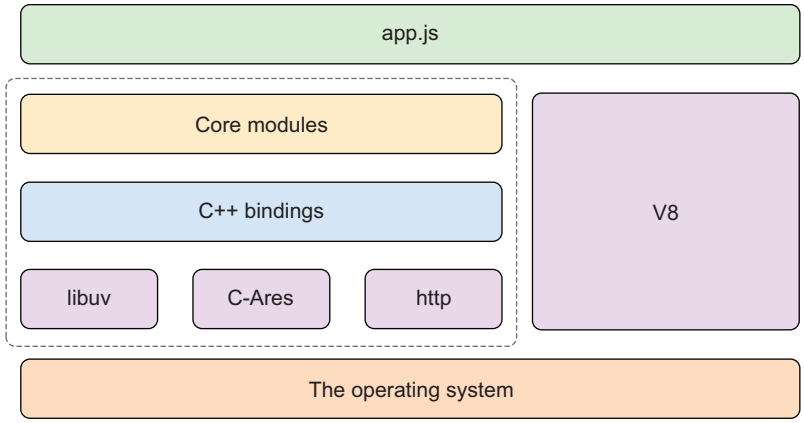


Figure 1.2 **Node's key parts in context**

EVENTEMITTER: AN API FOR EVENTS

Sooner or later every Node developer runs into `EventEmitter`. At first it seems like something only library authors would need to use, but it's actually the basis for most of Node's core modules. The streams, networking, and file system APIs derive from it.

You can inherit from `EventEmitter` to make your own event-based APIs. Let's say you're working on a PayPal payment-processing module. You could make it event-based, so instances of `Payment` objects emit events like `paid` and `refund`. By designing the class this way, you decouple it from your application logic, so you can reuse it in more than one project.

We have a whole chapter dedicated to events: see chapter 4 for more. Another interesting part of `EventEmitter` is that it's used as the basis for the `stream` module.

STREAM: THE BASIS FOR SCALABLE I/O

Streams inherit from `EventEmitter` and can be used to model data with unpredictable throughput—like a network connection where data speeds can vary depending on what other users on the network are doing. Using Node's `stream` API allows you to create an object that receives events about the connection: `data` for when new data comes in, `end` when there's no more data, and `error` when errors occur.

Rather than passing lots of callbacks to a readable stream constructor function, which would be messy, you subscribe to the events you're interested in. Streams can be piped together, so you could have one stream class that reads data from the network and then pipe it to a stream that transforms the data into something else. This could be data from an XML API that's transformed into JSON, making it easier to work with in JavaScript.

We love streams, so we've dedicated a whole chapter to them. Skip to chapter 5 to dive right in. You might think that events and streams sound abstract, and though that's true, it's also interesting to note that they're used as a basis for I/O modules, like `fs` and `net`.

FS: WORKING WITH FILES

Node's file system module is capable of reading and writing files using non-blocking I/O, but it also has synchronous methods. You can get information about files with `fs.stat`, and the synchronous equivalent is `fs.statSync`.

If you want to use streams to process the contents of a file in a super-efficient manner, then use `fs.createReadStream` to return a `ReadableStream` object. There's more about this in chapter 6.

NET: CREATE NETWORK CLIENTS AND SERVERS

The networking module is the basis for the `http` module and can be used to create generalized network clients and servers. Although Node development is typically thought of as web-based, chapter 7 shows you how to create TCP and UDP servers, which means you're not limited to HTTP.

GLOBAL OBJECTS AND OTHER MODULES

If you have some experience making web applications with Node, perhaps with the Express framework, then you've already been using the `http`, `net`, and `fs` core modules without necessarily realizing it. Other built-in features aren't headline-grabbing, but are critical to creating programs with Node.

One example is the idea of global objects and methods. The `process` object, for example, allows you to pipe data into and out of a Node program by accessing the standard I/O streams. Much like Unix and Windows scripting, you can `cat` data to a Node program. The ubiquitous `console` object, beloved by JavaScript developers everywhere, is also considered a global object.

Node's module system is also part of this global functionality. Chapter 2 is packed with techniques that show you how to use these features.

Now that you've seen some of the core modules, it's time to see them in action. The example will use the `stream` module to generate statistics on streams of text, and you'll be able to use it with files and HTTP connections. If you want to learn more about the basics behind streams or HTTP in Node, refer to *Node.js in Action*.

1.2 Building a Node application

Instead of wading through more theory, we'll show you how to build a Node application. It's not just any application, though: it uses some of Node's key features, like modules and streams. This will be a fast and intense tour of Node, so start up your favorite text editor and terminal and get ready.

Here's what you'll learn over the next 10 minutes:

- How to create a new Node project
- How to write your own stream class
- How to write a simple test and run it

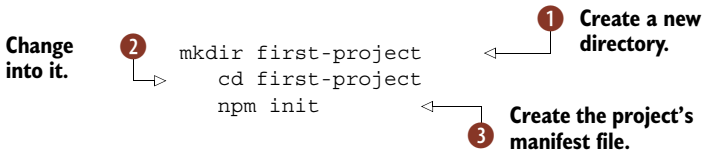
Streams are great for processing data, whether you're reading, writing, or transforming it. Imagine you want to convert data from a database into another format, like CSV. You could create a stream class that accepts input from a database and outputs it as a

stream of CSV. The output of this new CSV stream could be connected to an HTTP request, so you could stream CSV directly to a browser. The same class could even be connected to a writable file stream—you could even fork the stream to create a file *and* send it to a web browser.

In this example, the stream class will accept text input, count word matches based on a regular expression, and then emit the results in an event when the stream has finished being sent. You could use this to count word matches in a text file, or pipe data from a web page and count the number of paragraph tags—it's up to you. First we need to create a new project.

1.2.1 Creating a new Node project

You might be wondering how a professional Node developer creates a new project. This is a straightforward process, thanks to npm. Though you could create a JavaScript file and run `node file.js`, we'll use `npm init` to make a new project with a package.json file. Create a new directory (1), `cd` (2) into it, and then run `npm init` (3):



Get used to typing these commands: you'll be doing it often! You can press Return to accept the defaults when prompted by npm. Before you've written a line of JavaScript, you've already seen how cool one of Node's major features—npm—is. It's not just for installing modules, but also for managing projects.

> **When to use a package.json file**
>
> You may have an idea for a small script, and may be wondering if a package.json file is really necessary. It isn't always necessary, but in general you should create them as often as possible.
>
> Node developers prefer small modules, and expressing dependencies in package.json means your project, no matter how small, is super-easy to install in the future, or on another person's machine.

Now it's time to write some JavaScript. In the next section you'll create a new JavaScript file that implements a stream.

1.2.2 Making a stream class

Create a new file called countstream.js and use `util.inherits` to derive from `stream.Writable` and implement the required `_write` method. Too fast? Let's slow down. The full source is in the following listing.

Listing 1.1 A writable stream that counts

```
var Writable = require('stream').Writable;
var util = require('util');

module.exports = CountStream;

util.inherits(CountStream, Writable);

function CountStream(matchText, options) {
  Writable.call(this, options);
  this.count = 0;
  this.matcher = new RegExp(matchText, 'ig');
}

CountStream.prototype._write = function(chunk, encoding, cb) {
  var matches = chunk.toString().match(this.matcher);
  if (matches) {
    this.count += matches.length;
  }
  cb();
};

CountStream.prototype.end = function() {
  this.emit('total', this.count);
};
```

1 Inherit from the Writable stream.

2 Create a RegExp object that matches globally and ignores case.

3 Convert the current chunk of input into a string and use it to count matches.

4 When the stream has ended, "publish" the total number of matches.

This example illustrates how subsequent examples in this book work. We present a snippet of code, annotated with hints on the underlying code. For example, the first part of the class uses the `util.inherits` method to inherit from the `Writable` base class 1. This example won't be fully fleshed-out here—for more on writing your own streams, see technique 30 in chapter 5. For now, just focus on how regular expressions are passed to the constructor 2 and used to count text as it flows into instances of the class 3. Node's `Writable` class calls `_write` for us, so we don't need to worry about that yet.

> **STREAMS AND EVENTS** In listing 1.1 there was an event, `total`. This is one we made up—you can make up your own as well. Streams inherit from `EventEmitter`, so they have the same `emit` and `on` methods.

Node's `Writable` base class will also call `end` when there's no more data 4. This stream can be instantiated and piped as required. In the next section you'll see how to connect it using `pipe`.

1.2.3 Using a stream

Now that you've seen how to make a stream class, you're probably dying to try it out. Make another file, index.js, and add the code shown in the next listing.

Listing 1.2 Using the `CountStream` class

```
var CountStream = require('./countstream');
var countStream = new CountStream('book');
var http = require('http');

http.get('http://www.manning.com', function(res) {
  res.pipe(countStream);
});

countStream.on('total', function(count) {
  console.log('Total matches:', count);
});
```

1 **Load the countstream.js file.**

Instantiate a CountStream class that counts text matching "book." 2

Download www.manning.com. 3

4 **Pipe the data from the website to countStream, thereby counting the text**

You can run this example by typing `node index.js`. It should display something like `Total matches: 24`. You can experiment with it by changing the URL that it fetches.

This example loads the module from listing 1.1 1 and then instantiates it with the text `'book'` 2. It also downloads the text from a website using Node's standard `http` module 3 and then pipes the result through our `CountStream` class 4.

The significant thing here is `res.pipe(countStream)`. When you pipe data, it doesn't matter how big it is or if the network is slow: the `CountStream` class will dutifully count matches until the data has been processed. This Node program *does not* download the entire file first! It takes the file—piece by piece—and processes it. That's the big thing here, and a critical aspect to Node development.

To recap, figure 1.3 summarizes what you've done so far to create a new Node project. First you created a new directory, and ran `npm init` 1, then you created some JavaScript files 2, and finally you ran the code 3.

Another important part of Node development is testing. The next section wraps up this example by testing `CountStream`.

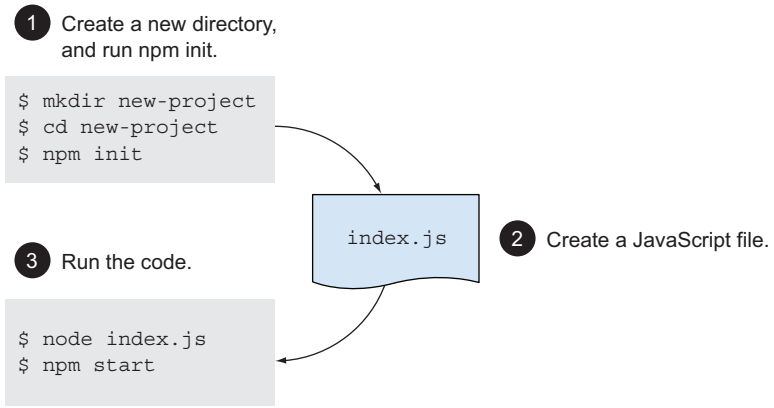


Figure 1.3 The three steps to creating a new Node project

1.2.4 Writing a test

We can write a short test for `CountStream` without using any third-party modules. Node comes with a built-in `assert` module, so we can use that for a quick test. Open test.js and add the code shown next.

Listing 1.3 Using the `CountStream` class

```
var assert = require('assert');
var CountStream = require('./countstream');
var countStream = new CountStream('example');
var fs = require('fs');
var passed = 0;

countStream.on('total', function(count) {
  assert.equal(count, 1);
  passed++;
});

fs.createReadStream(__filename).pipe(countStream);

process.on('exit', function() {
  console.log('Assertions passed:', passed);
});
```

1. The 'total' event will be emitted when the stream is finished.
2. Assert the count is the expected amount.
3. Create a readable stream of the current file, and pipe the data through CountStream.
4. Just before the program is about to exit, display how many assertions have been run.

This test can be run with `node test.js`, and you should see `Assertions passed: 1` printed in the console. The test actually reads the current file and passes the data through `CountStream`. It might invoke Ouroboros, but it's a useful example because it gives us content that we know something about—we can always be sure there is one match for the word *example*.

> **ASSERTIONS** Node comes with an assertion library called `assert`. A basic test can be made by calling the module directly – `assert(expression)`.

The first thing the test does is listen for the `total` event, which is emitted by instances of `CountStream` (1). This is a good place to assert that the number of matches should be the same as what is expected (2). A readable stream that represents the current file is opened and piped through our class (3). Just before the end of the program, we print out how many assertions were hit (4).

This is important because if the `total` event never fires, then `assert.equal` won't run at all. We have no way of knowing whether tests in callbacks are run, so a simple counter has been used to illustrate how Node programming can require patterns from the other programming languages and platforms that you might be familiar with.

If you're getting tired, you can rest here, but there's a bit of sugar to finish off our project. Node developers like to run tests and other scripts using npm on the command line. Open package.json and change the `"test"` property to look like this:

```
"scripts": {
    "test": "node test.js"
},
```

Now you can run tests just by typing `npm test`. This comes in handy when you have lots of tests and running them is more complicated. Running tests, test runners, and asynchronous testing issues are all covered in chapter 10.

> **npm scripts**
>
> The `npm test` and `npm start` commands can be configured by editing package.json. You can also run arbitrary commands, which are invoked with `npm run command`. All you need to do is set a property under `scripts`, just like listing 1.4.
>
> This is useful for specific types of tests or housekeeping routines—for example `npm run integration-tests`, or maybe even `npm run seed-data`.

Depending on your previous experience with Node, this example might have been intense, but it captures how Node developers think and take advantage of the powerful resources that come with Node.

Now that you've seen how a Node project is put together, we're done with the refresher course on Node. The next chapter introduces our first set of techniques, which is the bulk of this book's format. It covers ways of working with the global features that are available to all Node programs.

1.3 Summary

In this chapter you've learned about *Node.js in Practice*—what it covers and how it focuses on Node's impressive built-in core modules like the networking module and file system modules.

You've also learned about what makes Node tick, and how to use it. Some of the main points we covered were

- When to use Node, and how Node builds on non-blocking I/O, allowing you to write standard JavaScript but get great performance benefits.
- Node's standard library is referred to as its *core modules*.
- What the core modules do—I/O tasks like network protocols, and work with files and more generic features like streams.
- How to quickly start a new Node project, complete with a package.json file so dependencies and scripts can be added.
- How to use Node's powerful `stream` API to process data.
- Streams inherit from `EventEmitter`, so you can emit and respond to any events that you want to use in your application.
- How to write small tests just by using npm and the `assert` module—you can test out ideas without installing any third-party libraries.

Finally, we hope you learned something from our introductory application. Using event-based APIs, non-blocking I/O, and streams is really what Node is all about, but it's also important to take advantage of Node's unique tools like the package.json file and npm.

Now it's time for techniques. The next chapter introduces the features that you don't even have to load to use: the global objects.

2 Globals: Node's environment

This chapter covers

- Using modules
- What you can do without requiring a single module
- The process and console objects
- Timers

Global objects are available in all modules. They're universal. Whether you're writing network programs, command-line scripts, or web applications, your program will have access to these objects. That means you can always depend on features like `console.log` and `__dirname`—both are explained in detail in this chapter.

The goal of this chapter is to introduce Node's global objects and methods to help you learn what functionality is available to all Node processes. This will help you better understand Node and its relationship to the operating system, and how it compares with other JavaScript environments like browsers.

Node provides some important functionality out of the box, even without loading any modules. In addition to the features provided by the ECMAScript language, Node has several *host objects*—objects supplied by Node to help programs to execute.

A key global object is `process`, which is used to communicate with the operating system. Unix programmers will be familiar with standard I/O streams, and these are accessible through the `process` object using Node's streaming API.

Another important global is the `Buffer` class. This is included because JavaScript has traditionally lacked support for binary data. As the ECMAScript standards evolve, this is being addressed, but for now most Node developers rely on the `Buffer` class. For more about buffers, see chapter 3.

Some globals are a separate instance for each module. For example, `module` is available in every Node program, but is local to the current module. Since Node programs may consist of several modules, that means a given program has several different `module` objects—they behave like globals, but are in *module scope*.

In the next section you'll learn how to load modules. The objects and methods relating to modules are globals, and as such are always available and ready to be used.

2.1 Modules

Modules can be used to organize larger programs and distribute Node projects, so it's important to be familiar with the basic techniques required to install and create them.

TECHNIQUE 1 Installing and loading modules

Whether you're using a core module provided by Node or a third-party module from npm, support for modules is baked right into Node and is always available.

Problem

You want to load a third-party module from npm.

Solution

Install the module with the command-line tool, npm, and then load the module using `require`. The following listing shows an example of installing the `express` module.

Listing 2.1 Using npm

```
$ npm search express
express                Sinatra inspired web development framework
$ npm install express
express@x.x.x ./node_modules/express
└── methods@x.x.x
└── (Several more dependencies appear here)

$ node
> var express = require('express');
> typeof express
'function'
```

Search for a module based on keywords. (1)

(2) **Load the module using the require method.**

Discussion

The npm command-line tool is distributed with Node, and can be used to search, install, and manage packages. The website https://npmjs.org provides another interface for searching modules, and each module has its own page that displays the associated readme file and dependencies.

Once you know the name of a module, installation is easy: type `npm install module-name` ❶ and it will be installed into ./node_modules. Modules can also be "globally" installed—running `npm install -g module_name` will install it into a global folder. This is usually /usr/local/lib/node_modules on Unix systems. In Windows it should be wherever the `node.exe` binary is located.

After a module has been installed, it can be loaded with `require('module-name')` ❷. The `require` method usually returns an object or a method, depending on how the module has been set up.

Searching npm

By default, npm searches across several fields in each module's package.json file. This includes the module's name, description, maintainers, URL, and keywords. That means a simple search like `npm search express` yields hundreds of results.

You can reduce the number of matches by searching with a regular expression. Wrap a search term in slashes to trigger npm's regular expression matching: `npm search /^express$/`

However, this is still limited. Fortunately, there are open source modules that improve on the built-in search command. For example, `npmsearch` by Gorgi Kosev will order results using its own relevance rankings.

The question of whether to install a module globally is critical to developing maintainable projects. If other people need to work on your project, then you should consider adding modules as dependencies to your project's package.json file. Keeping project dependencies tightly managed will make it easier to maintain them in the future when new versions of dependencies are released.

TECHNIQUE 2 Creating and managing modules

In addition to installing and distributing open source modules, "local" modules can be used to organize projects.

Problem

You want to break a project up into separate files.

Solution

Use the `exports` object.

Discussion

Node's module system provides a solution to splitting code across multiple files. It's very different from `include` in C, or even `require` in Ruby and Python. The main difference is that `require` in Node returns an object rather than loading code into the current namespace, as would occur with a C preprocessor.

In technique 1 you saw how npm can be used to install modules, and how `require` is used to load them. npm isn't the only thing that manages modules, though—Node has a built-in module system based on the CommonJS Modules/1.1 specification (http://wiki.commonjs.org/wiki/Modules/1.1).

This allows objects, functions, and variables to be exported from a file and used elsewhere. The `exports` object is always present and, although this chapter specifically explores global objects, it's not really a global. It's more accurate to say that the `exports` object is in module scope.

When a module is focused around a single class, then users of the module will prefer to type `var MyClass = require('myclass');` rather than `var MyClass = require('myclass').MyClass`, so you should use `modules.export`. Listing 2.2 shows how this works. This is different from using the `exports` object, which requires that you set a property to export something.

Listing 2.2 Exporting modules

```
function MyClass() {
}

MyClass.prototype = {
  method: function() {
    return 'Hello';
  }
};

var myClass = new MyClass();

module.exports = myClass;
```

Objects can be exported, including other objects, methods, and properties.

Listing 2.3 shows how to export multiple objects, methods, or values, a technique that would typically be used for utility libraries that export multiple things.

Listing 2.3 Exporting multiple objects, methods, and values

```
exports.method = function() {
  return 'Hello';
};

exports.method2 = function() {
  return 'Hello again';
};
```

Finally, listing 2.4 shows how to load these modules with `require`, and how to use the functionality they provide.

Listing 2.4 Loading modules with require

```
var myClass = require('./myclass');
var module2 = require('./module-2');

console.log(myClass.method());
console.log(module2.method());
console.log(module2.method2());
```

1 **Load myclass.js.**

2 **Load module-2.js.**

Note that loading a local module always requires a path name—in these examples the path is just ./. Without it, Node will attempt to find a matching module in $NODE_PATH, and then ./node_modules, $HOME/.node_modules, $HOME/.node_libraries, or $PREFIX/lib/node.

In listing 2.4 notice that ./myclass is automatically expanded to ./myclass.js ❶, and ./module-2 is expanded to ./module-2.js ❷.

The output from this program would be as follows:

```
Hello
Hello
Hello again
```

Which module?

To determine the exact module Node will load, use `require.resolve(id)`. This will return a fully expanded filename.

Once a module is loaded, it'll be cached. That means that loading it multiple times will return the cached copy. This is generally efficient, and helps you heavily reuse modules within a project without worrying about incurring an overhead when using `require`. Rather than centrally loading all of the dependencies, you can safely call `require` on the same module.

Unloading modules

Although automatically caching modules fits many use cases in Node development, there may be rare occasions when you want to unload a module. The `require.cache` object makes this possible.

To remove a module from the cache, use the `delete` keyword. The full path of the module is required, which you can obtain with `require.resolve`. For example:

```
delete require.cache[require.resolve('./myclass')];
```

This should return `true`, which means the module was unloaded.

In the next technique you'll learn how to group related modules together and load them in one go.

TECHNIQUE 3 Loading a group of related modules

Node can treat directories as modules, offering opportunities for logically grouping related modules together.

Problem

You want to group related files together under a directory, and only have to load it with one call to `require`.

Solution

Create a file called index.js to load each module and export them as a group, or add a package.json file to the directory.

Discussion

Sometimes a module is logically self-contained, but it still makes sense to separate it into several files. Most of the modules you'll find on npm will be written this way. Node's module system supports this by allowing directories to act as modules. The easiest way to do this is to create a file called index.js that has a `require` statement to load each file. The following listing demonstrates how this works.

Listing 2.5 The group/index.js file

```
module.exports = {
  one: require('./one'),
  two: require('./two')
};
```

A module is exported that points to each file in the directory. (1)

The group/one.js and group/two.js files can then export values or methods (1) as required. The next listing shows an example of such a file.

Listing 2.6 The group/one.js file

```
module.exports = function() {
  console.log('one');
};
```

Code that needs to use a folder as a module can then use a single `require` statement to load everything in one go. The following listing demonstrates this.

Listing 2.7 A file loading the group of modules

```
var group = require('./group');

group.one();
group.two();
```

The call to require doesn't need any special handling to work with a directory of modules.

The output of listing 2.7 should look like this:

```
one
two
```

This approach is often used as an architectural technique to structure web applications. Related items, like controllers, models, and views, can be kept in separate folders to help separate concerns within the application. Figure 2.1 shows how to structure applications following this style.

Node also offers an alternative technique that supports this pattern. Adding a package.json file to a directory can help the module system figure out how to load all of the files in the directory at once. The JSON file should include a `main` property to point to a JavaScript file. This is actually the default file Node looks for when loading

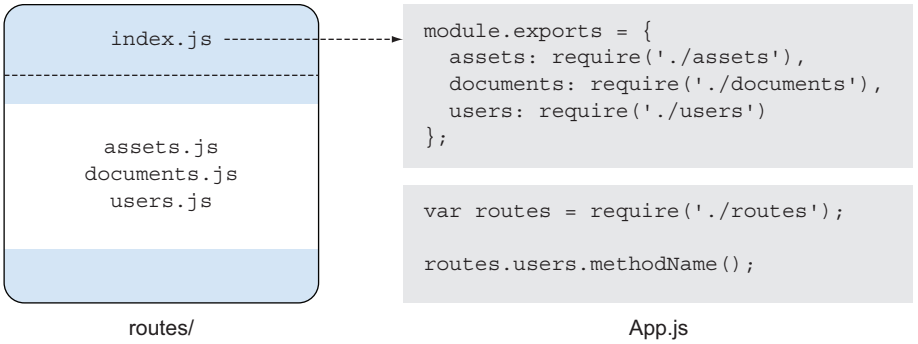


Figure 2.1 Folders as modules

modules—if no package.json is present, it'll then look for index.js. The next listing shows an example of a package.json file.

Listing 2.8 A package.json file for a directory containing a module

```
{ "name" : "group",
  "main" : "./index.js" }
```

This could point to any file.

File extensions

When loading a file, Node is configured to search for files with the .js, .json, and .node extensions. The `require.extensions` array can be used to tell `require` to load files with other extensions. Node's module system will take this into account when treating directories as modules, as well.

This feature is marked as deprecated in Node's documentation, but the module system is also marked as "locked" so it shouldn't go away. If you want to use it, you should check Node's documentation first.[1] If you're just trying to load a JavaScript file from a legacy system that has an unusual extension, then it might be suitable for experimentation.

The `require` API provides many ways to manage files. But what about when you want to load something relative to the current module, or the directory where the module is saved? Read on for an explanation in technique 4.

TECHNIQUE 4 Working with paths

Sometimes you need to open files based on the relative location. Node provides tools for determining the path to the current file, directory, and module.

Problem

You want to access a file that isn't handled by the module system.

[1] See http://nodejs.org/api/globals.html#globals_require_extensions.

■ **Solution**

Use `__dirname` or `__filename` to determine the location of the file.

■ **Discussion**

Sometimes you need to load data from a file that clearly shouldn't be handled by Node's module system, but you need to take the path of the current script into account—for example, a template in a web application. The `__dirname` and `__filename` variables are extremely useful in such cases.

Running the following listing will print the output of these values.

Listing 2.9 Path variables

```
console.log('__dirname:', __dirname);
console.log('__filename:', __filename);
```

These variables point to the fully resolved locations of the current script.

Most developers join these variables with path fragments using simple string concatenation: `var view = __dirname + '/views/view.html';`. This works with both Windows and Unix—the Windows APIs are clever enough to automatically switch the slashes to the native format, so you don't need special handling to support both operating systems.

If you really want to ensure paths are joined correctly, you can use the `path.join` method from Node's `path` module: `path.join(__dirname, 'views', 'view.html');`.

Apart from module management, there are globally available objects for writing to the standard I/O streams. The next set of techniques explores `process.stdout` and the `console` object.

2.2 Standard I/O and the console object

Text can be piped to a Node process by using command-line tools in Unix or Windows. This section includes techniques for working with these standard I/O streams, and also how to correctly use the `console` object for a wide range of logging-related tasks.

TECHNIQUE 5 Reading and writing to standard I/O

Whenever you need to get data into and out of a program, one useful technique is using the `process` object to read and write to standard I/O streams.

■ **Problem**

You want to pipe data to and from a Node program.

■ **Solution**

Use `process.stdout` and `process.stdin`.

■ **Discussion**

The `process.stdout` object is a writable stream to `stdout`. We'll look at streams in more detail in chapter 5, but for now you just need to know it's part of the `process` object that every Node program has access to, and is helpful for displaying and receiving text input.

The next listing shows how to pipe text from another command, process it, and output it again.

Listing 2.10 Path variables

```
// Run with:
//   cat file.txt | node process.js

process.stdin.resume();
process.stdin.setEncoding('utf8');

process.stdin.on('data', function(text) {
  process.stdout.write(text.toUpperCase());
});
```

Tell the stream we're ready to start reading.

This callback transforms the data in chunks when they're available.

Every time a chunk of text is read from the input stream, it'll be transformed with `toUpperCase()` and then written to the output stream. Figure 2.2 shows how data flows from one operating system process, through your Node program, and then out through another program. In the terminal, these programs would be linked together with the pipe (`|`) symbol.

This *pipe*-based approach works well when dealing with input in Unix, because many other commands are designed to work this way. That brings a LEGO-like modularity to Node programs that facilitates reuse.

If you just want to print out messages or errors, Node provides an easier API specifically tailored for this purpose through the `console` object. The next technique explains how to use it, and some of its less obvious features.

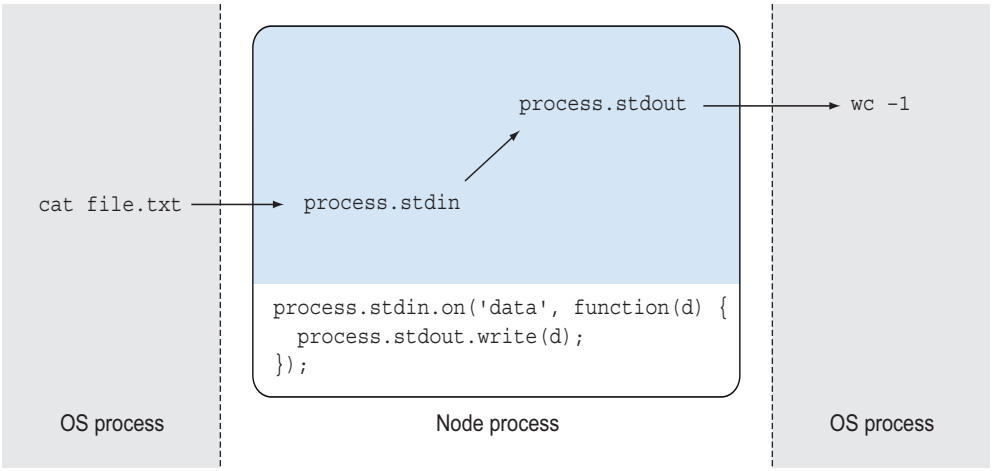


Figure 2.2 Data flows in a simple program that uses `stdio`.

TECHNIQUE 6 Logging messages

The easiest way to log information and errors from a program is by using the `console` object.

Problem

You want to log different types of messages to the console.

Solution

Use `console.log`, `console.info`, `console.error`, and `console.warn`. Be sure to take advantage of the built-in formatting facilities provided by these methods.

Discussion

The `console` object has several methods that can be used to output different types of messages. They'll be written to the relevant output stream, which means you can pipe them accordingly on a Unix system.

Although the basic usage is `console.log('message')`, more functionality is packed in. Variables can be interpolated, or simply appended alongside string literals. This makes it extremely easy to log messages that display the contents of primitive values or objects. The following listing demonstrates these features.

Listing 2.11 Path variables

```
var name = 'alex';
var user = { name: 'alex' };

console.log('Hello');
console.log('Hello %s', name);
console.log('Hello:', name);
console.log('Hello:', user);

console.error('Error, bad user:', user);
```

Simple variable interpolation can be used with strings or numbers.

A space will automatically be added after the colon.

The output of listing 2.11 looks like this:

```
Hello
Hello alex
Hello: alex
Hello: { name: "alex" } //
 Error, bad user: { name: 'alex' }
```

The user object is internally formatted using util.inspect.

When message strings are formatted, `util.format` is used. Table 2.1 shows the supported formatting placeholders.

Table 2.1 Formatting placeholders

Placeholder	Type	Example
`%s`	`String`	`'%s', 'value'`
`%d`	`Number`	`'%f', 3.14`
`%j`	`JSON`	`'%j', { name: 'alex' }`

These formatting placeholders are convenient, but just being able to simply include objects in `console.log` messages without manually appending strings is a handy way to log messages.

The `info` and `warn` methods are synonyms for `log` and `error`. The difference between `log` and `error` is the output stream used. In technique 5, you saw how Node makes standard input and output streams available to all programs. It also exposes the standard error stream through `process.stderr`. The `console.error` method will write to this stream, rather than `process.stdout`. This means you can redirect a Node process's `error` messages in the terminal or in a shell script.

If you ran the previous listing with `2> error-file.log`, the error messages would be redirected to `error-file.log`. The other messages would be printed to the console as usual:

```
node listings/globals/console-1.js 2> errors-file.log
```

The `2` handle refers to the error stream; `1` is standard output. That means you could redirect errors to a log file without having to open files within your Node program, or use a specific logging module. Good old-fashioned shell redirection is good enough for many projects.

Standard streams

Standard streams come in three flavors: stdin, stdout, and stderr. In Unix terminals, these are referred to with numbers. `0` is used for standard input, `1` is standard output, and `2` is standard error.

The same applies to Windows: running a program from the command prompt and adding `2> errors-file.log` will send the error messages to `errors-file.log`, just like Unix.

Stack traces

Another feature of the `console` object is `console.trace()`. This method generates a stack trace at the current point of execution. The generated stack trace includes line numbers for the code that invokes asynchronous callbacks, which can help when reporting errors that would otherwise be difficult to track down. For example, a trace generated inside an event listener will show where the event was triggered from. Technique 28 in chapter 5 explores this in more detail.

Another slightly more advanced use of `console` is its benchmarking feature. Continue reading for a detailed look.

TECHNIQUE 7 Benchmarking a program

Node makes it possible to benchmark programs without any additional tools.

Problem

You need to benchmark a slow operation.

Solution

Use `console.time()` and `console.timeEnd()`.

Discussion

In your career as a Node programmer, there will come a time when you're trying to determine why a particular operation is slow. Fortunately, the `console` object comes with some built-in benchmarking features.

Invoking `console.time('label')` records the current time in milliseconds, and then later calling `console.timeEnd('label')` displays the duration from that point. The time in milliseconds will be automatically printed alongside the label, so you don't have to make a separate call to `console.log` to print a label.

Listing 2.12 is a short program that accepts command-line arguments (see technique 9 for more on handling arguments), with benchmarking to see how fast the file input is read.

Listing 2.12 Benchmarking a function

```
var args = {
  '-h': displayHelp,
  '-r': readFile
};

function displayHelp() {
  console.log('Argument processor:', args);
}

function readFile(file) {
  if (file && file.length) {
    console.log('Reading:', file);
    console.time('read');
    var stream = require('fs').createReadStream(file)
    stream.on('end', function() {
      console.timeEnd('read');
    });
    stream.pipe(process.stdout);
  } else {
    console.error('A file must be provided with the -r option');
    process.exit(1);
  }
}

if (process.argv.length > 0) {
  process.argv.forEach(function(arg, index) {
    args[arg].apply(this, process.argv.slice(index + 1));
  });
}
```

Calling console.timeEnd() will cause the benchmark to be displayed.

Using several interleaved calls to `console.time` with different labels allows multiple benchmarks to be performed, which is perfect for exploring the performance of complex, nested asynchronous programs.

These functions calculate duration based on `Date.now()`, which gives accuracy in milliseconds. To get more accurate benchmarks, the third-party `benchmark` module

(https://npmjs.org/package/benchmark) can be used in conjunction with `microtime` (https://npmjs.org/package/microtime).

The `process` object is used for working with standard I/O streams, and used correctly, `console` handles many of the tasks that the uninitiated may tackle with third-party modules. In the next section we'll further explore the `process` object to look at how it helps integrate with the wider operating system.

2.3 Operating system and command-line integration

The `process` object can be used to obtain information about the operating system, and also communicate with other processes using exit codes and signal listeners. This section contains some more-advanced techniques for using these features.

TECHNIQUE 8 Getting platform information

Node has some built-in methods for querying operating system functionality.

Problem

You need to run platform-specific code based on the operating system or processor architecture.

Solution

Use the `process.arch` and `process.platform` properties.

Discussion

Node JavaScript is generally portable, so it's unlikely that you'll need to branch based on the operating system or process architecture. But you may want to tailor projects to take advantage of operating system–specific features, or simply collect statistics on what systems a script is executing on. Certain Windows-based modules that include bindings to binary libraries could switch between running a 32- or 64-bit version of a binary. The next listing shows how this could be supported.

Listing 2.13 Branching based on architecture

```
switch (process.arch) {
  case 'x64':
    require('./lib.x64.node');
    break;
  case 'ia32':
    require('./lib.Win32.node');
    break;
  default:
    throw new Error('Unsupported process.arch:', process.arch);
}
```

Other information from the system can also be gleaned through the `process` module. One such method is `process.memoryUsage()`—it returns an object with three properties that describe the process's current memory usage:

- `rss`—The *resident set size*, which is the portion of the process's memory that is held in RAM

- `heapTotal`—Available memory for dynamic allocations
- `heapUsed`—Amount of heap used

The next technique explores handling command-line arguments in more detail.

TECHNIQUE 9 Passing command-line arguments

Node provides a simple API to command-line arguments that you can use to pass options to programs.

Problem

You're writing a program that needs to receive simple arguments from the command line.

Solution

Use `process.argv`.

Discussion

The `process.argv` array allows you to check if any arguments were passed to your script. Because it's an array, you can use it to see how many arguments were passed, if any. The first two arguments are `node` and the name of the script.

Listing 2.14 shows just one way of working with `process.argv`. This example loops over `process.argv` and then slices it to "parse" argument flags with options. You could run this script with `node arguments.js -r arguments.js` and it would print out its own source.

Listing 2.14 Manipulating command-line arguments

```
var args = {
  '-h': displayHelp,
  '-r': readFile
};

function displayHelp() {
  console.log('Argument processor:', args);
}

function readFile(file) {
  console.log('Reading:', file);
  require('fs').createReadStream(file).pipe(process.stdout);
}

if (process.argv.length > 0) {
  process.argv.forEach(function(arg, index) {
    args[arg].apply(this, process.argv.slice(index + 1));
  });
}
```

1 This is a simple object used to model the valid arguments.

2 Pipe out a file through the standard output stream.

3 Call a matching method from the arg parameter model, and slice the full list of arguments to effectively support passing options from command-line flags.

The `args` object 1 holds each switch that the script supports. Then `createReadStream` is used 2 to `pipe` the file to the standard output stream. Finally, the function referenced by the command-line switch in `args` is executed using `Function.prototype.apply` 3.

Although this is a toy example, it illustrates how handy `process.argv` can be without relying on a third-party module. Since it's a JavaScript `Array`, it's extremely easy to work with: you can use methods like `map`, `forEach`, and `slice` to process arguments with little effort.

Complex arguments

For more complex programs, use an option parsing module. The two most popular are `optimist` (https://npmjs.org/package/optimist) and `commander` (https://npmjs.org/package/commander). `optimist` converts arguments into an `Object`, which makes them easier to manipulate. It also supports default values, automatic usage generation, and simple validation to ensure certain arguments have been provided. `commander` is slightly different: it uses an abstracted notion of a *program* that allows you to specify your program's accepted arguments using a chainable API.

Good Unix programs handle arguments when needed, and they also exit by returning a suitable status code. The next technique presents how and when to use `process.exit` to signal the successful—or unsuccessful—completion of a program.

TECHNIQUE 10 Exiting a program

Node allows you to specify an exit code when a program terminates.

Problem

Your Node program needs to exit with specific status codes.

Solution

Use `process.exit()`.

Discussion

Exit status codes are significant in both Windows and Unix. Other programs will examine the exit status to determine whether a program ran correctly. This becomes more important when writing Node programs that take part in larger systems, and helps with monitoring and debugging later on.

By default, a Node program returns a `0` exit status. This means the program ran and terminated correctly. Any non-zero status is considered an error. In Unix, this status code is generally accessed by using `$?` in a shell. The Windows equivalent is `%errorlevel%`.

Listing 2.15 shows a modification to listing 2.14 that causes the program to exit cleanly with a relevant status code when no filename is specified with the `-r` option.

Listing 2.15 Returning meaningful exit status codes

```
var args = {
  '-h': displayHelp,
  '-r': readFile
};

function displayHelp() {
```

```
  console.log('Argument processor:', args);
}

function readFile(file) {
  if (file && file.length) {
    console.log('Reading:', file);
    require('fs').createReadStream(file).pipe(process.stdout);
  } else {
    console.error('A file must be provided with the -r option');
    process.exit(1);
  }
}

if (process.argv.length > 0) {
  process.argv.forEach(function(arg, index) {
    args[arg].apply(this, process.argv.slice(index + 1));
  });
}
```

Both console.error and process.exit are used to correctly indicate an error occurred. (1)

After running listing 2.15, typing `echo $?` in a Unix terminal will display `1`. Also note that `console.error` (1) is used to output an error message. This will cause the message to be written to `process.stderr`, which allows users of the script to easily pipe error messages somewhere.

Exit codes with special meanings

In the Advanced Bash-Scripting Guide (http://tldp.org/LDP/abs/html/index.html), a page is dedicated to status codes called Exit Codes With Special Meanings (http://tldp.org/LDP/abs/html/exitcodes.html). This attempts to generalize error codes, although there's no standard list of status codes for scripting languages, outside of non-zero indicating an error occurred.

Because many Node programs are asynchronous, there are times when you may need to specifically call `process.exit()` or close down an I/O connection to cause the Node process to end gracefully. For example, scripts that use the Mongoose database library (http://mongoosejs.com/) need to call `mongoose.connection.close()` before the Node process will be able to exit.

You may need to track the number of pending asynchronous operations in order to determine when it's safe to call `mongoose.connection.close()`, or the equivalent for another database module. Most people do this using a simple counter variable, incrementing it just before asynchronous operations start, and then decrementing it once their callbacks fire. Once it reaches `0`, it'll be safe to close the connection.

Another important facet to developing correct programs is creating signal handlers. Continue reading to learn how Node implements signal handlers and when to use them.

TECHNIQUE 11 Responding to signals

Node programs can respond to signals sent by other processes.

Problem

You need to respond to signals sent by other processes.

Solution

Use the signal events that are sent to the `process` object.

Discussion

Most modern operating systems use signals as a way of sending a simple message to a program. Signal handlers are typically used in programs that run in the background, because it might be the only way of communicating with them. There are other cases where they can be useful in the kinds of programs you're most likely write—consider a web application that cleanly closes its connection to a database when it receives `SIGTERM`.

The `process` object is an `EventEmitter`, which means you can add event listeners to it. Adding a listener for a POSIX signal name should work—on a Unix system, you can type `man sigaction` to see the names of all of the signals.

Signal listeners enable you to cater to the expected behavior of Unix programs. For example, many servers and daemons will reload configuration files when they receive a `SIGHUP` signal. The next listing shows how to attach a listener to `SIGHUP`.

Listing 2.16 Adding a listener for a POSIX signal

```
process.stdin.resume();
process.on('SIGHUP', function () {
  console.log('Reloading configuration...');
});

console.log('PID:', process.pid);
```

Binding a listener to the SIGHUP signal. (2)

(1) Read from stdin so the program will run until CTRL-C is pressed or it's killed.

(3) The PID is displayed so you can use it to send signals using the kill command.

Before doing anything with standard input, `resume` should be called (1) to prevent Node from exiting straight away. Next, a listener is added to the `SIGHUP` event on the `process` object (2). Finally, the PID is displayed for the current process (3).

Once the program in listing 2.16 is running, it'll display the process's PID. The PID can be used with the `kill` command to send the process signals. For example, `kill -HUP 94962` will send the `HUP` signal to PID `94962`. If you send another signal, or just type `kill 94962`, then the process will exit.

It's important to realize that signals can be sent from any process to any other, permissions notwithstanding. Your Node process can send another process a signal by using `process.kill(pid, [signal])`—in this case `kill` doesn't mean the process will be "killed," but simply sent a given signal. The method is named `kill` after the C standard library function in `signal.h`.

Figure 2.3 shows a broad overview of how signals originate from any process in an operating system and can be received by your Node processes.

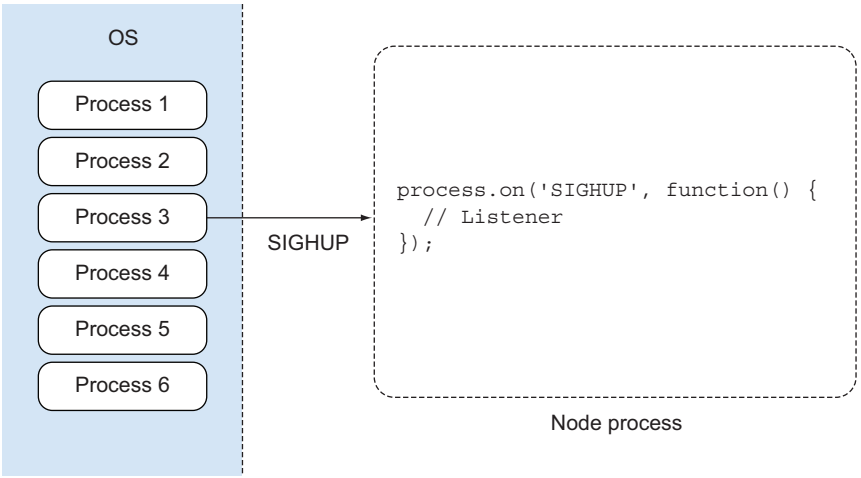


Figure 2.3 Signals originate from a process, and are handled with an event listener.

You don't have to respond to signals in your Node programs, but if you're writing a long-running network server, then signal listeners can be extremely useful. Supporting signals like `SIGHUP` will make your program fit into existing systems more naturally.

A large part of Node's appeal is its asynchronous APIs and non-blocking I/O features. Sometimes it's desirable to fake this behavior—perhaps in automated tests—or simply to just force code to execute later on. In the next section we'll look at how Node implements JavaScript timers, which support this type of functionality.

2.4 Delaying execution with timers

Node implements the JavaScript timer functions `setTimeout`, `setInterval`, `clearTimeout`, and `clearInterval`. These functions are globally available. Although they're part of JavaScript as defined by Mozilla, they're not defined in the ECMAScript standard. Instead, timers are part of the HTML DOM Level 0 specification.

TECHNIQUE 12 Executing functions after a delay with setTimeout

It's possible to run code once after a delay using Node's `setTimeout` global method.

Problem

You want to execute a function after a delay.

Solution

Use `setTimeout`, and use `Function.prototype.bind` if necessary.

Discussion

The most basic usage of `setTimeout` is simple: pass it a function to execute and a delay in milliseconds:

```
setTimeout(function() {
  console.log('Hello from the past!');
}, 1000);
```

This seems simple and contrived, but you'll see it used most commonly in tests where asynchronous APIs are being tested and a small delay is necessary to simulate real-world behavior. Node supports JavaScript timers for just such cases.

Methods can also easily be passed to `setTimeout` by using `Function.prototype.bind`. This can be used to bind the first argument to `this`, or more often the object that the method belongs to. The following listing shows how `bind` can be used with a simple object.

Listing 2.17 Combining `setTimeout` with `Function.prototype.bind`

```
function Bomb() {
  this.message = 'Boom!';
}

Bomb.prototype.explode = function() {
  console.log(this.message);
};

var bomb = new Bomb();

setTimeout(bomb.explode.bind(bomb), 1000);
```

Call .bind to ensure the method is bound correctly so it can access internal properties.

Binding ensures that the code inside the method can access the object's internal properties. Otherwise, `setTimeout` would cause the method to run with `this` bound to the global object. Binding a method can be more readable than creating a new anonymous function.

To cancel scheduled functions, retain a reference to the `timeoutId` returned by `setTimeout` and then call `clearTimeout(timeoutId)` (1). The next listing demonstrates `clearTimeout`.

Listing 2.18 Using `clearTimeout` to prevent scheduled functions

```
function Bomb() {
  this.message = 'Boom!';
}

Bomb.prototype.explode = function() {
  console.log(this.message);
};

var bomb = new Bomb();

var timeoutId = setTimeout(bomb.explode.bind(bomb), 1000);

clearTimeout(timeoutId);
```

Defuse the bomb by calling clearTimeout to prevent bomb.expode from running. (1)

When exactly does the callback run?

Although you can specify when a callback runs in milliseconds, Node isn't quite *that* precise. It can guarantee that the callback will run *after* the specified time, but it may be slightly late.

As well as delaying execution, you can also call functions periodically. The next technique discusses how to achieve this by using `setInterval`.

TECHNIQUE 13 Running callbacks periodically with timers

Node can also run callbacks at regular intervals using `setInterval`, which works in a fashion similar to `setTimeout`.

Problem

You want to run a callback at a regular interval.

Solution

Use `setInterval`, and `clearInterval` to stop the timer.

Discussion

The `setInterval` method has been around for years in browsers, and it behaves in Node much like the client-side counterparts. The callback will be executed on or just after the specified delay, and will run in the event loop just after I/O (and any calls to `setImmediate`, as detailed in technique 14).

The next listing shows how to combine `setInterval` with `setTimeout` to schedule two functions to execute in a sequence.

Listing 2.19 Using `setInterval` and `setTimeout` together

```
function tick() {
  console.log('tick:', Date.now());
}

function tock() {
  console.log('tock:', Date.now());
}

setInterval(tick, 1000);

setTimeout(function() {
  setInterval(tock, 1000);
}, 500);
```

❶ Run another setInterval after the first one.

The `setInterval` method itself returns a reference to the timer, which can be stopped by calling `clearInterval` and passing the reference. Listing 2.19 uses a second call to `setTimeout` ❶ to trigger a second interval timer that runs 500 milliseconds after the first.

Because `setInterval` prevents a program from exiting, there are cases where you might want to exit a program if it isn't doing anything else. For example, let's say you're running a program that should exit when a complex operation has finished, and you'd like to monitor it at regular intervals using `setInterval`. Once the complex operation has finished, you don't want to monitor it any more.

Rather than calling `clearInterval`, Node 0.10 allows you to call `timerRef.unref()` at any time before the complex operation has finished. This means you can use `setTimeout` or `setInterval` with operations that don't signal their completion.

Listing 2.20 uses `setTimeout` to simulate a long-running operation that will keep the program running while the timer displays the process's memory usage. Once the timeout's delay has been reached, the program will exit *without* calling `clearTimeout`.

Listing 2.20 Keeping a timer alive until the program cleanly exits

```
function monitor() {
  console.log(process.memoryUsage());
}

var id = setInterval(monitor, 1000);
id.unref();

setTimeout(function() {
  console.log('Done!');
}, 5000);
```

Tell Node to stop the interval when the program has finished the long-running operation.

This is extremely useful in situations where there isn't a good place to call `clearInterval`.

Once you've mastered timers, you'll encounter cases where it's useful to run a callback after the briefest possible delay. Using `setTimeout` with a delay of zero isn't the optimum solution, even though it seems like the obvious strategy. In the next technique you'll see how to do this correctly in Node by using `process.nextTick`.

TECHNIQUE 14 Safely managing asynchronous APIs

Sometimes you want to delay an operation just slightly. In traditional JavaScript, it might be acceptable to use `setTimeout` with a small delay value. Node provides a more efficient solution: `process.nextTick`.

Problem

You want to write a method that returns an instance of `EventEmitter` or accepts a callback that *sometimes* makes an asynchronous API call, but not in all cases.

Solution

Use `process.nextTick` to wrap the synchronous operation.

Discussion

The `process.nextTick` method allows you to place a callback at the head of the next cycle of the run loop. That means it's a way of *slightly* delaying something, and as a result it's more efficient than just using `setTimeout` with a zero delay argument.

It can be difficult to visualize why this is useful, but consider the following example. Listing 2.21 shows a function that returns an `EventEmitter`. The idea is to provide an event-oriented API, allowing users of the API to subscribe to events as needed, while being able to run asynchronous calls internally.

Listing 2.21 Incorrectly triggering asynchronous methods with events

```
var EventEmitter = require('events').EventEmitter;

function complexOperations() {
```

```
  var events = new EventEmitter();

  events.emit('success'); 1((callout-globals-nexttick-1))

  return events;
}

complexOperations().on('success', function() {
  console.log('success!');
});
```

1 This is an event that is triggered outside of any asynchronous callbacks.

Running this example will fail to trigger the `success` listener 1 at the end of the example. Why is this the case? Well, the event is emitted before the listener has been subscribed. In most cases, events would be emitted inside callbacks for some asynchronous operation or another, but there are times when it makes sense to emit events early—perhaps in cases where arguments are validated and found to contain errors, so `error` can be emitted very quickly.

To correct this subtle flaw, any sections of code that emit events can be wrapped in `process.nextTick`. The following listing demonstrates this by using a function that returns an instance of `EventEmitter`, and then emits an event.

Listing 2.22 Triggering events inside `process.nextTick`

```
var EventEmitter = require('events').EventEmitter;

function complexOperations() {
  var events = new EventEmitter();

  process.nextTick(function() {
    events.emit('success');
  });

  return events;
}

complexOperations().on('success', function() {
  console.log('success!');
});
```

The event will now be emitted when the listener is ready.

Node's documentation recommends that APIs should always be 100% asynchronous or synchronous. That means if you have a method that accepts a callback and *may* call it asynchronously, then you should wrap the synchronous case in `process.nextTick` so users can rely on the order of execution.

Listing 2.23 uses an asynchronous call to read a file from the disk. Once it has read the file, it'll keep a cached version in memory. Subsequent calls will return the cached version. When returning the cached version, `process.nextTick` is used so the API still behaves asynchronously. That makes the output in the terminal read in the expected order.

Listing 2.23 Creating the illusion of an always asynchronous API

```
var EventEmitter = require('events').EventEmitter;
var fs = require('fs');
var content;

function readFileIfRequired(cb) {
  if (!content) {
    fs.readFile(__filename, 'utf8', function(err, data) {
      content = data;
      console.log('readFileIfRequired: readFile');
      cb(err, content);
    });
  } else {
    process.nextTick(function() {
      console.log('readFileIfRequired: cached');
      cb(null, content);
    });
  }
}

readFileIfRequired(function(err, data) {
  console.log('1. Length:', data.length);

  readFileIfRequired(function(err, data2) {
    console.log('2. Length:', data2.length);
  });

  console.log('Reading file again...');
});

console.log('Reading file...');
```

1 If the content hasn't been read into memory, read it asynchronously.

2 If the content has been read, pass the cached version to the callback, but first use process.nextTick to ensure the callback is executed later.

3 Make subsequent calls to the asynchronous operation to ensure it behaves as expected.

In this example, a file is cached to memory by using `fs.readFile` to read it 1, and then return a copy of it 2 for every subsequent call. This is wrapped in a process that's called multiple times 3 so you can compare the behavior of the non-blocking file system operation to `process.nextTick`.

Visualizing the event loop: setImmediate and process.maxTickDepth

The `setImmediate` and `clearImmediate` global functions accept a callback and optional arguments, and will run *after* any upcoming I/O events but *before* `setTimeout` and `setInterval`.

Callbacks added this way are pushed onto a queue, and one callback will be executed per run loop. This is different from `process.nextTick`, which causes `process.maxTickDepth` callbacks to run per iteration of the run loop.

The callbacks that have been passed with `process.nextTick` are usually run at the end of the current event loop. The number of callbacks that can be safely run is controlled by `process.maxTickDepth`, which is 1000 by default to allow I/O operations to continue to be handled.

Figure 2.4 illustrates how each of the timer functions is positioned within a single iteration of the event loop.

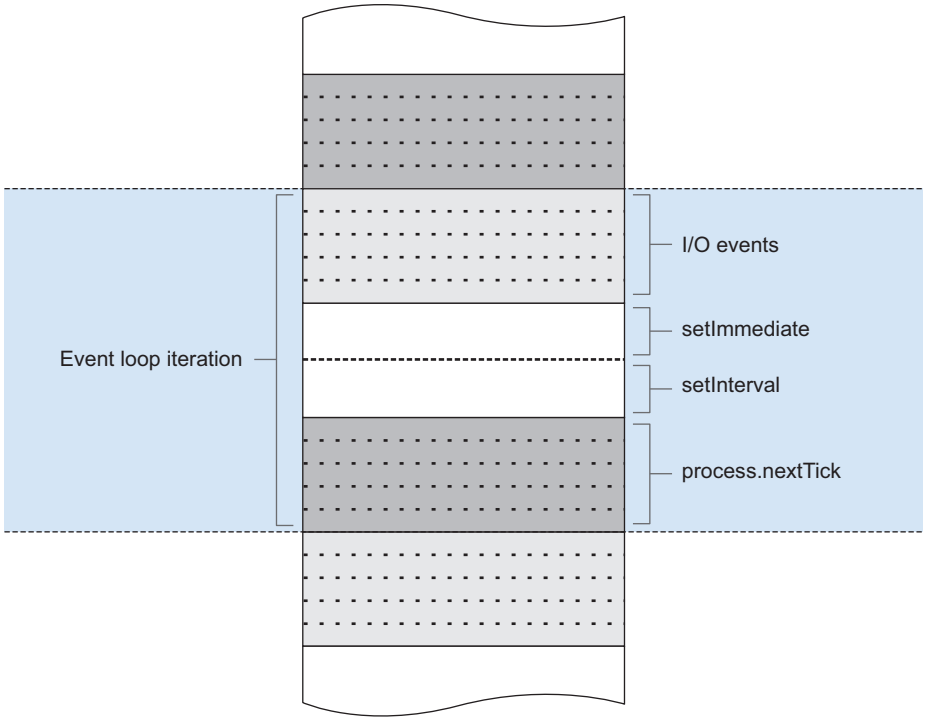


Figure 2.4 Scheduling `nextTick` on the event loop

When you're creating your own classes and methods that behave asynchronously, keep behavior consistent and predictable by using `process.nextTick`.

Node's implementation of the standard browser-based JavaScript timers fits in well with its event loop and non-blocking I/O. Although these functions are typically used for testing asynchronous code, a deep understanding of when `setTimeout`, `setImmediate`, and `process.nextTick` will be executed provides mastery over the event loop.

2.5 Summary

In this chapter you've seen some of the surprisingly powerful things that are built into Node programs without going to the trouble of loading a module. The next time you want to group related modules together, you can create an index.js file, as described in technique 3. And if you need to read standard input, you can use the `process` object's `stdin` property (technique 5).

In addition to the `process` object, there's also the often overlooked `console` object, which will help you debug and maintain programs (technique 6).

In the next chapter you'll learn about buffers. Buffers are great for working with binary data, which has traditionally been seen as a weakness of JavaScript. Buffers also underpin some of Node's powerful features such as streams.

3 Buffers: Working with bits, bytes, and encodings

This chapter covers

- Introduction to the Buffer data type
- Changing data encodings
- Converting binary files to JSON
- Creating your own binary protocol

JavaScript has historically had subpar binary support. Typically, parsing binary data would involve various tricks with strings to extract the data you want. Not having a good mechanism to work with raw memory in JavaScript was one of the problems Node core developers had to tackle when the project started getting traction. This was mostly for performance reasons. All of the raw memory accumulated in the `Buffer` data type.

Buffers are raw allocations of the heap, exposed to JavaScript in an array-like manner. They're exposed globally and therefore don't need to be required, and can be thought of as just another JavaScript type (like `String` or `Number`):

Allocate 255 bytes.

```
var buf = new Buffer(255);
buf[0] = 23;
```

Write integer 23 to the first byte.

If you haven't worked much with binary data, don't worry; this chapter is designed to be friendly to newcomers but also equip those who are more familiar with the concept. We'll cover simple and more advanced techniques:

- Converting a `Buffer` to different encodings
- Using the `Buffer` API to transform a binary file to JSON
- Encoding and decoding your own binary protocol

Let's look first at changing encodings for buffers.

3.1 Changing data encodings

If no encoding is given, file operations and many network operations will return data as a `Buffer`. Take this `fs.readFile` as an example:

```
var fs = require('fs');
fs.readFile('./names.txt', function (er, buf) {
  Buffer.isBuffer(buf); // true
});
```

isBuffer returns true if it's a Buffer.

But many times you already know a file's encoding, and it's more useful to get the data as an encoded string instead. We'll look at converting between `Buffers` and other formats in this section.

TECHNIQUE 15 Converting buffers into other formats

By default, Node's core APIs return a buffer unless an encoding is specified. But buffers easily convert to other formats. In this next technique we'll look at how to convert buffers.

Problem

You want to turn a `Buffer` into plain text.

Solution

The `Buffer` API allows you to convert a `Buffer` into a string value.

Discussion

Let's say we have a file that we know is just plain text. For our purposes we'll call this file names.txt and it will include a person's name on each line of the file:

```
Janet
Wookie
Alex
Marc
```

If we were to load the file using a method from the file system (`fs`) API, we'd get a `Buffer` (`buf`) by default

```
var fs = require('fs');
fs.readFile('./names.txt', function (er, buf) {
  console.log(buf);
});
```

which, when logged out, is shown as a list of octets (using hex notation):

```
<Buffer 4a 61 6e 65 74 0a 57 6f 6f 6b 69 65 0a 41 6c 65 78 0a
                      4d 61 72 63 0a>
```

This isn't very useful since we know that the file is plain text. The `Buffer` class provides a method called `toString` to convert our data into a UTF-8 encoded string:

```
var fs = require('fs');
fs.readFile('./names.txt', function (er, buf) {
  console.log(buf.toString());
});
```

toString by default will convert data into a UTF-8 encoded string.

This will yield the same output as our original file:

```
Janet
Wookie
Alex
Marc
```

But since we know that this data is only comprised of ASCII characters,[1] we could also get a performance benefit by changing the encoding to ASCII rather than UTF-8. To do this, we provide the type of encoding as the first argument for `toString`:

```
var fs = require('fs');
fs.readFile('./names.txt', function (er, buf) {
  console.log(buf.toString('ascii'));
});
```

toString accepts an encoding as the first argument.

The `Buffer` API provides other encodings such as `utf16le`, `base64`, and `hex`, which you can learn more about by viewing the `Buffer` API online documentation.[2]

TECHNIQUE 16 Changing string encodings using buffers

In addition to converting buffers, you can also utilize buffers to turn one string encoding into another.

Problem

You want to change from one string encoding to another.

Solution

The Node `Buffer` API provides a mechanism to change encodings.

Discussion

Example 1: Creating a Basic authentication header

Sometimes it's helpful to build a string of data and then change its encoding. For example, if you wanted to request data from a server that uses Basic authentication,[3] you'd need to send the username and password encoded using Base64:

```
Authorization: Basic am9obm55OmMtYmFk
```

am9obm55OmMtYmFk is encoded credentials

[1] See http://en.wikipedia.org/wiki/ASCII.
[2] See http://nodejs.org/api/buffer.html.
[3] See http://en.wikipedia.org/wiki/Basic_access_authentication.

Before Base64 encoding is applied, Basic authentication credentials combine the username and password, separating the two using a : (colon). For our example, we'll use `johnny` as the username and `c-bad` as the password:

```
var user = 'johnny';
var pass = 'c-bad';
var authstring = user + ':' + pass;
```

username and password are separated using colon

Now we have to convert this into a `Buffer` in order to change it into another encoding. Buffers can be allocated by bytes, as we've seen already by simply passing in a number (for example, `new Buffer(255)`). They also can be allocated by passing in string data:

```
var buf = new Buffer(authstring);
```

String data converted to a Buffer

Specifying an encoding

When strings are used to allocate a `Buffer`, they're assumed to be UTF-8 strings, which is typically what you want. But you can specify the encoding of the incoming data using a second, optional, encoding argument:

```
new Buffer('am9obm55OmMtYmFk', 'base64')
```

Now that we have our data as a `Buffer`, we can turn it back into a Base64-encoded string by using `toString('base64')`:

```
var encoded = buf.toString('base64');
```

Result: am9obm55OmMtYmFk

This process can be compacted as well, since instance methods can be called on the returned `Buffer` instance right away and the `new` keyword can be omitted:

```
var encoded = Buffer(user + ':' + pass).toString('base64');
```

Example 2: Working with data URIs

Data URIs[4] are another example of when using the `Buffer` API can be helpful. Data URIs allow a resource to be embedded inline on a web page using the following scheme:

```
data:[MIME-type][;charset=<encoding>[;base64],<data>
```

For example, this PNG image of a monkey can be represented as a data URI:

```
...
```

And when read in the browser, the data URI will display our primate as shown in figure 3.1.

Figure 3.1 Data URI read in a browser displays the monkey as an image

Let's look at how we can create a data URI using the `Buffer` API. In our primate example, we were using a PNG image that has the MIME type of `image/png`:

```
var mime = 'image/png';
```

[4] See http://en.wikipedia.org/wiki/Data_URI_scheme.

Binary files can be represented in data URIs using Base64 encoding, so let's set up a variable for that:

```
var encoding = 'base64';
```

With our MIME type and encoding, we can construct the start of our data URI:

```
var mime = 'image/png';
var encoding = 'base64';
var uri = 'data:' + mime + ';' + encoding + ',';
```

We need to add the actual data next. We can use `fs.readFileSync` to read in our data synchronously and return the data inline. `fs.readFileSync` will return a `Buffer`, so we can then convert that to a Base64 string:

```
var encoding = 'base64';
var data = fs.readFileSync('./monkey.png').toString(encoding);
```

Let's put this all together and make a program that will output our data URI:

```
var fs = require('fs');
var mime = 'image/png';
var encoding = 'base64';
var data = fs.readFileSync('./monkey.png').toString(encoding);
var uri = 'data:' + mime + ';' + encoding + ',' + data;
console.log(uri);
```

Require fs module to use fs.readFileSync

Output data URI

Construct data URI

The output of this program will be

```
...
```

Let's flip the scenario around. What if you have a data URI but you want to write it out to an actual file? Again, we'll work with our monkey example. First, we `split` the array to grab only the data:[5]

```
var uri = '...';
var data = uri.split(',')[1];
```

We can then create a `Buffer` using our `data` string and specifying the encoding:

```
var buf = Buffer(data, 'base64');
```

Next, we use `fs.writeFileSync` to write this synchronously to disk, giving it a file name and the `Buffer`:

```
fs.writeFileSync('./secondmonkey.png', buf);
```

Putting this example all together looks like this:

```
var fs = require('fs');
var uri = '...';
var data = uri.split(',')[1];
var buf = Buffer(data, 'base64');
fs.writeFileSync('./secondmonkey.png', buf);
```

Require fs module to use fs.writeFileSync

[5] This is not prescriptive for all data URIs, as commas could appear elsewhere.

When opened in our default image viewer, this gives us our monkey, as shown in figure 3.2.

Most of the time, when you deal with `Buffer` objects in Node, it will be to convert them to other formats, and sometimes you'll change encodings. But you may find yourself having to deal with a binary file format, and the `Buffer` API—which we'll look at next—provides a rich set of tools to work with that as well.

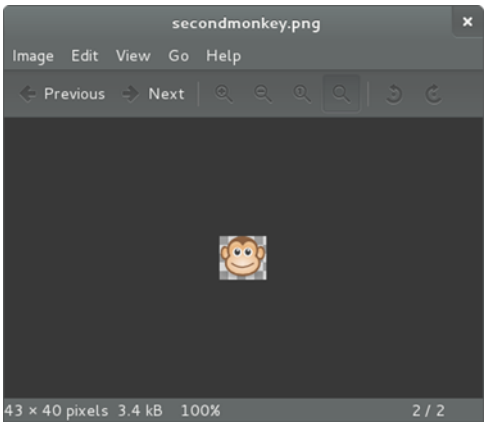


Figure 3.2 Generated secondmonkey.png file from a data URI

3.2 Converting binary files to JSON

Working with binary data is kind of like solving a puzzle. You're given clues by reading a specification of what the data means and then you have to go out and turn that data into something usable in your application.

TECHNIQUE 17 Using buffers to convert raw data

What if you could utilize a binary format to do something useful in your Node program? In this technique we'll cover, in depth, working with binary data to convert a common file format into JSON.

Problem

You want to convert a binary file into a more usable format.

Solution

The Node API extends JavaScript with a `Buffer` class, exposing an API for raw binary data access and tools for dealing more easily with binary data.

Discussion

For the purposes of our example, namely, file conversion, you can think of the process in terms of figure 3.3.

Binary data is read, processed, and written out in a more usable format using the binary specification as a guide and the binary API as the mechanism for accomplishing the transformation. This is not the only use of binary data. For example, you could do processing on a binary protocol to pass messages back and forth and the diagram would look different.

For our technique, the binary file format we'll work with is DBase 5.0 (.dbf). That format may sound obscure, but (to put it into context) it was a popular database format that's still heavily in use for attribution of geospatial data. You could think of it as a simplified Excel spreadsheet. The sample we'll work with is located at buffers/world.dbf.

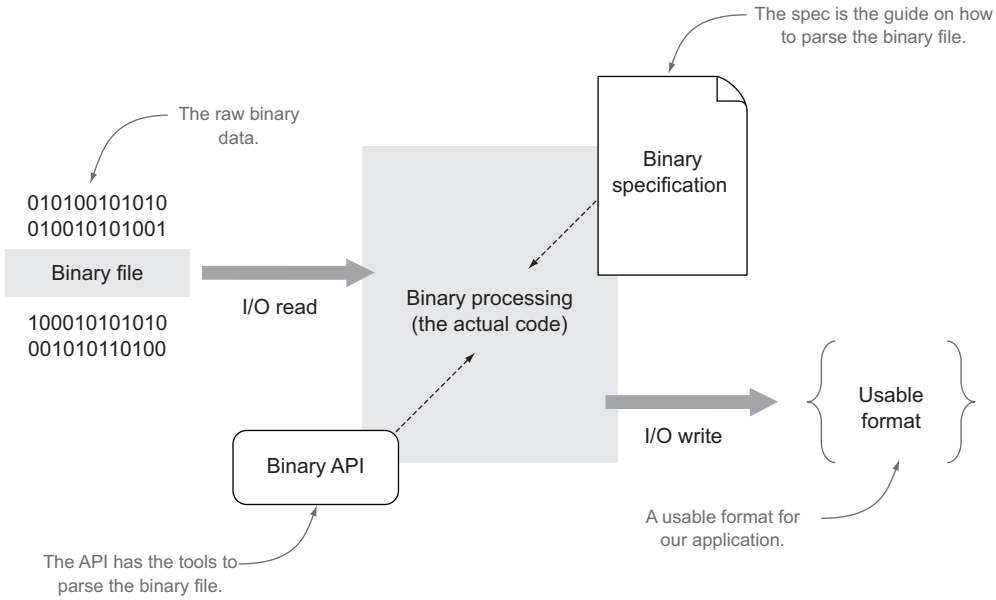


Figure 3.3 The transformation of binary data into a more usable/programmable format

The file contains geospatial information for the countries of the world. Unfortunately, if you were to open it in your text editor, it wouldn't be very useful.

> **WHY ARE WE COVERING IN DEPTH A BINARY FORMAT THAT I MAY NEVER USE?** Although we could've picked a number of binary formats, DBase 5.0 is one that will teach you a lot of different ways of approaching problems with reading binary files that are common to many other formats. In addition, binary formats are unfamiliar to many coming from a web development background, so we're taking some time to focus on reading binary specifications. Please feel free to skim if you're already familiar.

Since we want to use it in our Node application, JSON would be a good format choice because it can be natively parsed in JavaScript and resembles native JavaScript objects. This is illustrated in figure 3.4.

Figure 3.5 shows an example of the transformation we want to make: on the left is the raw binary opened in a text editor, and on the right is the converted JSON format.

The header

Before we can start tackling this problem, we'll need to do some research to find out the specification for the binary format we want to deal with. In our case, a number of similar specifications were found online from search engine queries. For DBase 5.0, the primary specification we'll use for this example is found at http://mng.bz/i7K4.

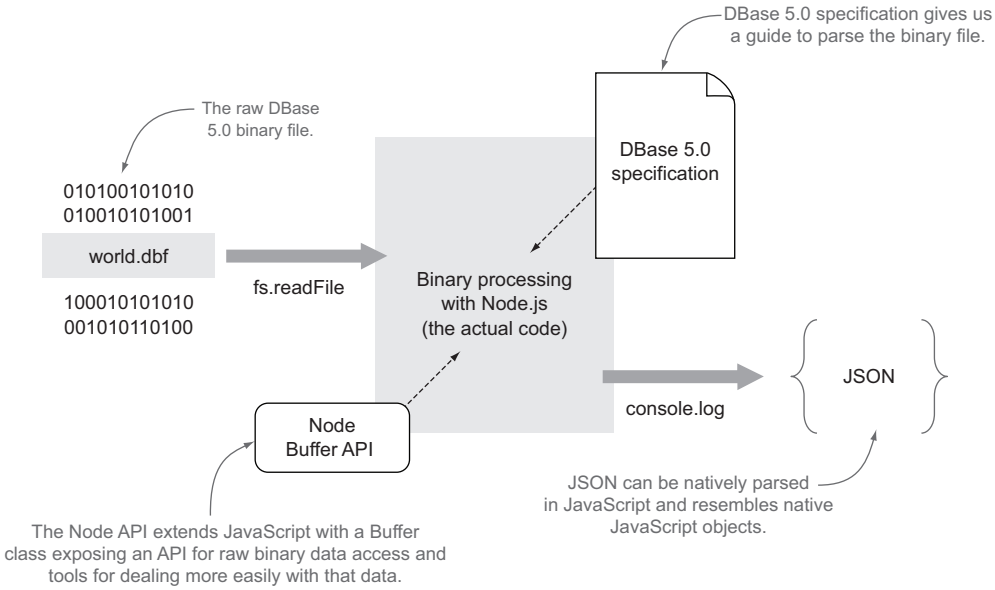


Figure 3.4 Binary data is read using `FileSystem` API into Node.js, transformed using the `Buffer` API into an easier-to-use JSON format.

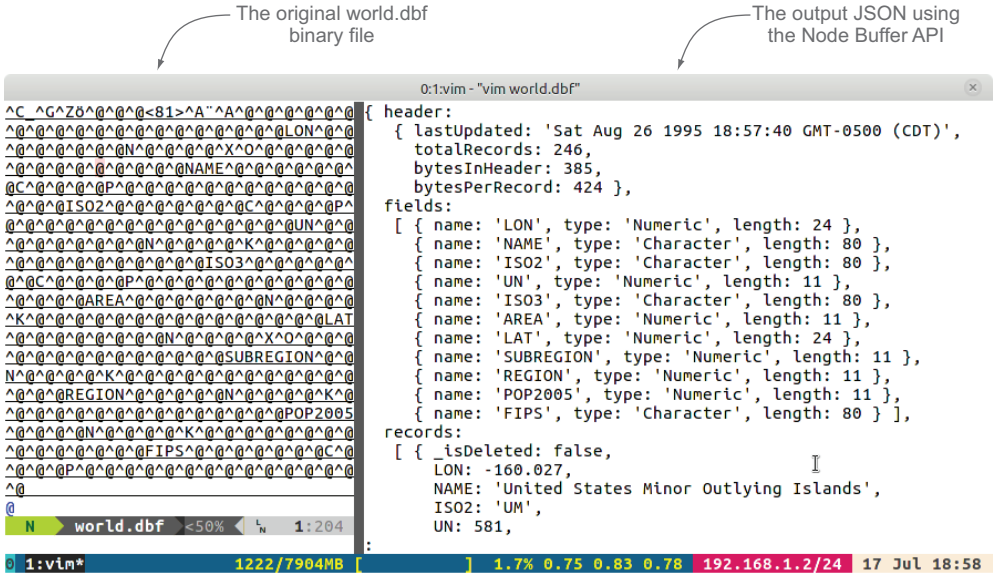


Figure 3.5 Final result of our transformation

The first portion of the specification is called the *header*. Many binary formats will use a header as a place to store metadata about the file; table 3.1 shows what the specification looks like for DBase 5.0.

Table 3.1 DBase 5.0 header specification

Byte	Contents	Description
0	1 byte	Valid dBASE for Windows table file; bits 0-2 indicate version number ...
1-3	3 bytes	Date of last update; in YYMMDD format
4-7	32-bit number	Number of records in the table
8-9	16-bit number	Number of bytes in the header
10-11	16-bit number	Number of bytes in the record
...	...	...
32-n each	32 bytes	Field descriptor array
n+1	1 byte	0Dh stored as the field terminator

Let's take a peek at the first row.

Byte	Contents	Description
0	1 byte	Valid dBASE for Windows table file; bits 0-2 indicate version number ...

This row tells us the byte located at position 0 contains the information specified in the description. So how do we access the byte at position 0? Thankfully, this is really simple with buffers.

In Node, *unless you specify a particular encoding* for the data you're reading in, you'll get back a Node `Buffer`, as seen in this example:

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

fs.readFile('./world.dbf', function (er, buf) {
  Buffer.isBuffer(buf); // true
});
```

`fs.readFile` isn't the only way to get back a buffer but, for the sake of simplicity, we'll use that method so we get the entire buffer back as an object after it's read. This method may not be ideal for large binary files where you wouldn't want to load the whole buffer into memory at once. In that case, you could stream the data with `fs.createReadStream` or manually read in parts of the file at a time with `fs.read`. It should also be noted that buffers aren't available only for files; they exist pretty much anywhere you can get streams of data (for example, post data on an HTTP request).

If you wanted to view a string representation of a buffer, a simple `buf.toString()` call would suffice (this defaults to UTF-8 encoding). This is nice if you're pulling in data that you know is just text:

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

fs.readFile('./world.dbf', function (er, buf) {
  console.log(buf.toString());
});
```

Returns a UTF-8 string by default

In our case, `buf.toString()` would be just as bad as opening up the world.dbf file in a text editor: unusable. We need to make sense of the binary data first.

> **NOTE** From here forward, whenever you see our variable `buf`, it refers to an instance of a `Buffer`, therefore part of the Node `Buffer` API.

In the table we talked about byte position 0. Buffers in Node act very similar to JavaScript arrays *but the indices are byte positions in memory*. So byte position 0 is `buf[0]`. In `Buffer` syntax, `buf[0]` is synonymous with the byte, the octet, the unsigned 8-bit integer, or positive signed 8-bit integer at position 0.

For this example, we don't really care about storing information about this particular byte. Let's move on to the next byte definition.

Byte	Contents	Description
1-3	3 bytes	Date of last update; in YYMMDD format

Here's something interesting: the date of the last update. But this spec doesn't tell us anything more than that it's 3 bytes and in YYMMDD format. All this is to say that you may not find all you're looking for in one spot. Subsequent web searches landed this information:

> *Each byte contains the number as a binary. YY is added to a base of 1900 decimal to determine the actual year. Therefore, YY has possible values from 0x00-0xFF, which allows for a range from 1900-2155.*[6]

That's more helpful. Let's look at parsing this in Node:

```
var header = {};

var date = new Date();
date.setUTCFullYear(1900 + buf[1]);
date.setUTCMonth(buf[2]);
date.setUTCDate(buf[3]);
header.lastUpdated = date.toUTCString();
```

Result: "Sat Aug 26 1995 ..."

Here we use a JavaScript `Date` object and set its year to 1900 plus the integer we pulled out of `buf[1]`. We use integers at positions 2 and 3 to set the month and date. Since JSON doesn't store JavaScript `Date` types, we'll store it as a UTC `Date` string.

[6] See http://www.dbase.com/Knowledgebase/INT/db7_file_fmt.htm.

Let's pause to recap. "Sat Aug 26 1995..." as shown here is the result of parsing a portion of world.dbf binary data into a JavaScript string. We'll see more examples of this as we continue.

Byte	Contents	Description
4-7	32-bit number	Number of records in the table

This next definition gives us two clues. We know the byte starts at offset 4 and it's a 32-bit number with the least significant byte first. Since we know the number shouldn't be negative, we can assume either a positive signed integer or an unsigned integer. Both are accessed the same way in the `Buffer` API:

```
header.totalRecords = buf.readUInt32LE(4);
```
Result: 246

`buf.readUInt32LE` will read an unsigned 32-bit integer with little-endian format from the offset of 4, which matches our description from earlier.

The next two definitions follow a similar pattern except they're 16-bit integers. Following are their definitions.

Byte	Contents	Description
8-9	16-bit number	Number of bytes in the header
10-11	16-bit number	Number of bytes in the record

And here's the corresponding code:

```
header.bytesInHeader = buf.readUInt16LE(8);
header.bytesPerRecord = buf.readUInt16LE(10);
```
Result: 385

Result: 424

The transformation that has taken place between the specification and the code for this header section is illustrated in figure 3.6.

The field descriptor array

Only one more relevant piece of information for this example remains in the header of the world.dbf file. It's the definitions for the fields, including type and name information, seen in the following lines.

Byte	Contents	Description
32-n each	32 bytes	Field descriptor array
n+1	1 byte	0Dh stored as the field terminator

From this we know that each field description is stored as 32 bytes of information. Since this database could have one or more fields for data, we'll know it's finished

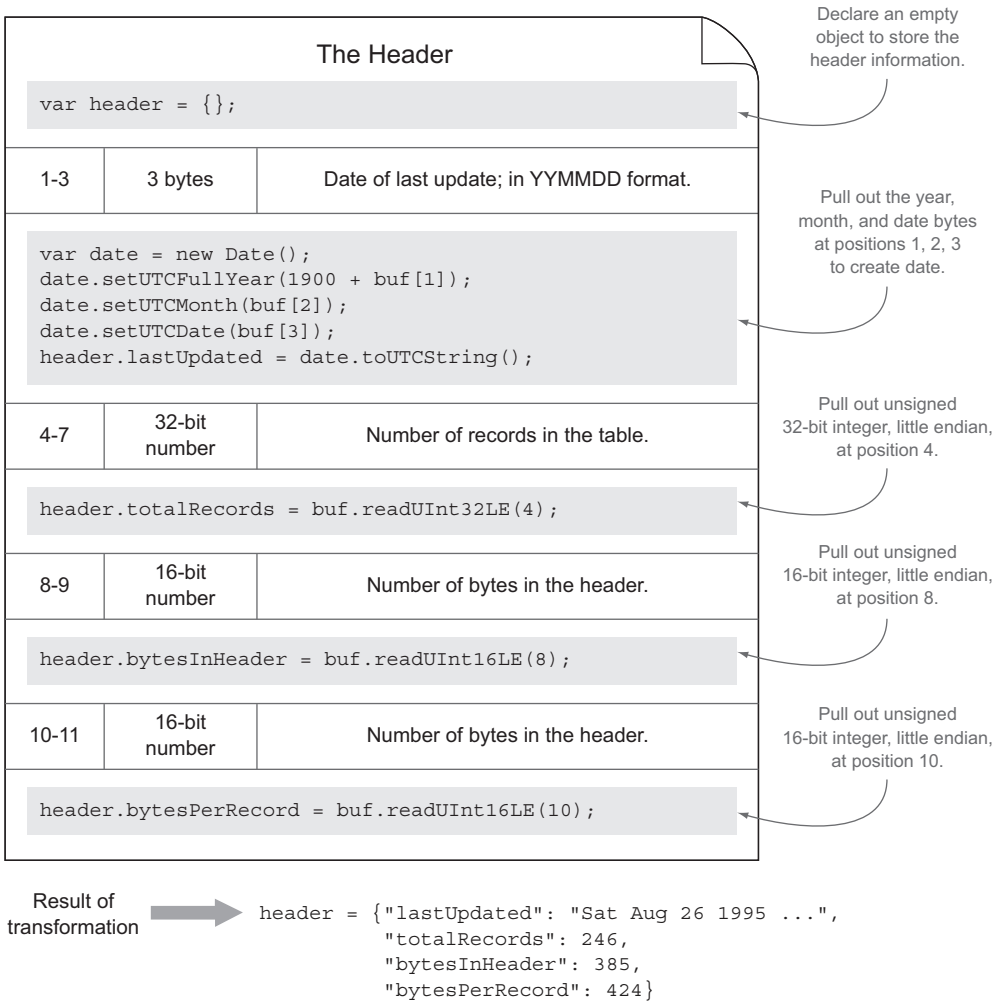


Figure 3.6 The header: transformation from the specification to code using the Node `Buffer` API

when we hit the 1 byte field terminator (0Dh) shown in the second row. Let's write a structure to handle this:

```
var fields = [];
var fieldOffset = 32;
var fieldTerminator = 0x0D;

while (buf[fieldOffset] != fieldTerminator) {
  // here is where we parse each field
  fieldOffset += 32;
}
```

JavaScript hex literal notation to represent 0Dh

Here we loop through the buffer 32 bytes at a time until we hit the `fieldTerminator`, which is represented in hexadecimal notation.

Now we need to handle the information concerning each field descriptor. The specification has another table specifically for this; the relevant information for our example is shown in table 3.2.

Table 3.2 DBase 5.0 field descriptor array specification

Byte	Contents	Description
0-10	11 bytes	Field name in ASCII (zero-filled)
11	1 byte	Field type in ASCII (C, N, ...)
...	...	...
16	1 byte	Field length in binary

Note that the indexing for the bytes starts over at 0, even though we're well past byte position 0 in our reading of the file. It would be nice to start over at each record so we could follow the specification more closely. `Buffer` provides a `slice` method for us to do just that:

```
var fields = [];
var fieldOffset = 32;
var fieldTerminator = 0x0D;

while (buf[fieldOffset] != fieldTerminator) {
  var fieldBuf = buf.slice(fieldOffset, fieldOffset+32);
  // here is where we parse each field
  fieldOffset += 32;
}
```

`buf.slice(start,end)` is very similar to a standard array slice method in that it returns a buffer indexed at `start` to `end`. But it differs in that it doesn't return a new copy of the data. It returns just a snapshot of the data at those points. So if you manipulate the data in the sliced buffer in any way, *it will also be manipulated in the original buffer.*

With our new `fieldBuf` indexed at zero for each iteration, we can approach the specification without doing extra math in our heads. Let's look at the first line.

Byte	Contents	Description
0-10	11 bytes	Field name in ASCII (zero-filled)

Here's the code to extract the field name:

```
var field = {};

field.name = fieldBuf.toString('ascii', 0, 11).replace(/\u0000/g,'');
```

Result, such as "LON" (longitude)

By default, `buf.toString()` assumes `utf8`, but Node `Buffers` support other encodings as well,[7] including `ascii`, which is what our spec calls for. `buf.toString()` also

[7] See http://nodejs.org/api/buffer.html#buffer_buffer.

allows you to pass in the range that you want converted. We also have to `replace()` the zero-filled characters with empty strings if the field was shorter than 11 bytes so we don't end up with zero-filled characters (\u0000) in our names.

The next relevant field is a field data type.

Byte	Contents	Description
11	1 byte	Field type in ASCII (C, N, ...)

But the characters C and N don't really mean anything to us yet. Further down the specification, we get definitions for these types, as shown in table 3.3.

Table 3.3 Field types specification

Data type	Data input
C (Character)	All OEM code page characters
N (Numeric)	- . 0 1 2 3 4 5 6 7 8 9

It would be nice to convert this data to relevant types for our application. JavaScript doesn't use the language *character* or *numeric*, but it does have *String* and *Number*; let's keep that in mind when we parse the actual records. For now we can store this in a little lookup object to do the conversion later:

```
var FIELD_TYPES = {
  C: 'Character',
  N: 'Numeric'
}
```

Now that we have a lookup table, we can pull out the relevant information as we continue converting the binary data:

```
field.type = FIELD_TYPES[fieldBuf.toString('ascii', 11, 12)];
```

Result: will be "Character or "Numeric"

`buf.toString()` will give us our one ASCII character that we then look up in the hash to get the full type name.

There's only one other bit of information we need to parse the remaining file from each field description—the field size.

Byte	Contents	Description
16	1 byte	Field length in binary

We write this now-familiar code:

```
field.length = fieldBuf[16];
```

Result, such as 435

The transformation that has taken place between the specification and the code for this field descriptor array section is illustrated in figure 3.7.

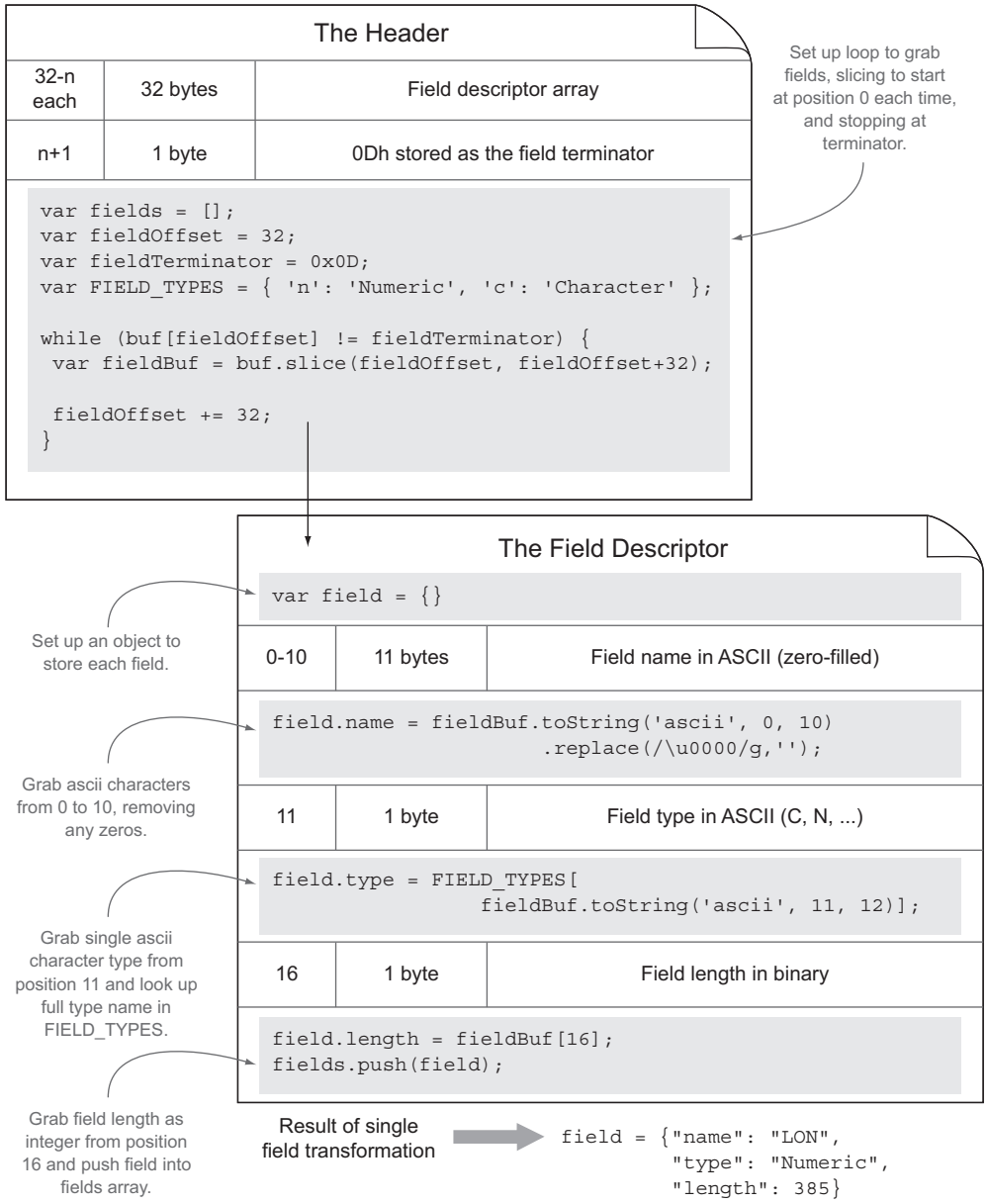


Figure 3.7 The field descriptor array: transformation from the specification to code using the Node `Buffer` API

The records

Now that we've parsed the header, including the field descriptors, we have one more part to process: the actual record data. The specification tells us this:

> *The records follow the header in the table file. Data records are preceded by one byte, that is, a space (20h) if the record is not deleted, an asterisk (2Ah) if the record is deleted. Fields are packed into records without field separators or record terminators. The end of the file is marked by a single byte, with the end-of-file marker, an OEM code page character value of 26 (1Ah).*

Let's break this down for discussion:

> *The records follow the header in the table file.*

Although we could've kept track of the byte position after the `fieldOffset`, the header had a field for number of bytes in the header, which we stored as `header.bytesInHeader`. So we know we need to start there:

```
var startingRecordOffset = header.bytesInHeader;
```

We also learned a couple other things from our parsing of the header. The first is how many records exist in the data, which we stored as `header.totalRecords`. The second is how many bytes are allocated for each record, which was stored as `header.bytesPerRecord`. Knowing where to start, how many to iterate, and how much of a jump per iteration helps us set up a nice `for` loop for handling each record:

```
for (var i = 0; i < header.totalRecords; i++) {
  var recordOffset = startingRecordOffset +
                       (i * header.bytesPerRecord);
  // here is where we parse each record
}
```

Now, at the beginning of each iteration, we know the byte position we want to start at is stored as `recordOffset`. Let's continue reading the specification:

> *Data records are preceded by one byte, that is, a space (20h) if the record is not deleted, an asterisk (2Ah) if the record is deleted.*

Next we have to check that first byte to see if the record was deleted:

```
var record = {};
record._isDel = buf.readUInt8(recordOffset) == 0x2A;
recordOffset++;
```

Note: We could've also used buf[recordOffset]

Similar to when we tested for the `fieldTerminator` in our header file, here we test to see if the integer matches `0x2A` or the ASCII "asterisk" character. Let's continue reading:

> *Fields are packed into records without field separators or record terminators.*

Lastly, we can pull in the actual record data. This pulls in the information we learned from parsing the field descriptor array. We stored a `field.type`, `field.name`, and `field.length` (in bytes) for each field. We want to store the name as a key in the record where the value is the data for that length of bytes converted to the correct type. Let's look at it in simple pseudo code:

```
record[name] = cast type for (characters from length)
e.g.
record['pop2005'] = Number("13119679")
```

We also want to do this type conversion for every field per record, so we use another `for` loop:

```
for (var j = 0; j < fields.length; j++) {
  var field = fields[j];
  var Type = field.type == 'Numeric' ? Number : String;
  record[field.name] = Type(buf.toString('ascii', recordOffset,
                            recordOffset+field.length).trim());
  recordOffset += field.length;
}
```

We loop through each of the fields:

1. First, we find out which JavaScript type we want to cast the value to and store it in a variable `Type`.
2. Then, we use `buf.toString` to pull out the characters from `recordOffset` to the next `field.length`. We also have to `trim()` the data because we don't know if all the bytes were used to store relevant data or just filled with spaces.
3. Lastly, we increment the `recordOffset` with the `field.length` so that we keep the location to start at for the next field when we go around the `for` loop again.

The transformation that has taken place between the specification and the code for this records section is illustrated in figure 3.8.

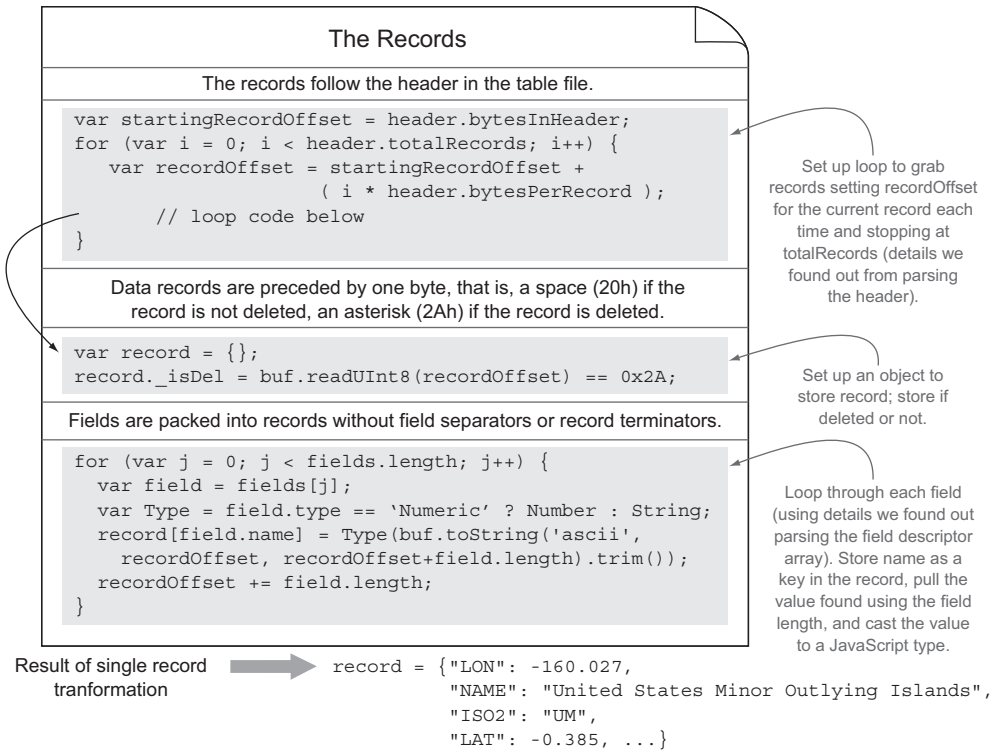
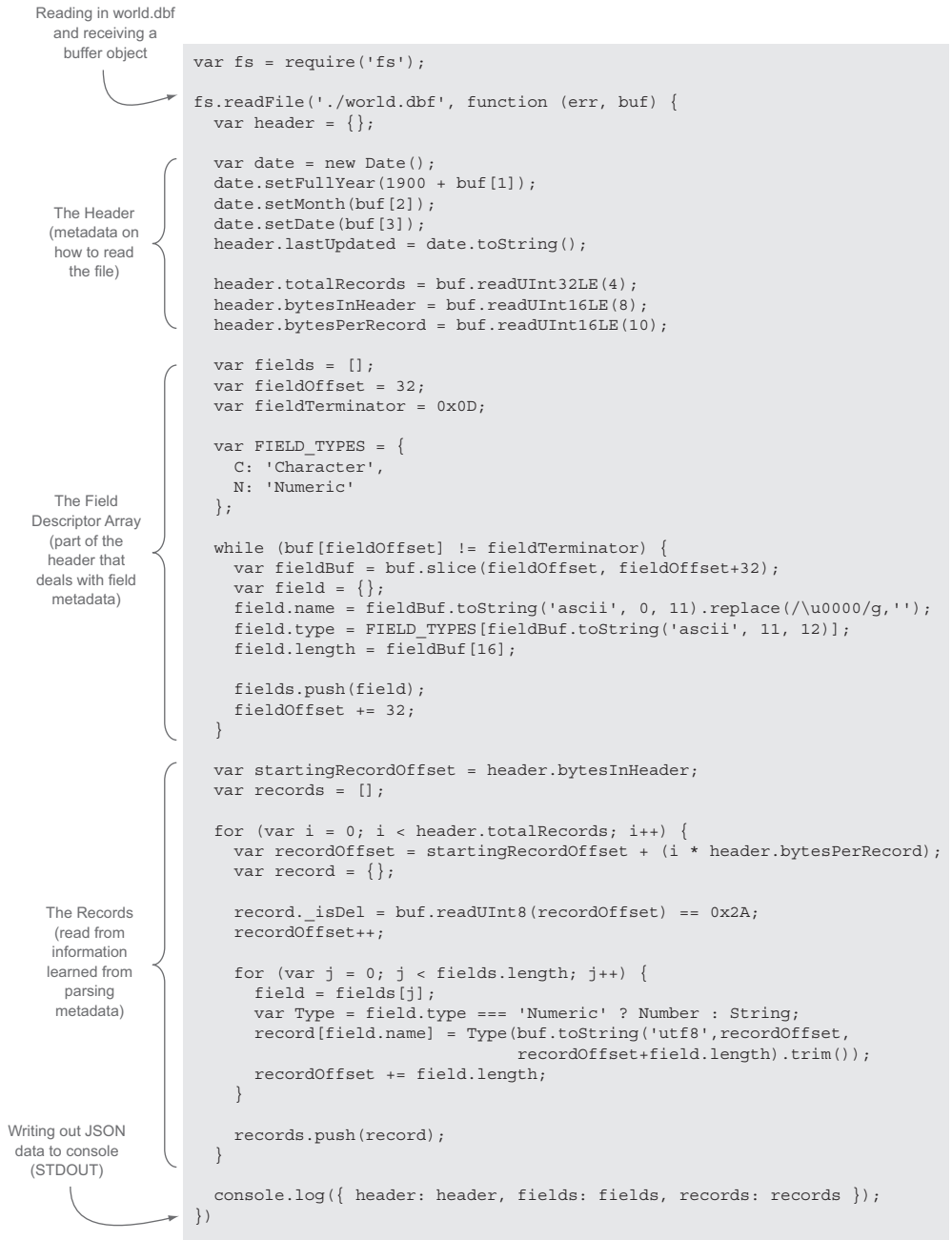


Figure 3.8 The records: transformation from the specification to code using the Node `Buffer` API

Still with me? I hope so. The complete code sample is shown in figure 3.9.




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

fs.readFile('./world.dbf', function (err, buf) {
  var header = {};

  var date = new Date();
  date.setFullYear(1900 + buf[1]);
  date.setMonth(buf[2]);
  date.setDate(buf[3]);
  header.lastUpdated = date.toString();

  header.totalRecords = buf.readUInt32LE(4);
  header.bytesInHeader = buf.readUInt16LE(8);
  header.bytesPerRecord = buf.readUInt16LE(10);

  var fields = [];
  var fieldOffset = 32;
  var fieldTerminator = 0x0D;

  var FIELD_TYPES = {
    C: 'Character',
    N: 'Numeric'
  };

  while (buf[fieldOffset] != fieldTerminator) {
    var fieldBuf = buf.slice(fieldOffset, fieldOffset+32);
    var field = {};
    field.name = fieldBuf.toString('ascii', 0, 11).replace(/\u0000/g,'');
    field.type = FIELD_TYPES[fieldBuf.toString('ascii', 11, 12)];
    field.length = fieldBuf[16];

    fields.push(field);
    fieldOffset += 32;
  }

  var startingRecordOffset = header.bytesInHeader;
  var records = [];

  for (var i = 0; i < header.totalRecords; i++) {
    var recordOffset = startingRecordOffset + (i * header.bytesPerRecord);
    var record = {};

    record._isDel = buf.readUInt8(recordOffset) == 0x2A;
    recordOffset++;

    for (var j = 0; j < fields.length; j++) {
      field = fields[j];
      var Type = field.type === 'Numeric' ? Number : String;
      record[field.name] = Type(buf.toString('utf8',recordOffset,
                                recordOffset+field.length).trim());
      recordOffset += field.length;
    }

    records.push(record);
  }

  console.log({ header: header, fields: fields, records: records });
})
```

Figure 3.9 The full set of code for parsing a DBF file into JSON

Using the Node `Buffer` API, we were able to turn a binary file into a usable JSON format. The output of running this application is shown next:

```
{ header:
   { lastUpdated: 'Sat Aug 26 1995 21:55:03 GMT-0500 (CDT)',
     totalRecords: 246,
     bytesInHeader: 385,
     bytesPerRecord: 424 },
  fields:
   [ { name: 'LON', type: 'Numeric', length: 24 },
     { name: 'NAME', type: 'Character', length: 80 },
     { name: 'ISO2', type: 'Character', length: 80 },
     { name: 'UN', type: 'Numeric', length: 11 },
     { name: 'ISO3', type: 'Character', length: 80 },
     { name: 'AREA', type: 'Numeric', length: 11 },
     { name: 'LAT', type: 'Numeric', length: 24 },
     { name: 'SUBREGION', type: 'Numeric', length: 11 },
     { name: 'REGION', type: 'Numeric', length: 11 },
     { name: 'POP2005', type: 'Numeric', length: 11 },
     { name: 'FIPS', type: 'Character', length: 80 } ],
  records:
   [ { _isDel: false,
       LON: -160.027,
       NAME: 'United States Minor Outlying Islands',
       ISO2: 'UM',
       UN: 581,
       ISO3: 'UMI',
       AREA: 0,
       LAT: -0.385,
       SUBREGION: 0,
       REGION: 0,
       POP2005: 0,
       FIPS: '' },
     { _isDel: false,
       LON: 35.278,
       NAME: 'Palestine',
       ISO2: 'PS',
       UN: 275,
       ISO3: 'PSE',
       AREA: 0,
       LAT: 32.037,
       SUBREGION: 145,
       REGION: 142,
       POP2005: 3762005,
       FIPS: '' },
       ...
 }
```

And almost magically a binary file that wasn't human-readable is turned into, not only a readable format, but also a usable data format to work with and do more transformations with. Of course, it isn't magic, but rather investing the time to learn a binary format and using the tools available to do a conversion. The `Buffer` API provides good tools to do this.

Using fs methods

We could've also chosen to write the resulting code out to a file using `fs.writeFile` and friends.[a] Just like most APIs in Node can read in a buffer object, most also can write out a buffer object. In our case we didn't end up with a buffer but rather a JSON object, so we could've used `JSON.stringify` in conjunction with `fs.writeFile` to write that data out:

```
fs.writeFile('world.json', JSON.stringify(result), ...
```

[a] See http://nodejs.org/api/fs.html.

Binary file formats can be a lot of fun to crack. Another fun but practical use for `Buffers` is working binary protocols, which we'll tackle next.

3.3 Creating your own binary protocol

It feels like you've cracked a code when you read a binary file and make sense out of it. It can be just as fun to write your own puzzles and decode them. Of course, this isn't just for fun. Using a well-defined binary protocol can be a compact and efficient way to transfer data.

TECHNIQUE 18 Creating your own network protocol

In this technique we'll cover some additional aspects of working with binary data, like bit masks and protocol design. We'll also look into compressing binary data.

Problem

You want create an efficient transport of messages across the network or in process.

Solution

JavaScript and the Node `Buffer` API give you tools to create your own binary protocol.

Discussion

To create a binary protocol, you first have to define what kind of information you want to send across the wire and how you'll represent that information. Like you learned in the last technique, a specification provides a good roadmap for this.

For this technique, we'll develop a simple and compact database protocol. Our protocol will involve

- Using a bitmask to determine which database(s) to store the message in
- Writing data to a particular key that will be an unsigned integer between 0-255 (one byte)
- Storing a message that is compressed data of any length using zlib

Table 3.4 shows how we could write the specification.

Table 3.4 Simple key-value database protocol

Byte	Contents	Description
0	1 byte	Determines which database(s) to write the data to based on which bits are toggled on. Each bit position represents a database from 1–8.
1	1 byte	An unsigned integer of one byte (0–255) used as the database key to store the data in.
2-n	0-n bytes	The data to store, which can be any amount of bytes that have been compressed using `deflate` (zlib).

Playing with bits to select databases

Our protocol states that the first byte will be used to represent which databases should record the information transferred. On the receiving end, our main database will be a simple multidimensional array that will hold spots for eight databases (since there are eight bits in a byte). This can be simply represented using array literals in JavaScript:

```
var database = [ [], [], [], [], [], [], [], [] ];
```

Whatever bits are turned on will indicate which database or databases will store the message received. For example, the number `8` is represented as `00001000` in binary. In this case we'd store the information in database 4, since the fourth bit is on (bits are read from right to left).

> **ZERO-INDEXED ARRAYS** Arrays are zero-indexed in JavaScript, so database 4 is in array position 3, but to avoid complicating things, we're intentionally calling our databases 1 through 8 instead of 0 through 7 to match our language more closely when talking about bits in a byte.

If you're ever curious about a number's binary representation in JavaScript, you can use the built-in `toString` method, giving it a base `2` as the first argument:

```
8..toString(2) // '1000'
```

Two dots (..) are needed to call a method on a number, since the first is parsed as a decimal point.

Numbers can have more than one bit turned on as well; for example, 20 is `00010100` in binary, and for our application that would mean we wanted to store the message in databases 3 and 5.

So how do we test to see which bits are turned on for any given number? To solve this, we can use a *bitmask*. A bitmask represents the bit pattern we're interested in testing. For example, if we were interested in finding out whether we should store some data in database 5, we could create a bitmask that has the fifth bit turned on. In binary, this would look like `00010000`, which is the number 32 (or `0x20` in hex notation).

We then have to test our bitmask against a value, and JavaScript includes various bitwise operators[8] to do this. One is the `&` (bitwise AND) operator. The `&` operator

[8] See https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Operators/Bitwise_Operators.

behaves similarly to the `&&` operator, but instead of testing for two conditions to be true, it tests for two bits to be on (have ones and not zeros) and keeps the bits on (or one) where that's the case:

```
  000101000
& 000100000
-----------
  000100000
```

Bit position 5 was on for both values, so it remains when using `&`. Armed with this knowledge, we can see that a value compared with the bitmask *will be the bitmask* if it has the same bit or bits turned on. With this information, we can set up a simple conditional to test:

```
if ( (value & bitmask) === bitmask) { .. }
```

It's important that the `&` expression be surrounded by parentheses; otherwise, the equality of the bitmasks would be checked first because of operator precedence.[9]

To test the first byte received in our binary protocol, we'll want to set up a list of bitmasks that correspond with the indexes of our databases. If the bitmask matches, we know the database at that index will need the data written to it. The "on" bits for every position are an array

```
var bitmasks = [ 1, 2, 4, 8, 16, 32, 64, 128 ]
```

which corresponds to this:

```
1        2        4        8        16       32       64       128
-------------------------------------------------------------------------
00000001 00000010 00000100 00001000 00010000 00100000 01000000 10000000
```

Now we know that if a byte matches `1` in our bitmasks array, it will match database 1 or array position 0. We can set up a simple loop to test each bitmask against the value of the first byte:

```
var database = [ [], [], [], [], [], [], [], [] ];
var bitmasks = [ 1, 2, 4, 8, 16, 32, 64, 128 ];

function store (buf) {
  var db = buf[0];                          <-- Grabbing the byte from position 0
   bitmasks.forEach(function (bitmask, index) {
    if ( (db & bitmask) === bitmask) {
      // found a match for database[index]
    }
  });
}
```

Working with bits can be tricky at first, but once you understand more about how they work, they become more manageable. So far all of what we've covered is available not only in Node, but in browser JavaScript too. We've made it far enough to determine

[9] See https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Operators/Operator_Precedence.

which database we should put our incoming data in; we still have to find out which key to store the data in.

Looking up the key to store the data

This is the easiest part of our example, because you've already learned this from the previous technique. Before we begin, let's look at the relevant part of the specification defined earlier in table 3.4.

Byte	Contents	Description
1	1 byte	An unsigned integer of one byte (0–255) used as the database key to store the data in.

We know we'll be receiving at byte position 1 an unsigned integer of one byte (0-255) that will be used as a database key to store the data in. We purposely set up the database to be a multidimensional array where the first dimension is the databases. Now we can use the second dimension as a place to store the keys and values, and since the keys are numbers, an array will work.[10] Let's illustrate to make this more concrete. Here's what storing the value `'foo'` inside the first and third databases at key `0` would look like:

```
[
  ['foo'],
  [],
  ['foo'],
  [],
  [],
  [],
  [],
  []
]
```

To get the key value out of position 1, we can use the hopefully now familiar `readUInt8` method:

```
var key = buf.readUInt8(1);
```

Note that buf[1] does the same thing

Let's add that to our previous main code sample we're building:

```
var database = [ [], [], [], [], [], [], [], [] ];
var bitmasks = [ 1, 2, 4, 8, 16, 32, 64, 128 ];

function store (buf) {
  var db = buf[0];
  var key = buf.readUInt8(1);

  bitmasks.forEach(function (bitmask, index) {
    if ( (db & bitmask) === bitmask) {
```

[10] Although there are more-ideal alternatives coming in ECMAScript 6.

```
      database[index][key] = 'some data';
    }
  });
}
```

'some data' is a placeholder for now.

Now that we're able to parse the database(s) and the keys within those database(s), we can get to parsing the actual data to store.

Inflating data with zlib

It's a smart idea to compress string/ASCII/UTF-8 data when sending it across the wire, as compression can really cut down on bandwidth usage. In our simple database protocol, we assume that the data we get to store has been compressed; let's look at the specification to see the relevant description.

Node includes a built-in `zlib` module that exposes `deflate` (compress) and `inflate` (uncompress) methods. It also includes gzip compression. To avoid getting malformed messages, we can check that the received message was indeed properly compressed, and if not, we'll refuse to inflate it. Typically, the first byte of zlib "deflated" data is `0x78`,[11] so we can test for that accordingly:

```
if (buf[2] === 0x78) { .. }
```

Remember, we start at byte position 2 because the previous were the key (1) and the database byte (0) we covered earlier.

Now that we know that we're most likely dealing with deflated data, we can inflate it using `zlib.inflate`. We'll also need to use `buf.slice()` to get just the data portion of our message (since leaving the first two bytes would cause an error):

zlib.inflate returns a buffer, so we convert it into a UTF-8 string to store.

```
var zlib = require('zlib');
...
if (buf[2] === 0x78) {
  zlib.inflate(buf.slice(2), function (er, inflatedBuf) {
    if (er) return console.error(er);

    var data = inflatedBuf.toString();
  })
}
```

Even though we checked, something else could have failed; if so, we log it out and don't continue.

We have everything we need to store some data in our database using our simple database protocol. Let's put all the components together:

```
var zlib = require('zlib');
var database = [ [], [], [], [], [], [], [], [] ];
var bitmasks = [ 1, 2, 4, 8, 16, 32, 64, 128 ];

function store (buf) {
  var db = buf[0];
  var key = buf.readUInt8(1);

  if (buf[2] === 0x78) {
    zlib.inflate(buf.slice(2), function (er, inflatedBuf) {
      if (er) return console.error(er);
```

[11] A more robust implementation should do more checks; see http://tools.ietf.org/html/rfc6713.

```
      var data = inflatedBuf.toString();

      bitmasks.forEach(function (bitmask, index) {
        if ( (db & bitmask) === bitmask) {
          database[index][key] = data;
        }
      });
    });
  }
}
```

The actual data is stored in the key of every database that matched.

Now we have the code in place to store some data. We could generate a message by using the following:

```
var zlib = require('zlib');
var header = new Buffer(2);

header[0] = 8;
header[1] = 0;

zlib.deflate('my message', function (er, deflateBuf) {
  if (er) return console.error(er);
  var message = Buffer.concat([header, deflateBuf]);
  store(message);
})
```

Store in key 0

Store in database 4 (8 = 00001000)

Concat header and data into one message

Deflate the data 'my message'

Store message

We could write an example that sends messages over TCP and do more error handling. But let's leave that as an exercise for you to tackle as you learn about networking in Node in a later chapter.

3.4 Summary

In this chapter you learned about buffers and how to turn buffers into different encoded strings using the `toString` method. We dove into the complicated task of turning a binary file into something more usable using the `Buffer` API. Lastly, we had some fun creating our own protocol and learning about bitmasks and compression.

We covered some common uses of buffers in Node, varying in difficulty to hopefully make you more comfortable using them and making the most of them. Go forth and tackle a binary format conversion and publish your work on NPM, or maybe a protocol that better fits your business needs is waiting to be written.

In the next chapter we'll look at another core part of Node—events.

4 Events: Mastering EventEmitter and beyond

This chapter covers

- Using Node's EventEmitter module
- Managing errors
- How third-party modules use EventEmitter
- How to use domains with events
- Alternatives to EventEmitter

Node's `events` module currently includes just a single class: `EventEmitter`. This class is used throughout both Node's built-in modules and third-party modules. It contributes to the overall architecture of many Node programs. Therefore it's important to understand `EventEmitter` and how to use it.

It's a simple class, and if you're familiar with DOM or jQuery events, then you shouldn't have much trouble understanding it. The major consideration when using Node is in error handling, and we'll look at this in technique 21.

`EventEmitter` can be used in various ways—it's generally used as a base class for solving a wide range of problems, from building network servers to architecting

application logic. In view of the fact that it's used as the basis for key classes in popular Node modules like Express, learning how it works can be useful for writing idiomatic code that plays well alongside existing modules.

In this chapter you'll learn how to use `EventEmitter` to make custom classes, and how it's used within Node and open source modules. You'll also learn how to solve problems found when using `EventEmitter`, and see some alternatives to it.

4.1 Basic usage

To use `EventEmitter`, the base class must be inherited from. This section includes techniques for inheriting from `EventEmitter` and mixing it into other classes that already inherit from another base class.

TECHNIQUE 19 Inheriting from EventEmitter

This technique demonstrates how to create custom classes based on `EventEmitter`. By understanding the principles in this technique, you'll learn how to use `EventEmitter`, and how to better use modules that are built with it.

Problem

You want to use an event-based approach to solve a problem. You have a class that you'd like to operate when asynchronous events occur.

Web, desktop, and mobile user interfaces have one thing in common: they're event-based. Events are a great paradigm for dealing with something inherently asynchronous: the input from human beings. To show how `EventEmitter` works, we'll use a music player as an example. It won't really play music, but the underlying concept is a great way to learn how to use events.

Solution

The canonical example of using events in Node is inheriting from `EventEmitter`. This can be done by using a simple prototype class—just remember to call `EventEmitter`'s constructor from within your new constructor.

The first listing shows how to inherit from `EventEmitter`.

Listing 4.1 Inheriting from `EventEmitter`

```
var util = require('util');
var events = require('events');

function MusicPlayer() {
  events.EventEmitter.call(this);
}

util.inherits(MusicPlayer, events.EventEmitter);
```

Using util.inherits is the idiomatic Node way to inherit from prototype classes.

Discussion

The combination of a simple constructor function and `util.inherits` is the easiest and most common way to create customized event-based classes. The next listing extends the previous listing to show how to `emit` and bind listeners using `on`.

Listing 4.2 Inheriting from `EventEmitter`

```
var util = require('util');
var events = require('events');
var AudioDevice = {
  play: function(track) {
    // Stub: Trigger playback through iTunes, mpg123, etc.
  },

  stop: function() {
  }
};

function MusicPlayer() {
  this.playing = false;
  events.EventEmitter.call(this);
}

util.inherits(MusicPlayer, events.EventEmitter);

var musicPlayer = new MusicPlayer();

musicPlayer.on('play', function(track) {
  this.playing = true;
  AudioDevice.play(track);
});

musicPlayer.on('stop', function() {
  this.playing = false;
  AudioDevice.stop();
});

musicPlayer.emit('play', 'The Roots - The Fire');

setTimeout(function() {
  musicPlayer.emit('stop');
}, 1000);
```

The class's state can be configured, and then EventEmitter's constructor can be called as required.

The inherits method copies the methods from one prototype into another—this is the general pattern for creating classes based on EventEmitter.

The emit method is used to trigger events.

This might not seem like much, but suppose we need to do something else when `play` is triggered—perhaps the user interface needs to be updated. This can be supported simply by adding another listener to the `play` event. The following listing shows how to add more listeners.

Listing 4.3 Adding multiple listeners

```
var util = require('util');
var events = require('events');

function MusicPlayer() {
  this.playing = false;
  events.EventEmitter.call(this);
}

util.inherits(MusicPlayer, events.EventEmitter);
```

```
var musicPlayer = new MusicPlayer();

musicPlayer.on('play', function(track) {
  this.playing = true;
});

musicPlayer.on('stop', function() {
  this.playing = false;
});

musicPlayer.on('play', function(track) {
  console.log('Track now playing:', track);
});

musicPlayer.emit('play', 'The Roots - The Fire');

setTimeout(function() {
  musicPlayer.emit('stop');
}, 1000);
```

New listeners can be added as needed.

Listeners can be removed as well. `emitter.removeListener` removes a listener for a specific event, whereas `emitter.removeAllListeners` removes all of them. You'll need to store the listener in a variable to be able to reference it when removing a specific listener, which is similar to removing timers with `clearTimeout`. The next listing shows this in action.

Listing 4.4 Removing listeners

```
function play(track) {
  this.playing = true;
}

musicPlayer.on('play', play);

musicPlayer.removeListener('play', play);
```

A reference to the listener is required to be able to remove it.

`util.inherits` works by wrapping around the ES5 method `Object.create`, which inherits the properties from one prototype into another. Node's implementation also sets the superconstructor in the `super_` property. This makes accessing the original constructor a lot easier—after using `util.inherits`, your prototype class will have access to `EventEmitter` through `YourClass.super_`.

You can also respond to an event once, rather than every time it fires. To do that, attach a listener with the `once` method. This is useful where an event can be emitted multiple times, but you only care about it happening a single time. For example, you could update listing 4.3 to track if the play event has ever been triggered:

```
musicPlayer.once('play', {
  this.audioFirstStarted = new Date();
});
```

When inheriting from `EventEmitter`, it's a good idea to use `events.EventEmitter.call(this)` in your constructor to run `EventEmitter`'s constructor. The reason for this is because it'll attach the instance to a `domain` if domains are being used. To learn more about domains, see technique 22.

The methods we've covered here—`on`, `emit`, and `removeListener`—are fundamental to Node development. Once you've mastered `EventEmitter`, you'll find it cropping up everywhere: in Node's built-in modules and beyond. Creating TCP/IP servers with `net.createServer` will return a server based on `EventEmitter`, and even the `process` global object is an instance of `EventEmitter`. In addition, popular modules like `Express` are based around `EventEmitter`—you can actually create an Express `app` object and call `app.emit` to send messages around an Express project.

TECHNIQUE 20 Mixing in EventEmitter

Sometimes inheritance isn't the right way to use `EventEmitter`. In these cases, mixing in `EventEmitter` may work.

Problem

This is an alternative option to technique 19. Rather than using `EventEmitter` as a base class, it's possible to copy its methods into another class. This is useful when you have an existing class and can't easily rework it to inherit directly from `EventEmitter`.

Solution

Using a for-in loop is sufficient for copying the properties from one `prototype` to another. In this way you can copy the necessary properties from `EventEmitter`.

Discussion

This example might seem a little contrived, but sometimes it really is useful to copy `EventEmitter`'s properties rather than inherit from it in the usual way. This approach is more akin to a mixin, or multiple inheritance; see this demonstrated in the following listing.

Listing 4.5 Mixing in `EventEmitter`

```
var EventEmitter = require('events').EventEmitter;

function MusicPlayer(track) {
  this.track = track;
  this.playing = false;

  for (var methodName in EventEmitter.prototype) {
    this[methodName] = EventEmitter.prototype[methodName];
  }
}

MusicPlayer.prototype = {
  toString: function() {
    if (this.playing) {
      return 'Now playing: ' + this.track;
    } else {
      return 'Stopped';
```

This is the for-in loop that copies the relevant properties.

```
    }
  }
};

var musicPlayer = new MusicPlayer('Girl Talk - Still Here');

musicPlayer.on('play', function() {
  this.playing = true;
  console.log(this.toString());
});

musicPlayer.emit('play');
```

One example of multiple inheritance in the wild is the Connect framework.[1] The core `Server` class inherits from multiple sources, and in this case the Connect authors have decided to make their own property copying method, shown in the next listing.

Listing 4.6 `utils.merge` from Connect

```
exports.merge = function(a, b){
  if (a && b) {
    for (var key in b) {
      a[key] = b[key];
    }
  }
  return a;
};
```

This technique may be useful when you already have a well-established class that could benefit from events, but can't easily be a direct descendant of `EventEmitter`.

Once you've inherited from `EventEmitter` you'll need to handle errors. The next section explores techniques for handling errors generated by `EventEmitter` classes.

4.2 Error handling

Although most events are treated equally, `error` events are a special case and are therefore treated differently. This section looks at two ways of handling errors: one attaches a listener to the `error` event, and the other uses domains to collect errors from groups of `EventEmitter` instances.

TECHNIQUE 21 Managing errors

Error handling with `EventEmitter` has its own special rules that must be adhered to. This technique explains how error handling works.

Problem

You're using an `EventEmitter` and want to gracefully handle when errors occur, but it keeps raising exceptions.

[1] See http://www.senchalabs.org/connect/.

Solution

To prevent `EventEmitter` from throwing exceptions whenever an `error` event is emitted, add a listener to the `error` event. This can be done with custom classes or any standard class that inherits from `EventEmitter`.

Discussion

To handle errors, bind a listener to the `error` event. The following listing demonstrates this by building on the music player example.

Listing 4.7 Event-based errors

```
var util = require('util');
var events = require('events');

function MusicPlayer() {
  events.EventEmitter.call(this);
}

util.inherits(MusicPlayer, events.EventEmitter);

var musicPlayer = new MusicPlayer();

musicPlayer.on('play', function(track) {
  this.emit('error', 'unable to play!');
});

musicPlayer.on('error', function(err) {
  console.error('Error:', err);
});

setTimeout(function() {
  musicPlayer.emit('play', 'Little Comets - Jennifer');
}, 1000);
```

Listening for an error event

This example is perhaps simple, but it's useful because it should help you realize how `EventEmitter` handles errors. It feels like a special case, and that's because it is. The following excerpt is from the Node documentation:

> *When an `EventEmitter` instance experiences an error, the typical action is to emit an `error` event. Error events are treated as a special case in Node. If there is no listener for it, then the default action is to print a stack trace and exit the program.*

You can try this out by removing the `'error'` handler from listing 4.7. A stack trace should be displayed in the console.

This makes sense semantically—otherwise the absence of an error handler would lead to potentially dangerous activity going unnoticed. The event name, or *type* as it's referred to internally, has to appear exactly as `error`—extra spaces, punctuation, or uppercase letters won't be considered an `error` event.

This convention means there's a great deal of consistency across event-based error-handling code. It might be a special case, but it's one worth paying attention to.

TECHNIQUE 22 Managing errors with domains

Dealing with errors from multiple instances of `EventEmitter` can feel like hard work ... unless domains are used!

Problem

You're dealing with multiple non-blocking APIs, but are struggling to effectively handle errors.

Solution

Node's `domain` module can be used to centralize error handling for a set of asynchronous operations, and this includes `EventEmitter` instances that emit unhandled `error` events.

Discussion

Node's `domain` API provides a way of wrapping existing non-blocking APIs and exceptions with error handlers. This helps centralize error handling, and is particularly useful in cases where multiple interdependent I/O operations are being used.

Listing 4.8 builds on the music player example by using two `EventEmitter` descendants to show how a single error handler can be used to handle errors for separate objects.

Listing 4.8 Managing errors with `domain`

```
var util = require('util');
var domain = require('domain');
var events = require('events');
var audioDomain = domain.create();

function AudioDevice() {
  events.EventEmitter.call(this);
  this.on('play', this.play.bind(this));
}

util.inherits(AudioDevice, events.EventEmitter);

AudioDevice.prototype.play = function() {
  this.emit('error', 'not implemented yet');
};

function MusicPlayer() {
  events.EventEmitter.call(this);

  this.audioDevice = new AudioDevice();
  this.on('play', this.play.bind(this));

  this.emit('error', 'No audio tracks are available');
}

util.inherits(MusicPlayer, events.EventEmitter);
```

The Domain module must be loaded, and then a suitable instance created with the create method.

This error and any other errors will be caught by the same error handler.

```
MusicPlayer.prototype.play = function() {
  this.audioDevice.emit('play');
  console.log('Now playing');
};

audioDomain.on('error', function(err) {
  console.log('audioDomain error:', err);
});

audioDomain.run(function() {
  var musicPlayer = new MusicPlayer();
  musicPlayer.play();
});
```

Any code that raises errors inside this callback will be covered by the domain.

Domains can be used with `EventEmitter` descendants, networking code, and also the asynchronous file system methods.

To visualize how domains work, imagine that the `domain.run` callback wraps around your code, even when the code inside the callback triggers events that occur outside of it. Any errors that are thrown will still be caught by the domain. Figure 4.1 illustrates this process.

Without a domain, any errors raised using `throw` could potentially place the interpreter in an unknown state. Domains avoid this and help you handle errors more gracefully.

Now that you know how to inherit from `EventEmitter` and handle errors, you should be starting to see all kinds of useful ways that it can be used. The next section broadens these techniques by introducing some advanced usage patterns and higher-level solutions to program structure issues relating to events.

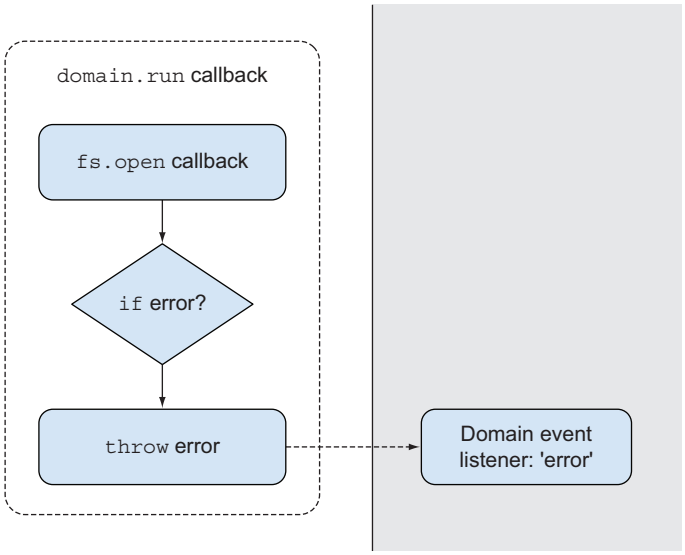


Figure 4.1 Domains help catch errors and handle them with an `EventEmitter`-style API.

4.3 Advanced patterns

This section offers some best practice techniques for solving structural issues found when using `EventEmitter`.

TECHNIQUE 23 Reflection

Sometimes you need to dynamically respond to changes to an instance of an `EventEmitter`, or query its listeners. This technique explains how to do this.

Problem

You need to either catch when a listener has been added to an emitter, or query the existing listeners.

Solution

To track when listeners are added, `EventEmitter` emits a special event called `newListener`. Listeners added to this event will receive the event name and the listener function.

Discussion

In some ways, the difference between writing good Node code and *great* Node code comes down to a deep understanding of `EventEmitter`. Being able to correctly reflect on `EventEmitter` objects gives rise to a whole range of opportunities for creating more flexible and intuitive APIs. One dynamic way of doing this is through the `newListener` event, emitted when listeners are added using the `on` method. Interestingly, this event is emitted by using `EventEmitter` itself—it's implemented by using `emit`.

The next listing shows how to track `newListener` events.

Listing 4.9 Keeping tabs on new listeners

```
var util = require('util');
var events = require('events');

function EventTracker() {
  events.EventEmitter.call(this);
}

util.inherits(EventTracker, events.EventEmitter);

var eventTracker = new EventTracker();

eventTracker.on('newListener', function(name, listener) {
  console.log('Event name added:', name);
});

eventTracker.on('a listener', function() {
  // This will cause 'newListener' to fire
});
```

Track whenever new listeners are added.

Even though `'a listener'` is never explicitly emitted in this example, the `newListener` event will still fire. Since the listener's callback function is passed as well as the event name, this is a great way to create simplified public APIs for things that require

access to the original listener function. Listing 4.10 demonstrates this concept by automatically starting a timer when listeners for `pulse` events are added.

Listing 4.10 Automatically triggering events based on new listeners

```
var util = require('util');
var events = require('events');

function Pulsar(speed, times) {
  events.EventEmitter.call(this);

  var self = this;
  this.speed = speed;
  this.times = times;

  this.on('newListener', function(eventName, listener) {
    if (eventName === 'pulse') {
      self.start();
    }
  });
}

util.inherits(Pulsar, events.EventEmitter);

Pulsar.prototype.start = function() {
  var self = this;
  var id = setInterval(function() {
    self.emit('pulse');
    self.times--;
    if (self.times === 0) {
      clearInterval(id);
    }
  }, this.speed);
};

var pulsar = new Pulsar(500, 5);

pulsar.on('pulse', function() {
  console.log('.');
});
```

Display a dot for each pulse.

We can go a step further and query `EventEmitter` objects about their listeners by calling `emitter.listeners(event)`. A list of *all* listeners can't be returned in one go, though. The entire list is technically available within the `this._events` object, but this property should be considered private. The `listeners` method currently returns an `Array` instance. This could be used to iterate over multiple listeners if several have been added to a given event—perhaps to remove them at the end of an asynchronous process, or simply to check if any listeners have been added.

In cases where an array of events is available, the `listeners` method will effectively return `this._events[type].slice(0)`. Calling `slice` on an array is a JavaScript shortcut for creating a *copy* of an array. The documentation states that this behavior may

change in the future, so if you really want to create a copy of attached listeners, then call `slice` yourself to ensure you really get a copy and not a reference to a data structure within the emitter instance.

Listing 4.11 adds a `stop` method to the `Pulsar` class. When `stop` is called, it checks to see if there are any listeners; otherwise, it raises an error. Checking for listeners is a good way to prevent incorrect usage, but you don't have to do this in your own code.

Listing 4.11 Querying listeners

```
Pulsar.prototype.stop = function() {
  if (this.listeners('pulse').length === 0) {
    throw new Error('No listeners have been added!');
  }
};

var pulsar = new Pulsar(500, 5);

pulsar.stop();
```

TECHNIQUE 24 Detecting and exploiting EventEmitter

A lot of successful open source Node modules are built on `EventEmitter`. It's useful to spot where `EventEmitter` is being used and to know how to take advantage of it.

Problem

You're working on a large project with several components and want to communicate between them.

Solution

Look for the `emit` and `on` methods whenever you're using either Node's standard modules or open source libraries. For example, the Express `app` object has these methods, and they're great for sending messages within an application.

Discussion

Usually when you're working on a large project, there's a major component that's central to your problem domain. If you're building a web application with Express, then the `app` object is one such component. A quick check of the source shows that this object mixes in `EventEmitter`, so you can take advantage of events to communicate between the disparate components within your project.

Listing 4.12 shows an Express-based example where a listener is bound to an event, and then the event is emitted when a specific route is accessed.

Listing 4.12 Reusing `EventEmitter` in Express

```
var express = require('express');
var app = express();

app.on('hello-alert', function() {
  console.warn('Warning!');
});
```

```
app.get('/', function(req, res){
  res.app.emit('hello-alert');
  res.send('hello world');
});

app.listen(3000);
```

The app object is also available in res.app.

This might seem contrived, but what if the route were defined in another file? In this case, you wouldn't have access to the `app` object, unless it was defined as a global.

Another example of a popular project built on `EventEmitter` is the Node Redis client (https://npmjs.org/package/redis). Instances of `RedisClient` inherit from `EventEmitter`. This allows you to hook into useful events, like the `error` event, as shown in the next listing.

Listing 4.13 Reusing `EventEmitter` in the `redis` module

```
var redis = require('redis'),
var client = redis.createClient();

client.on('error', function(err) {
  console.error('Error:', err);
});

client.on('monitor', function(timestamp, args) {
  console.log('Time:', timestamp, 'arguments:', args);
});

client.on('ready', function() {
  // Start app here
});
```

The monitor event emitted by the redis module for tracking when various internal activities occur

In cases where the route separation technique has been used to store routes in several files, you can actually send events by calling `res.app.emit(event)`. This allows route handlers to communicate back to the `app` object itself.

This might seem like a highly specific Express example, but other popular open source modules are also built on `EventEmitter`—just look for the `emit` and `on` methods. Remember that Node's internal modules like the `process` object and `net.createServer` inherit from `EventEmitter`, and well-written open source modules tend to inherit from these modules as well. This means there's a huge amount of scope for event-based solutions to architectural problems.

This example also highlights another benefit of building projects around `EventEmitter`—asynchronous processes can respond as soon as possible. If the `hello-alert` event performs a very slow operation like sending an email, the person browsing the page might not want to wait for this process to finish. In this case, you can render the requested page while effectively performing a slower operation in the background.

The Node Redis client makes excellent use of `EventEmitter` and the author has written documentation for what each of the methods do. This is a good idea—if

somebody joins your project, they may find it hard to get an overall picture of the events that are being used.

TECHNIQUE 25 Categorizing event names

Some projects just have too many events. This technique shows how to deal with bugs caused by mistyped event names.

Problem

You're losing track of the events in your program, and are concerned that it may be too easy to write an incorrect event name somewhere causing a difficult-to-track bug.

Solution

The easiest way to solve this problem is to use an object to act as a central dictionary for all of the event names. This creates a centralized location of each event in the project.

Discussion

It's hard to keep track of event names littered throughout a project. One way to manage this is to keep each event name in one place. Listing 4.14 demonstrates using an object to categorize event names, based on the previous examples in this chapter.

Listing 4.14 Categorizing event names using an object

```
var util = require('util');
var events = require('events');

function MusicPlayer() {
  events.EventEmitter.call(this);
  this.on(MusicPlayer.events.play, this.play.bind(this));
}

var e = MusicPlayer.events = {
  play: 'play',
  pause: 'pause',
  stop: 'stop',
  ff: 'ff',
  rw: 'rw',
  addTrack: 'add-track'
};

util.inherits(MusicPlayer, events.EventEmitter);

MusicPlayer.prototype.play = function() {
  this.playing = true;
};

var musicPlayer = new MusicPlayer();

musicPlayer.on(e.play, function() {
  console.log('Now playing');
});

musicPlayer.emit(e.play);
```

The object used to store the event list is aliased for convenience.

When adding new listeners, users of the class can refer to the events list rather than writing the event names as strings.

Although `EventEmitter` is an integral part of Node's standard library, and an elegant solution to many problems, it can be the source of a lot of bugs in larger projects where people may forget the name of a given event. One way around this is to avoid writing events as strings. Instead, an object can be used with properties that refer to the event name strings.

If you're writing a reusable, open source module, you should consider making this part of the public API so it's easy for people to get a centralized list of event names.

There are other observer pattern implementations that avoid using string event names to effectively type check events. In the next technique we'll look at a few that are available through npm.

Although `EventEmitter` provides a wide array of solutions when working on Node projects, there are alternative implementations out there. The next section includes some popular alternatives.

4.4 Third-party modules and extensions

`EventEmitter` is essentially an *observer pattern* implementation. There are other interpretations of this pattern, which can help scale Node programs to run across several processes or over a network. The next technique introduces some of the more popular alternatives created by the Node community.

TECHNIQUE 26 Alternatives to EventEmitter

`EventEmitter` has a great API and works well in Node programs, but sometimes a problem requires a slightly different solution. This technique explores some alternatives to `EventEmitter`.

Problem

You're trying to solve a problem that doesn't quite fit `EventEmitter`.

Solution

Depending on the exact nature of the problem you're trying to solve, there are several alternatives to `EventEmitter`: publish/subscribe, AMQP, and `js-signals` are some popular alternatives with good support in Node.

Discussion

The `EventEmitter` class is an implementation of the *observer pattern*. A related pattern is publish/subscribe, where publishers send messages that are characterized into classes to subscribers without knowing the details of the subscribers themselves.

The publish/subscribe pattern is often useful in cases where horizontal scaling is required. If you need to run multiple Node processes on multiple servers, then technologies like AMQP and ØMQ can help implement this. They're both specifically designed to solve this class of problem, but may not be as convenient as using the Redis publish/subscribe API if you're already using Redis.

If you need to horizontally scale across a distributed cluster, then an AMQP implementation like RabbitMQ (http://www.rabbitmq.com/) will work well. The `rabbitmq-nodejs-client` (https://github.com/adrai/rabbitmq-nodejs-client) module has a

publish/subscribe API. The following listing shows a simple example of RabbitMQ in Node.

Listing 4.15 Using RabbitMQ with Node

```
var rabbitHub = require('rabbitmq-nodejs-client');
var subHub = rabbitHub.create( { task: 'sub', channel: 'myChannel' } );
var pubHub = rabbitHub.create( { task: 'pub', channel: 'myChannel' } );

subHub.on('connection', function(hub) {
  hub.on('message', function(msg) {
    console.log(msg);
  }.bind(this));
});
subHub.connect();

pubHub.on('connection', function(hub) {
  hub.send('Hello World!');
});
pubHub.connect();
```

Print the message when it's received.

ØMQ (http://www.zeromq.org/) is more popular in the Node community. Justin Tulloss and TJ Holowaychuk's `zeromq.node` module (https://github.com/JustinTulloss/zeromq.node) is a popular binding. The next listing shows just how simple this API is.

Listing 4.16 Using ØMQ with Node

```
var zmq = require('zmq');
var push = zmq.socket('push');
var pull = zmq.socket('pull');

push.bindSync('tcp://127.0.0.1:3000');
pull.connect('tcp://127.0.0.1:3000');
console.log('Producer bound to port 3000');

setInterval(function() {
  console.log('sending work');
  push.send('some work');
}, 500);

pull.on('message', function(msg) {
  console.log('work: %s', msg.toString());
});
```

If you're already using Redis with Node, then it's worth trying out the Pub/Sub API (http://redis.io/topics/pubsub). Listing 4.17 shows an example of this using the Node Redis client (https://github.com/mranney/node_redis).

Listing 4.17 Using Redis Pub/Sub with Node

```
var redis = require('redis');
var client1 = redis.createClient();
var client2 = redis.createClient();
```

```
var msg_count = 0;

client1.on('subscribe', function(channel, count) {
  client2.publish('channel', 'Hello world.');
});

client1.on('message', function(channel, message) {
  console.log('client1 channel ' + channel + ': ' + message);
  client1.unsubscribe();
  client1.end();
  client2.end();
});

client1.subscribe('channel');
```

Be sure to close client connections when using the Redis module.

Finally, if publish/subscribe isn't what you're looking for, then you may want to take a look at `js-signals` (https://github.com/millermedeiros/js-signals). This module is a messaging system that doesn't use strings for the signal names, and dispatching or listening to events that don't yet exist will raise errors.

Listing 4.18 shows how `js-signals` sends and receives messages. Notice how signals are properties of an object, rather than strings, and that listeners can receive an arbitrary number of arguments.

Listing 4.18 Using Redis Pub/Sub with Node

```
var signals = require('signals');
var myObject = {
  started: new signals.Signal()
};

function onStarted(param1, param2){
  console.log(param1, param2);
}

myObject.started.add(onStarted);
myObject.started.dispatch('hello', 'world');
```

Binding a listener to the started signal

Dispatching the signal using two parameters

`js-signals` provides a way of using properties for signal names, as mentioned in technique 25, but in this case the module will raise an error if an unregistered listener is dispatched or bound to. This approach is more like "strongly typed" events, and is very different from most publish/subscribe and event observer implementations.

4.5 Summary

In this chapter you've learned how `EventEmitter` is used through inheritance and multiple inheritance, and how to manage errors with and without domains. You've also seen how to centralize event names, how open source modules build on `EventEmitter`, and some alternative solutions.

What you should take away from this chapter is that although `EventEmitter` is usually used as a base class for inheritance, it's also possible to mix it into existing classes. Also, although `EventEmitter` is a great solution to many problems and used throughout

Node's internals, sometimes other solutions are more optimal. For example, if you're using Redis, then you can take advantage of its publish/subscribe implementation. Finally, `EventEmitter` isn't without its problems; managing large amounts of event names can cause bugs, and now you know how to avoid this by using an object with properties that act as event names.

In the next chapter we'll look at a related topic: streams. Streams are built around an event-based API, so you'll be able to use some of these `EventEmitter` techniques there as well.

5 Streams: Node's most powerful and misunderstood feature

This chapter covers

- What streams are and how to use them
- How to use Node's built-in streaming APIs
- The stream API used in Node 0.8 and below
- The stream primitive classes bundled since Node 0.10
- Strategies for testing streams

Streams are an event-based API for managing and modeling data, and are wonderfully efficient. By leveraging `EventEmitter` and Node's non-blocking I/O libraries, the `stream` module allows data to be dynamically processed when it's available, and then released when it's no longer needed.

The idea of a stream of data isn't new, but it's an important concept and integral to Node. After chapter 4, mastering streams is the next step on the path to becoming truly competent at Node development.

The `stream` core module provides abstract tools for building event-based stream classes. It's likely that you'll use modules that implement streams, rather than creating your own. But to exploit streams to their fullest, it's important to understand how they really work. This chapter has been designed with that goal in mind: understanding streams, working with Node's built-in streaming APIs, and finally creating and testing your own streams. Despite the conceptually abstract nature of the `stream` module, once you've mastered the major concepts, you'll start to see uses for streams everywhere.

The next section provides a high-level overview of streams and addresses the two APIs that Node supports as of Node 0.10.

5.1 Introduction to streams

In Node, streams are an *abstract interface* adhered to by several different objects. When we talk about streams, we're referring to a way of doing things—in a sense, they're a protocol. Streams can be readable or writable, and are implemented with instances of `EventEmitter`—see chapter 4 for more on events. Streams provide the means for creating data flows between objects, and can be composed with LEGO-like modularity.

5.1.1 Types of streams

Streams always involve I/O of some kind, and they can be classified into groups based on the type of I/O they deal with. The following types of streams were taken from James Halliday's `stream-handbook` (https://github.com/substack/stream-handbook/), and will give you an idea of the wide array of things you can do with streams:

- *Built-in*—Many of Node's core modules implement streaming interfaces; for example, `fs.createReadStream`.
- *HTTP*—Although technically network streams, there are streaming modules designed to work with various web technologies.
- *Parsers*—Historically parsers have been implemented using streams. Popular third-party modules for Node include XML and JSON parsers.
- *Browser*—Node's event-based streams have been extended to work in browsers, offering some unique opportunities for interfacing with client-side code.
- *Audio*—James Halliday has written some novel audio modules that have streamable interfaces.
- *RPC (Remote Procedure Call)*—Sending streams over the network is a useful way to implement interprocess communication.
- *Test*—There are stream-friendly test libraries, and tools for testing streams themselves.
- *Control, meta, and state*—There are also more abstract uses of streams, and modules designed purely for manipulating and managing other streams.

The best way to understand why streams are important is to first consider what happens when data is processed without them. Let's look at this in more detail by comparing Node's asynchronous, synchronous, and stream-based APIs.

5.1.2 When to use streams

When reading a file synchronously with `fs.readFileSync`, the program will block, and all of the data will be read to memory. Using `fs.readFile` will prevent the program from blocking because it's an asynchronous method, but it'll still read the entire file into memory.

What if there were a way to tell `fs.readFile` to read a chunk of data into memory, process it, and then ask for more data? That's where streams come in.

Memory becomes an issue when working with large files—compressed backup archives, media files, large log files, and so on. Instead of reading the entire file into memory, you could use `fs.read` with a suitable buffer, reading in a specific length at a time. Or, preferably, you could use the streams API provided by `fs.createReadStream`. Figure 5.1 illustrates how only a chunk of a file is read at a time with `fs.createReadStream`, compared to the entire file with `fs.readFile`.

Streams are asynchronous by design. Rather than reading that entire file into memory, a buffer's worth will be read, the desired operations will be performed, and then the result will be written to the output stream. This approach is as close to idiomatic Node as you can get. What's more, streams are implemented with plain old JavaScript. Take `fs.createReadStream`—it offers a more scalable solution, but ultimately just wraps simple file system operations with a better API.

Node's streaming APIs feel idiomatic, yet streams have been around in computer science for a long time. This history is examined briefly in the next section to give you some background on where streams come from and where they're used.

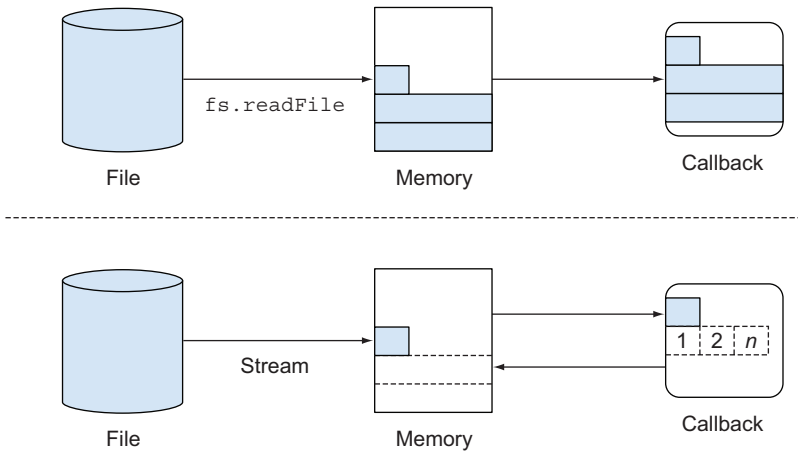


Figure 5.1 Using streamable APIs means I/O operations potentially use less memory.

5.1.3 History

So where did streams originate? Historically, streams in computer science have been used to solve problems similar to streams in Node. For example, in C the standard way to represent a file is by using a stream. When a C program starts, it has access to the standard I/O streams. The standard I/O streams are also available in Node, and can be used to allow programs to work well with large amounts of data in the shell.

Traditionally, streams have been used to implement efficient parsers. This has also been the case in Node: the `node-formidable` module (https://github.com/felixge/node-formidable) is used by Connect to efficiently parse form data with streams, and database modules like the Node `redis` module (https://npmjs.org/package/redis) use streams to represent the connection to the server and respond by parsing on demand.

If you're familiar with Unix, you're probably already aware of streams. If you've used pipes or I/O redirection, then you've used streams. You can literally think about Node streams as you would Unix pipes—except data is filtered through functions instead of command-line programs. The next section explains how streams have evolved in Node, up until version 0.10 when they changed significantly.

STREAMS OLD AND NEW

Streams are part of Node's core modules, and as such remain backward compatible with earlier versions. As of this writing, Node is at version 0.10, which has seen significant changes in the streams API. Though it remains backward compatible, the new streams syntax is in some ways stricter than earlier versions, yet ultimately more flexible. This boils down to the behavior of `pipe`—pipes must now originate from a `Readable` stream and end at a `Writable` stream. The `util.pump` method, found in earlier versions of Node, has now been deprecated in favor of the new `pipe` semantics.

The evolution of streams in Node came from a desire to use the event-based APIs to solve non-blocking I/O problems in an efficient way. Older solutions like `util.pump` sought to find efficiency in intelligent uses of "drain" events—this is emitted when a writable stream has emptied and it's safe to write again. This sounds a lot like pausing a stream, and the handling of paused streams was something the pre-0.10 streams API couldn't handle effectively.

Now Node has reached a point where the core developers have seen the types of problems people are tackling with streams, so the new API is richer thanks to the new stream primitive classes. Table 5.1 shows a summary of the classes available from Node 0.10 onward.

Learning to take advantage of streams will pay dividends when it comes to working with third-party modules that implement streams. In the next section, a selection of popular stream-oriented modules is examined.

5.1.4 Streams in third-party modules

The main use of streams in Node is for creating event-based APIs for I/O-like sources; parsers, network protocols, and database modules are the key examples. A network

Table 5.1 A summary of the classes available in *streams2*

Name	User methods	Description
`stream.Readable`	`_read(size)`	Used for I/O sources that generate data
`stream.Writable`	`_write(chunk, encoding, callback)`	Used to write to an underlying output destination
`stream.Duplex`	`_read(size)`, `_write(chunk, encoding, callback)`	A readable *and* writable stream, like a network connection
`stream.Transform`	`_flush(size)`, `_transform(chunk, encoding, callback)`	A duplex stream that changes data in some way, with no limitation on matching input data size with the output

protocol implemented with streams can be convenient when composition is desired—think about how easy it would be to add data compression to a network protocol if the data could be passed through the `gzip` module with a single call to `pipe`.

Similarly, database libraries that stream data can handle large result sets more efficiently; rather than collecting all results into an array, a single item at a time can be streamed.

The Mongoose MongoDB module (http://mongoosejs.com/) has an object called `QueryStream` that can be used to stream documents. The `mysql` module (https://npmjs.org/package/mysql) can also stream query results, although this implementation doesn't currently implement the `stream.Readable` class.

You can also find more creative uses of streams out there. The `baudio` module (see figure 5.2) by James Halliday can be used to generate audio streams that behave just like any other stream—audio data can be routed to other streams with `pipe`, and recorded for playback by standard audio software:

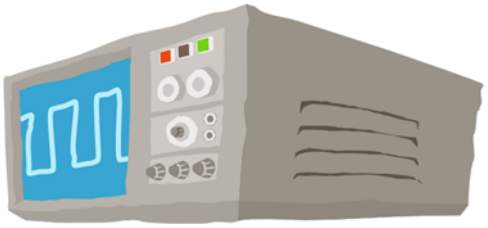


Figure 5.2 The `baudio` module by James Halliday (substack) supports the generation of audio streams (from https://github.com/substack/baudio).

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

var n = 0;
var b = baudio(function (t) {
  var x = Math.sin(t * 262 + Math.sin(n));
  n += Math.sin(t);
  return x;
});
b.play();
```

When selecting a network or database library for your Node projects, we strongly recommend ensuring it has a streamable API, because it'll help you write more elegant code while also potentially offering performance benefits.

One thing all stream classes have in common is they inherit from `EventEmitter`. The significance of this is investigated in the next section.

5.1.5 Streams inherit from EventEmitter

Each of the `stream` module base classes emits various events, which depend on whether the base class is readable, writable, or both. The fact that streams inherit from `EventEmitter` means you can bind to various standard events to manage streams, or create your own custom events to represent more domain-specific behavior.

When working with `stream.Readable` instances (see table 5.2 for guidance on selecting a stream base class), the `readable` event is important because it signifies that the stream is ready for calls to `stream.read()`.

Attaching a listener to `data` will cause the stream to behave like the old streams API, where data is passed to `data` listeners when it's available, rather than through calls to `stream.read()`.

The `error` event is covered in detail in technique 28. It'll be emitted if the stream encounters an error when receiving data.

The `end` event signifies that the stream has received an equivalent of the end-of-file character, and won't receive more data. There's also a `close` event that represents the case where the underlying resource has been closed, which is distinct from `end`, and the Node API documentation notes that not all streams will emit this event, so a rule of thumb is to bind to `end`.

The `stream.Writable` class changes the semantics for signifying the end of a stream to `close` and `finish`. The distinction between the two is that `finish` is emitted when `writable.end()` is called, whereas `close` means the underlying I/O resource has been closed, which isn't always required, depending on the nature of the underlying stream.

The `pipe` and `unpipe` events are emitted when passing a stream to the `stream.Readable.prototype.pipe` method. This can be used to adapt the way a stream behaves when it's piped. The listener receives the destination stream as the first argument, so this value could be inspected to change the behavior of the stream. This is a more advanced technique that's covered in technique 37.

> **About the techniques in this chapter**
>
> The techniques in this chapter all use the *streams2* API. This is the nickname of the newer API style found in Node 0.10 and 0.12. If you're using Node 0.8, forward compatibility is supported through the `readable-stream` module (https://github.com/isaacs/readable-stream).

In the next section you'll learn how to solve real-world problems using streams. First we'll discuss some of Node's built-in streams, and then we'll move on to creating entirely new streams and testing them.

5.2 Built-in streams

Node's core modules themselves are implemented using the `stream` module, so it's easy to start using streams without having to build your own classes. The next technique introduces some of this functionality through file system and network streaming APIs.

TECHNIQUE 27 Using built-in streams to make a static web server

Node's core modules often have streamable interfaces. They can be used to solve many problems more efficiently than their synchronous alternatives.

Problem

You want to send a file from a web server to a client in an efficient manner that will scale up to large files.

Solution

Use `fs.createReadStream` to open a file and *stream* it to the client. Optionally, `pipe` the resulting `stream.Readable` through another stream to handle features like compression.

Discussion

Node's core modules for file system and network operations, `fs` and `net`, both provide streamable interfaces. The `fs` module has helper methods to automatically create instances of the streamable classes. This makes using streams for some I/O-based problems fairly straightforward.

To understand why streams are important and compare them to nonstreaming code, consider the following example of a simple static file web server made with Node's core modules:

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

http.createServer(function(req, res) {
  fs.readFile(__dirname + '/index.html', function(err, data) { //
     if (err) {
      res.statusCode = 500;
      res.end(String(err));
    } else {
      res.end(data);
    }
  });
}).listen(8000);
```

Even though this code uses the `fs.readFile` method, which is non-blocking, it can easily be improved on by using `fs.createReadStream`. The reason is because it'll read the entire file into memory. This might seem acceptable with small files, but what if you don't know how large the file is? Static web servers often have to serve up potentially large binary assets, so a more adaptable solution is desirable.

The following listing demonstrates a streaming static web server.

Listing 5.1 A simple static web server that uses streams

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

http.createServer(function(req, res) {
  fs.createReadStream(__dirname + '/index.html').pipe(res);
}).listen(8000);
```

Data is piped from a file to Node's HTTP Response object.

This example uses less code than the first version, and improves its efficiency. Now instead of reading the entire file into memory, a buffer's worth will be read at a time and sent to the client. If the client is on a slow connection, the network stream will signal this by requesting that the I/O source pauses until the client is ready for more data. This is known as *backpressure*, and is one of the additional benefits using streams brings to your Node programs.

We can take this example a step further. Streams aren't just efficient and potentially more syntactically elegant, they're also extensible. Static web servers often compress files with gzip. The next listing adds that to the previous example, using streams.

Listing 5.2 A static web server with gzip

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

http.createServer(function(req, res) {
  res.writeHead(200, { 'content-encoding': 'gzip' });
  fs.createReadStream(__dirname + '/index.html')
    .pipe(zlib.createGzip())
    .pipe(res);
}).listen(8000);
```

Set the header so the browser knows gzip compression has been used.

Use two calls to pipe, compress, and stream the file back to the client.

Now if you open http://localhost:8000 in a browser and use its debugging tools to look at the network operations, you should see that the content was transferred using gzip. Figure 5.3 shows what our browser reported after running the example.

This could be expanded in several other ways—you can use as many calls to `pipe` as required. For example, the file could be piped through an HTML templating engine and then compressed. Just remember that the general pattern is `readable.pipe(writable)`.

Note that this example is simplified to illustrate how streams work and isn't sufficient for implementing a production HTTP asset server.

Now that you've seen a fleshed-out example of how streams are used to solve a common problem, it's time to look at another piece of the puzzle: error handling.

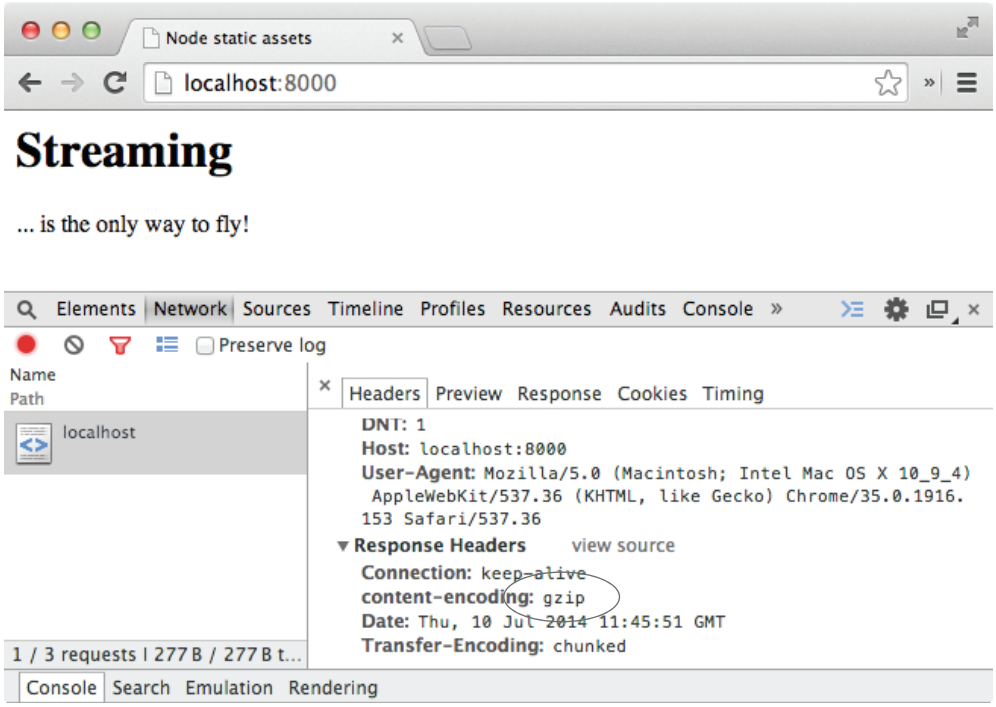


Figure 5.3 **The network inspector confirms the content was compressed.**

TECHNIQUE 28 Stream error handling

The stream classes inherit from `EventEmitter`, which means sane error handling comes as standard. This technique explains how to handle errors generated by a stream.

Problem

You want to catch errors generated by a stream.

Solution

Add an `error` listener.

Discussion

The standard behavior of `EventEmitter` is to throw an exception when an `error` event is emitted—unless there's a listener attached to the `error` event. The first argument to the listener will be the error that was raised, a descendent of the `Error` object.

The following listing shows an example of an intentionally generated error with a suitable `error` listener.

Listing 5.3 Catching errors during streaming

```
var fs = require('fs');
var stream = fs.createReadStream('not-found');

stream.on('error', function(err) {
  console.trace();
  console.error('Stack:', err.stack);
  console.error('The error raised was:', err);
});
```

Cause an error to be generated by trying to open a file that doesn't exist. (1)

Use the events API to attach an error handler. (2)

Here we attempt to open a file that doesn't exist (1), causing an `'error'` event to be triggered. The error object passed to the handler (2) will usually have extra information to aid in tracking down the error. For example, the `stack` property may have line number information, and `console.trace()` can be called to generate a full stack trace. In listing 5.3 `console.trace()` will show a trace up to the `ReadStream` implementation in Node's `events.js` core module. That means you can see exactly where the error was originally emitted.

Now that you've seen how some of Node's core modules use streams, the next section explores how third-party modules use them.

5.3 Third-party modules and streams

Streams are about as idiomatic Node as you can get, so it's no surprise that streamable interfaces crop up all over the open source Node landscape. In the next technique you'll learn how to use streamable interfaces found in some popular Node modules.

TECHNIQUE 29 Using streams from third-party modules

Many open source developers have recognized the importance of streams and incorporated streamable interfaces into their modules. In this technique you'll learn how to identify such implementations and use them to solve problems more efficiently.

Problem

You want to know how to use streams with a popular third-party module that you've downloaded with npm.

Solution

Look at the module's documentation or source code to figure out if it implements a streamable API, and if so, how to use it.

Discussion

We've picked three popular modules as examples of third-party modules that implement streamable interfaces. This guided tour of streams in the wild should give you a good idea of how developers are using streams, and how you can exploit streams in your own projects.

In the next section you'll discover some key ways to use streams with the popular web framework, Express.

Using streams with Express

The Express web framework (http://expressjs.com/) actually provides a relatively lightweight wrapper around Node's core HTTP module. This includes the `Request`

and `Response` objects. Express decorates these objects with some of its own methods and values, but the underlying objects are the same. That means everything you learned about streaming data to browsers in technique 27 can be reused here.

A simple example of an Express *route*—a callback that runs for a given HTTP method and URL—uses `res.send` to respond with some data:

```
var express = require('express');
var app = express();

app.get('/', function(req, res) {
  res.send('hello world');
});

app.listen(3000);
```

The `res` object is actually a *response* object, and inherits from Node's `http.ServerResponse`. In technique 27 you saw that HTTP requests can be streamed to by using the `pipe` method. Express is built in a way that allows buffers and objects to work with the `res.send` method, and for streams you can still use the `pipe` method.

Listing 5.4 is an Express web application that will run with Express 3 and streams content from a custom-readable stream by using `pipe`.

Listing 5.4 An Express application that uses streams

```
var stream = require('stream');
var util = require('util');
var express = require('express');
var app = express();

util.inherits(StatStream, stream.Readable);

function StatStream(limit) {
  stream.Readable.call(this);
  this.limit = limit;
}

StatStream.prototype._read = function(size) {
  if (this.limit === 0) {
    // Done
    this.push();
  } else {
    this.push(util.inspect(process.memoryUsage()));
    this.push('n');
    this.limit--;
  }
};

app.get('/', function(req, res) {
  var statStream = new StatStream(10);
  statStream.pipe(res);
});

app.listen(3000);
```

1. **Create a readable stream by inheriting from stream.Readable and calling the parent's constructor.**
2. **Respond with some data—this sends a string representation of the Node process's memory usage.**
3. **Use the standard readable.pipe(writable) pattern to send data back to the browser.**

Our custom readable stream, `StatStream`, inherits from `stream.Readable` (1) and implements the `_read` method, which just sends memory usage data (2). The `_read` method must be implemented whenever you want to make a readable stream. When sending the response back to the browser, the stream can be piped to the `res` object (3) provided by Express without any extra work.

The implementation of the `send` module that comes with Express 3 uses `fs.createReadStream`, as described in technique 27. The following sample code is taken from the source to `send`:

```
SendStream.prototype.stream = function(path, options){
  TODO: this is all lame, refactor meeee
  var self = this;
  var res = this.res;
  var req = this.req;

  pipe
  var stream = fs.createReadStream(path, options);
  this.emit('stream', stream);
  stream.pipe(res);
```

It takes a lot more work to correctly deal with things like HTTP `Content-Range` headers, but this snippet demonstrates that leveraging the built-in streaming APIs like `fs.createReadStream` can lead to solutions powerful enough to underpin major open source projects.

Using streams with Mongoose

The Mongoose module (http://mongoosejs.com/) for the MongoDB database server (http://www.mongodb.org/) has an interface called `QueryStream` that provides Node 0.8-style streams for query results. This class is used internally to allow query results to be streamed using the `stream` method. The following code shows a query that has its results piped through a hypothetical writable stream:

```
User
  .where('role')
  .equals('admin')
  .stream()
  .pipe(writeStream);
```

This pattern—using a class to wrap an external I/O source's streamable behavior, and then exposing streams through simple method calls—is the style employed by Node's core modules, and is popular with third-party module authors. This has been made clearer by the streams2 API's use of simple abstract classes that can be inherited from.

Using streams with MySQL

The third-party `mysql` module (https://npmjs.org/package/mysql) is often seen by Node developers as something low-level that should be built on with more complex libraries, like the Sequelize (http://www.sequelizejs.com/) object-relational mapper (ORM). But the `mysql` module itself shouldn't be underestimated, and supports streaming results with `pause` and `resume`. Here's an example of the basic API style:

```
var query = connection.query('SELECT * FROM posts');
query
.on('result', function(row) {
  connection.pause();
  processRow(row, function() {
    connection.resume();
  });
});
```

This streaming API uses domain-specific event names—there's also a `'fields'` event. To pause the result stream, `connection.pause` must be called. This signals to the underlying connection to MySQL that results should stop briefly until the receiver is ready for more data.

Summary

In this technique you've seen how some popular third-party modules use streams. They're all characterized by the fact they deal with I/O—both HTTP and database connections are network- or file-based protocols, and both can involve network connections and file system operations. In general, it's a good idea to look for Node network and database modules that implement streamable interfaces, because they help scale programs and also write them in a readable, idiomatic style.

Now that you've seen how to use streams, you're probably itching to learn how to create your own. The next section has a technique for using each base stream class, and also shows how to correctly inherit from them.

5.4 Using the stream base classes

Node's base stream classes provide templates for solving the kinds of problems that streams are best at. For example, `stream.Transform` is great for parsing data, and `stream.Readable` is perfect for wrapping lower-level APIs with a streamable interface.

The next technique explains how to inherit from the stream base classes, and then further techniques go into detail about how to use each base class.

TECHNIQUE 30 Correctly inheriting from the stream base classes

Node's base classes for streams can be used as a starting point for new modules and subclasses. It's important to understand what problems each solves, and how to correctly inherit from them.

Problem

You want to solve a problem by creating a streamable API, but you're not sure which base class to use and how to use it.

Solution

Decide on which base class closely matches the problem at hand, and inherit from it using `Object.prototype.call` and `util.inherits`.

Discussion

Node's base classes for streams, already summarized in table 5.1, should be used as the basis for your own streamable classes or modules. They're *abstract classes*, which means

they're methods that you must implement before they can be used. This is usually done through inheritance.

All of the stream base classes are found in the `stream` core module. The five base classes are `Readable`, `Writable`, `Duplex`, `Transform`, and `PassThrough`. Fundamentally, streams are either readable or writable, but `Duplex` streams are both. This makes sense if you consider the behavior of I/O interfaces—a network connection can be both readable and writable. It wouldn't be particularly useful, for example, if `ssh` were only able to send data.

`Transform` streams build on `Duplex` streams, but also change the data in some way. Some of Node's built-in modules use `Transform` streams, so they're fundamentally important. An example of this is the `crypto` module.

Table 5.2 offers some hints to help you choose which base class to use.

Table 5.2 Selecting a streams base class

Problem	Solution
You want to wrap around an underlying I/O source with a streamable API.	`Readable`
You want to get output from a program to use elsewhere, or send data elsewhere within a program.	`Writable`
You want to change data in some way by parsing it.	`Transform`
You want to wrap a data source that can also receive messages.	`Duplex`
You want to extract data from streams without changing it, from testing to analysis.	`PassThrough`

Inheriting from the base classes

If you've learned about inheritance in JavaScript, you might be tempted to inherit from the stream base classes by using `MyStream.prototype = new stream.Readable();`. This is considered bad practice, and it's better to use the ECMAScript 5 `Object.create` pattern instead. Also, the base class's constructor must be run, because it provides essential setup code. The pattern for this is shown next.

Listing 5.5 Inheriting from the `stream.Readable` base class

```
var Readable = require('stream').Readable;

function MyStream(options) {
  Readable.call(this, options);
}

MyStream.prototype = Object.create(Readable.prototype, {
  constructor: { value: MyStream }
});
```

1 Call the parent constructor, and be sure to pass any options to it as well.

2 Use Object.create to correctly set up the prototype chain.

Node includes a utility method called `util.inherits` that can be used instead of `Object.create`, but both approaches are widely used by Node developers. This example uses the `Object.create` method 1 instead so you can see what `util.inherits` does.

Note that in listing 5.5 the `options` argument ❷ is passed to the original `Readable` constructor. This is important because there's a standard set of options that Node supports for configuring streams. In the case of `Readable`, the options are as follows:

- *`highWaterMark`*—The number of bytes to store in the internal buffer before pausing reading from the underlying data source.
- *`encoding`*—Causes the buffer to be automatically decoded. Possible values include `utf8` and `ascii`.
- *`objectMode`*—Allows the stream to behave as a stream of objects, rather than bytes.

The `objectMode` option allows JavaScript objects to be handled by streams. An example of this has been provided in technique 31.

Summary

In this technique you've seen how to use Node's stream base classes to create your own stream implementations. This involves using `util.inherits` to set up the class, and then `.call` to call the original constructor. We also covered some of the options that these base classes use.

Properly inheriting from the base classes is one thing, but what about actually implementing a stream class? Technique 31 explains this in more detail for the `Readable` base class, but in that specific case it involves implementing a method called `_read` to read data from the underlying data source and `push` it onto an internal queue managed by the base class itself.

TECHNIQUE 31 Implementing a readable stream

Readable streams can be used to provide a flexible API around I/O sources, and can also act as parsers.

Problem

You'd like to wrap an I/O source with a streamable API that provides a higher-level interface than would otherwise be possible with the underlying data.

Solution

Implement a readable stream by inheriting from the `stream.Readable` class and creating a `_read(size)` method.

Discussion

Implementing a custom `stream.Readable` class can be useful when a higher level of abstraction around an underlying data source is required. For example, I (Alex) was working on a project where the client had sent in JSON files that contained millions of records separated by newlines. I decided to write a quick `stream.Readable` class that read a buffer's worth of data, and whenever a newline was encountered, `JSON.parse` was used to parse the record.

One way of using `stream.Readable` to parse newline-separated JSON records is shown next.

Listing 5.6 A JSON line parser

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

function JSONLineReader(source) {
  stream.Readable.call(this);
  this._source = source;
  this._foundLineEnd = false;
  this._buffer = '';

  source.on('readable', function() {
    this.read();
  }.bind(this));
}

util.inherits(JSONLineReader, stream.Readable);

JSONLineReader.prototype._read = function(size) {
  var chunk;
  var line;
  var lineIndex;
  var result;

  if (this._buffer.length === 0) {
    chunk = this._source.read();
    this._buffer += chunk;
  }

  lineIndex = this._buffer.indexOf('n');

  if (lineIndex !== -1) {
    line = this._buffer.slice(0, lineIndex);
    if (line) {
      result = JSON.parse(line);
      this._buffer = this._buffer.slice(lineIndex + 1);
      this.emit('object', result);
      this.push(util.inspect(result));
    } else {
      this._buffer = this._buffer.slice(1);
    }
  }
};

var input = fs.createReadStream(__dirname + '/json-lines.txt', {
  encoding: 'utf8'
});
var jsonLineReader = new JSONLineReader(input);

jsonLineReader.on('object', function(obj) {
  console.log('pos:', obj.position, '- letter:', obj.letter);
});
```

1. Always ensure the constructor's parent is called.
2. Call read() when the source is ready to trigger subsequent reads.
3. Inherit from stream.Readable to create a new class that can be customized.
4. All custom stream.Readable classes must implement the _read() method.
5. When the class is ready for more data, call read() on the source.
6. Slice from the start of the buffer to the first newline to grab some text to parse.
7. Emitting an "object" event whenever a JSON record has been parsed is unique to this class and isn't necessarily required.
8. Send the parsed JSON back to the internal queue.
9. Create an instance of JSONLineReader and give it a file stream to process.

Listing 5.6 uses a constructor function, `JSONLineReader` (1), that inherits from `stream.Readable` (3) to read and parse lines of JSON from a file. The source for `JSONLineReader` will be a readable stream as well, so a listener for the `readable` event is bound to, so instances of `JSONLineReader` know when to start reading data (2).

The `_read` method (4) checks whether the buffer is empty (5) and, if so, reads more data from the source and adds it to the internal buffer. Then the current line index is incremented, and if a line ending is found, the first line is sliced from the buffer (6). Once a complete line has been found, it's parsed and emitted using the `object` event (7)—users of the class can bind to this event to receive each line of JSON that's found in the source stream.

When this example is run, data from a file will flow through an instance of the class. Internally, data will be queued. Whenever `source.read` is executed, the latest "chunk" of data will be returned, so it can be processed when `JSONLineReader` is ready for it. Once enough data has been read and a newline has been found, the data will be split *up to the first newline*, and then the result will be collected by calling `this.push` (8).

Once `this.push` is called, `stream.Readable` will queue the result and forward it on to a consuming stream. This allows the stream to be further processed by a writable stream using `pipe`. In this example JSON objects are emitted using a custom `object` event. The last few lines of this example attach an event listener for this event and process the results (9).

The `size` argument to `Readable.prototype._read` is *advisory*. That means the underlying implementation can use it to know how much data to fetch—this isn't always needed so you don't always implement it. In the previous example we parsed the entire line, but some data formats could be parsed in chunks, in which case the size argument would be useful.

In the original code that I based this example on, I used the resulting JSON objects to populate a database. The data was also redirected and gzipped into another file. Using streams made this both easy to write and easy to read in the final project.

The example in listing 5.6 used strings, but what about objects? Most streams that deal directly with I/O—files, network protocols, and so on—will use raw bytes or strings of characters. But sometimes it's useful to create streams of JavaScript objects. Listing 5.7 shows how to safely inherit from `stream.Readable` and pass the `objectMode` option to set up a stream that deals with JavaScript objects.

Listing 5.7 A stream configured to use `objectMode`

```
var stream = require('stream');
var util = require('util');

util.inherits(MemoryStream, stream.Readable);

function MemoryStream(options) {
  options = options || {};
  options.objectMode = true;
  stream.Readable.call(this, options);
}
```

(1) **This stream should always use objectMode, so set it here and pass the rest of the options to the stream.Readable constructor.**

```
MemoryStream.prototype._read = function(size) {
  this.push(process.memoryUsage());
};

var memoryStream = new MemoryStream();
memoryStream.on('readable', function() {
  var output = memoryStream.read();
  console.log('Type: %s, value: %j', typeof output, output);
});
```

2 Generate an object by calling Node's built-in process.memoryUsage() method.

Attach a listener to readable to track when the stream is ready to output data; then call stream.read() to fetch recent values. 3

The `MemoryStream` example in listing 5.7 uses objects for data, so `objectMode` is passed to the `Readable` constructor as an option 1. Then `process.memoryUsage` is used to generate some suitable data 2. When an instance of this class emits `readable` 3, indicating that it's ready to be read from, then the memory usage data is logged to the console.

When using `objectMode`, the underlying behavior of the stream is changed to remove the internal buffer merge and length checks, and to ignore the size argument when reading and writing.

TECHNIQUE 32 Implementing a writable stream

Writable streams can be used to output data to underlying I/O sinks.

Problem

You want to output data from a program using an I/O destination that you want to wrap with a streamable interface.

Solution

Inherit from `stream.Writable` and implement a `_write` method to send data to the underlying resource.

Discussion

As you saw in technique 29, many third-party modules offer streamable interfaces for network services and databases. Following this trend is advantageous because it allows your classes to be used with the `pipe` API, which helps keep chunks of code reusable and decoupled.

You might be simply looking to implement a writable stream to act as the destination of a `pipe` chain, or to implement an unsupported I/O resource. In general, all you need to do is correctly inherit from `stream.Writable`—for more on the recommended way to do this, see technique 30—and then add a `_write` method.

All the `_write` method needs to do is call a supplied callback when the data has been written. The following code shows the method's arguments and the overall structure of a sample `_write` implementation:

```
MyWritable.prototype._write = function(chunk, encoding, callback) {
  this.customWriteOperation(chunk, function(err) {
    callback(err); //
  });
};
```

The chunk argument is an instance of Buffer or a String. 1

customWriteOperation is your class's custom write operation. It can be asynchronous, so the callback can be safely called later. 2

The callback provided by Node's internal code is called with an error if one was generated. 3

A `_write` method supplies a callback ❶ that you can call when writing has finished. This allows `_write` to be asynchronous. This `customWriteOperation` method ❷ is simply used as an example here—in a real implementation it would perform the underlying I/O. This could involve talking to a database using sockets, or writing to a file. The first argument provided to the callback should be an error ❸, allowing `_write` to propagate errors if needed.

Node's `stream.Writable` base class doesn't need to know *how* the data was written, it just cares whether the operation succeeded or failed. Failures can be reported by passing an `Error` object to `callback`. This will cause an `error` event to be emitted. Remember that these `stream` base classes descend from `EventEmitter`, so you should usually add a listener to `error` to catch and gracefully handle any errors.

The next listing shows a complete implementation of a `stream.Writable` class.

Listing 5.8 An example implementation of a writable stream

```
var stream = require('stream');
GreenStream.prototype = Object.create(stream.Writable.prototype, {
  constructor: { value: GreenStream }
});

function GreenStream(options) {
  stream.Writable.call(this, options);
}

GreenStream.prototype._write = function(chunk, encoding, callback) {
  process.stdout.write('u001b[32m' + chunk + 'u001b[39m');
  callback();
};

process.stdin.pipe(new GreenStream());
```

Use the usual inheritance pattern to create a new writable stream class.

Decorate the chunk with the ANSI escape sequences for green text.

Call the callback once the text has been sent to stdout.

Pipe stdin through stdout to transform text into green text.

This short example changes input text into green text. It can be used by running it with `node writable.js`, or by piping text through it with `cat file.txt | node writable.js`.

Although this is a trivial example, it illustrates how easy it is to implement streamable classes, so you should consider doing this the next time you want to make something that stores data work with `pipe`.

Chunks and encodings

The `encoding` argument to write is only relevant when strings are being used instead of buffers. Strings can be used by setting `decodeStrings` to `false` in the options that are passed when instantiating a writable stream.

Streams don't always deal with `Buffer` objects because some implementations have optimized handling for strings, so dealing directly with strings can be more efficient in certain cases.

TECHNIQUE 33 Transmitting and receiving data with duplex streams

Duplex streams allow data to be transmitted and received. This technique shows you how to create your own duplex streams.

Problem

You want to create a streamable interface to an I/O source that needs to be both readable *and* writable.

Solution

Inherit from `stream.Duplex` and implement `_read` and `_write` methods.

Discussion

Duplex streams are a combination of the `Writable` and `Readable` streams, which are explained in techniques 31 and 32. As such, Duplex streams require inheriting from `stream.Duplex` and implementations for the `_read` and `_write` methods. Refer to technique 30 for an explanation of how to inherit from the stream base classes.

Listing 5.9 shows a small `stream.Duplex` class that reads and writes data from `stdin` and `stdout`. It prompts for data and then writes it back out with ANSI escape codes for colors.

Listing 5.9 A duplex stream

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

HungryStream.prototype = Object.create(stream.Duplex.prototype, {
  constructor: { value: HungryStream }
});

function HungryStream(options) {
  stream.Duplex.call(this, options);
  this.waiting = false;
}

HungryStream.prototype._write = function(chunk, encoding, callback) {
  this.waiting = false;
  this.push('u001b[32m' + chunk + 'u001b[39m');
  callback();
};

HungryStream.prototype._read = function(size) {
  if (!this.waiting) {
    this.push('Feed me data! > ');
    this.waiting = true;
  }
};

var hungryStream = new HungryStream();
process.stdin.pipe(hungryStream).pipe(process.stdout);
```

1. This property tracks if the prompt is being displayed.
2. This _write implementation pushes data onto the internal queue and then calls the supplied callback.
3. Display a prompt when waiting for data.
4. Pipe the standard input through the duplex stream, and then back out to standard output.

The `HungryStream` class in listing 5.9 will display a prompt, wait for input, and then return the input with ANSI color codes. To track the state of the prompt, an internal property called `waiting` (1) is used. The `_write` method, which will be called by Node automatically, sets the `waiting` property to `false`, indicating that input has been received, and then the data is pushed to the internal buffer with color codes attached. Finally, the callback that gets automatically passed to `_write` is executed (2).

When the class is waiting for data, the `_read` method pushes a message that acts as the prompt (3). This can be made interactive by piping the standard input stream through an instance of `HungryStream` and then back out through the standard output stream (4).

The great thing about duplex streams is they can sit in the middle of pipes. A simpler way to do this is to use the `stream.PassThrough` base class, which only relays data, allowing you to plug into the middle of a pipe and track data as it flows through it. The diagram in figure 5.4 shows how chunks of data flow through the duplex stream object, from the input to the output stream.

Several `stream.Duplex` implementations in the wild implement a `_write` method but keep the `_read` method as a blank stub. This is purely to take advantage of duplex streams as something that can enhance the behavior of other streams through pipes. For example, `hiccup` by Naomi Kyoto (https://github.com/naomik/hiccup) can be used to simulate slow or sporadic behavior of underlying I/O sources. This novel use of streams comes in handy when you're writing automated tests.

Duplex streams are useful for piping readable streams to writable streams and analyzing the data. Transform streams are specifically designed for changing data; the next technique introduces `stream.Transform` and the `_transform` method.

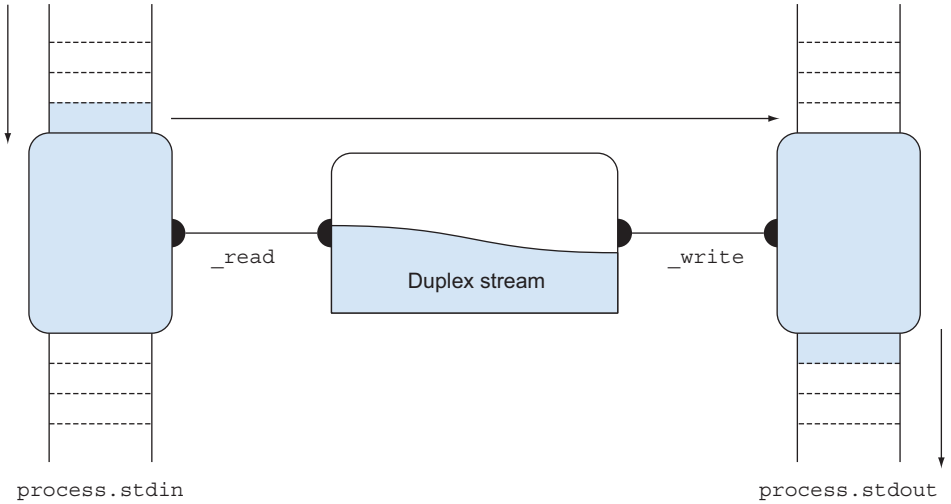


Figure 5.4 A duplex stream

TECHNIQUE 34 Parsing data with transform streams

Streams have long been used as a way to create efficient parsers. The `stream.Transform` base class can be used to do this in Node.

Problem

You want to use streams to change data into another format in a memory-efficient manner.

Solution

Inherit from `stream.Transform` and implement the `_transform` method.

Discussion

On the surface, transform streams sound a little bit like duplex streams. They can also sit in the middle of a pipe chain. The difference is that they're expected to transform data, and they're implemented by writing a `_transform` method. This method's signature is similar to `_write`—it takes three arguments, `chunk`, `encoding`, and `callback`. The callback should be executed when the data has been transformed, which allows transform streams to parse data asynchronously.

Listing 5.10 shows a transform stream that parses (albeit simplified) CSV data. The CSV is expected to contain comma-separated values without extra spaces or quotes, and should use Unix line endings.

Listing 5.10 A CSV parser implemented using a transform stream

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

CSVParser.prototype = Object.create(stream.Transform.prototype, {
  constructor: { value: CSVParser }
});

function CSVParser(options) {
  stream.Transform.call(this, options);

  this.value = '';
  this.headers = [];
  this.values = [];
  this.line = 0;
}

CSVParser.prototype._transform = function(chunk, encoding, done) {
  var c;
  var i;

  chunk = chunk.toString();

  for (i = 0; i < chunk.length; i++) {
    c = chunk.charAt(i);

    if (c === ',') {
      this.addValue();
```

1 These properties are used to track the state of the parser.

2 The _transform implementation.

3 The input data is turned into a string and then iterated over, character by character.

4 If the character is a comma, add the previously collected data to the internal list of headers or values.

```
    } else if (c === 'n') {
      this.addValue();
      if (this.line > 0) {
        this.push(JSON.stringify(this.toObject()));
      }
      this.values = [];
      this.line++;
    } else {
      this.value += c;
    }
  }

  done();
};

CSVParser.prototype.toObject = function() {
  var i;
  var obj = {};
  for (i = 0; i < this.headers.length; i++) {
    obj[this.headers[i]] = this.values[i];
  }
  return obj;
};

CSVParser.prototype.addValue = function() {
  if (this.line === 0) {
    this.headers.push(this.value);
  } else {
    this.values.push(this.value);
  }
  this.value = '';
};

var parser = new CSVParser();
fs.createReadStream(__dirname + '/sample.csv')
  .pipe(parser)
  .pipe(process.stdout);
```

5 If the character is a line ending, record the previously collected header or field, and then use push to send a JSON version of the data fields to the internal queue.

6 When processing has finished, call the callback provided by Node.

7 Convert the internal array of headers and the most recent line of fields into an object that can be converted to JSON.

8 Headers are assumed to be on the first line; otherwise the most recently collected data is assumed to be a data value.

Parsing CSV involves tracking several variables—the current value, the headers for the file, and the current line number **1**. To do this, a `stream.Transform` descendent with suitable properties can be used. The `_transform` implementation **2** is the most complex part of this example. It receives a chunk of data, which is iterated over one character at a time using a `for` loop **3**. If the character is a comma, the current value is saved **4** (if there is one). If the current character is a newline, the line is transformed into a JSON representation **5**. This example is synchronous, so it's safe to execute the callback supplied to `_transform` at the end of the method **6**. A `toObject` method has been included to make it easier to change the internal representation of the headers and values into a JavaScript object **7**.

The last line in the example creates a readable file stream of CSV data and pipes it through the CSV parser, and that output is piped again back through `stdout` so the results can be viewed **8**. This could also be piped through a compression module to

directly support compressed CSV files, or anything else you can think of doing with `pipe` and streams.

This example doesn't implement all of the things real-world CSV files can contain, but it does show that building streaming parsers with `stream.Transform` isn't too complicated, depending on the file format or protocol.

Now that you've learned how to use the base classes, you're probably wondering what the `options` argument in listing 5.10 was used for. The next section includes some details on how to use options to optimize stream throughput, and details some more advanced techniques.

5.5 Advanced patterns and optimization

The stream base classes accept various options for tailoring their behavior, and some of these options can be used to tune performance. This section has techniques for optimizing streams, using the older streams API, adapting streams based on input, and testing streams.

TECHNIQUE 35 Optimizing streams

Built-in streams and the classes used to build custom streams allow the internal buffer size to be configured. It's useful to know how to optimize this value to attain the desired performance characteristics.

Problem

You want to read data from a file, but are concerned about either speed or memory performance.

Solution

Optimize the stream's buffer size to suit your application's requirements.

Discussion

The built-in stream functions take a buffer size parameter, which allows the performance characteristics to be tailored to a given application. The `fs.createReadStream` method takes an `options` argument that can include a `bufferSize` property. This option is passed to `stream.Readable`, so it'll control the internal buffer used to temporarily store file data before it's used elsewhere.

The stream created by `zlib.createGzip` is an instance of `streams.Transform`, and the `Zlib` class creates its own internal buffer object for storing data. Controlling the size of this buffer is also possible, but this time the options property is `chunkSize`. Node's documentation has a section on optimizing the memory usage of zlib,[1] based on the documentation in the zlib/zconf.h header file, which is part of the low-level source code used to implement zlib itself.

In practice it's quite difficult to push Node's streams to exhibit different CPU performance characteristics based on buffer size. But to illustrate the concept, we've included a small benchmarking script that includes some interesting ideas about measuring stream performance. The next listing attempts to gather statistics on memory and elapsed time.

[1] See "Memory Usage Tuning"—http://nodejs.org/docs/latest/api/all.html#all_process_memoryusage.

Listing 5.11 Benchmarking streams

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

function benchStream(inSize, outSize) {
  var time = process.hrtime();
  var watermark = process.memoryUsage().rss;
  var input = fs.createReadStream('/usr/share/dict/words', {
    bufferSize: inSize
  });
  var gzip = zlib.createGzip({ chunkSize: outSize });
  var output = fs.createWriteStream('out.gz', { bufferSize: inSize });

  var memoryCheck = setInterval(function() {
    var rss = process.memoryUsage().rss;

    if (rss > watermark) {
      watermark = rss;
    }
  }, 50);

  input.on('end', function() {
    var memoryEnd = process.memoryUsage();
    clearInterval(memoryCheck);

    var diff = process.hrtime(time);
    console.log([
      inSize,
      outSize,
      (diff[0] * 1e9 + diff[1]) / 1000000,
      watermark / 1024].join(', ')
    );
  });

  input.pipe(gzip).pipe(output);

  return input;
}

console.log('file size, gzip size, ms, RSS');

var fileSize = 128;
var zipSize = 5024;

function run(times) {
  benchStream(fileSize, zipSize).on('end', function() {
    times--;
    fileSize *= 2;
    zipSize *= 2;

    if (times > 0) {
      run(times);
    }
  });
```

1. hrtime is used to get precise nanosecond measurements of the current time.
2. A timer callback is used to periodically check on memory usage and record the highest usage for the current benchmark.
3. When the input has ended, gather the statistics.
4. Log the results of the memory usage and time, converting the nanosecond measurements into milliseconds.
5. Stream the input file through the gzip instance and back out to a file.
6. A callback that will run when each benchmark finishes.
7. Recursively call the benchmark function.

```
}

run(10); 8((callout-streams-buffer-size-8))
```

8 Initially call the benchmark function with the number of times we want it to run.

This is a long example, but it just uses some of Node's built-in functionality to gather memory statistics over time for streams designed to use different buffer sizes. The `benchStream` function performs most of the work and is executed several times. It records the current time using `hrtime` 1, which returns more precise measurements than `Date.now()` would. The input stream is the Unix dictionary file, which is piped through a gzip stream and then out to a file 5. Then `benchStream` uses `setInterval` to run a periodic check on the memory usage 2. When the input stream ends 3, the memory usage is calculated based on the values before and after the input file was gzipped.

The `run` function doubles the input file's buffer and gzip buffer 6 to show the impact on memory and the time taken to read the streams over time. When the reading of the input file completes, the memory usage and elapsed time will be printed 4. The input file is returned by the `benchStream` function so `run` can easily be called when benchmarking has finished. The `run` function will be called repeatedly 7, depending on the first argument passed to it 8.

Note that `process.hrtime` has been used to accurately benchmark the elapsed time. This method can be used for benchmarking because it's precise, and also accepts a `time` argument for automatically calculating the elapsed time.

I (Alex) ran this program with a 20 MB file to try to generate more interesting results than /usr/share/dict/words, and I've included a graph of the results in figure 5.5.

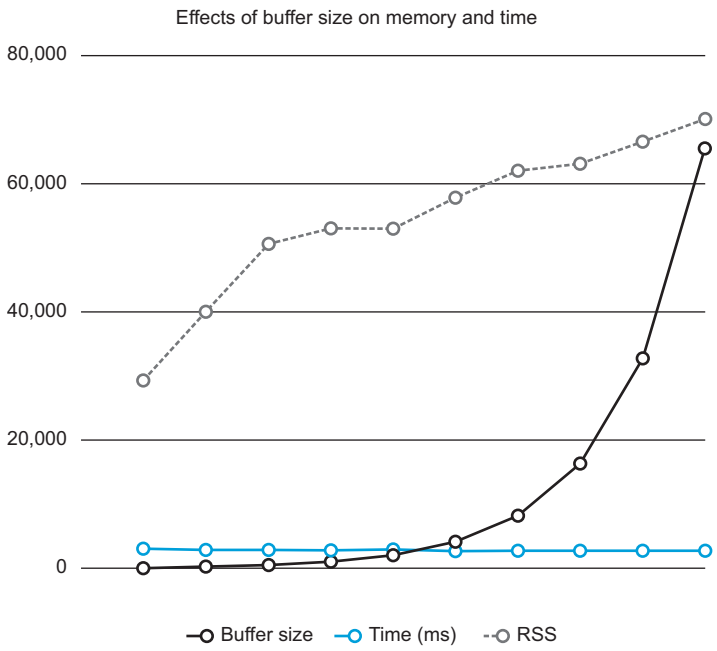


Figure 5.5 A graphical representation of the memory usage of streams

I found when I experimented with various files that the results indicated that elapsed time was far less affected than the memory usage. That indicates that it's generally desirable to use smaller buffers and be more conservative about memory usage, although this test should be repeated with a load-testing benchmark to really see how long it takes Node to process those buffers.

Node had an older API for streams that had different semantics for pausing a stream. Although the newer API should be used where possible, it's possible to use the older API alongside the newer one. The next technique demonstrates how to use modules written with the older API.

TECHNIQUE 36 Using the old streams API

Before Node 0.10 (and technically 0.9.4), streams had a different API. Code written using that API can be used with the newer APIs by wrapping it to behave like the newer `stream.Readable` class.

Problem

You want to use a module that implements the old-style streaming API with classes that use the newer APIs.

Solution

Use `Readable.prototype.wrap`.

Discussion

The older stream API had readable and writable streams, but pausing a stream was "advisory" only. This led to a different API design that wasn't based around the newer streams2 classes. As people gradually realized how useful streamable classes are, a wealth of modules appeared on npm. Although the newer API solves key problems with the older design, there are still useful modules that haven't been updated.

Fortunately, older classes can be wrapped using the `Readable.prototype.wrap` method provided by the `stream` module. It literally wraps the older interface to make it behave like the newer `stream.Readable` class—it effectively creates a `Readable` instance that uses the older class as its data source.

Listing 5.12 shows an example of a stream implemented with the older API that has been wrapped with the newer `Readable` class.

Listing 5.12 An old-style stream that has been wrapped

```
var stream = require('stream');
var Readable = stream.Readable;
var util = require('util');

util.inherits(MemoryStream, stream);

function MemoryStream(interval) {
  this.readable = true;

  setInterval(function() {
    var data = process.memoryUsage();
    data.date = new Date();
    this.emit('data', JSON.stringify(data) + 'n');
```

1 **The older API required that classes inherited from the stream module and set the readable property to true.**

2 **The data event is emitted with some example values. Make sure strings or Buffer instances are used.**

```
  }.bind(this), interval);
}

var memoryStream = new MemoryStream(250);
var wrappedStream = new Readable().wrap(memoryStream);

wrappedStream.pipe(process.stdout);
```

An instance of the original stream must be wrapped to become an instance of the newer class. 3

Here the new stream is piped to a writable stream that is compatible with the newer streams API. 4

The example in listing 5.12 presents a simple class that inherits from the Node 0.8 stream module. The `readable` property 1 is part of the old API, and signifies that this is a readable stream. Another indicator that this is a legacy stream is the `data` event 2. The newer `Readable.prototype.wrap` method 3 is what translates all of this to make it compatible with the streams2 API style. At the end, the wrapped stream is piped to a Node 0.10 stream 4.

Now you should be able to use older streams with the newer APIs!

Sometimes streams need to change their behavior depending on the type of input that has been provided. The next technique looks at ways of doing just that.

TECHNIQUE 37 Adapting streams based on their destination

Stream classes are typically designed to solve a specific problem, but there's also potential for customizing their behavior by detecting how the stream is being used.

Problem

You want to make a stream behave differently when it's piped to the TTY (the user's shell).

Solution

Bind a listener to the `pipe` event, and then use `stream.isTTY` to check if the stream is bound to a terminal.

Discussion

This technique is a specific example of adapting a stream's behavior to its environment, but the general approach could be adapted to other problems as well. Sometimes it's useful to detect whether a stream is writing output to a TTY or something else—perhaps a file—because different behavior in each is desirable. For example, when printing to a TTY, some commands will use ANSI colors, but this isn't usually advisable when writing files because strange characters would clutter the results.

Node makes detecting whether the current process is connected to a TTY simple—just use `process.stdout.isTTY` and `process.stdin.isTTY`. These are Boolean properties that are derived from OS-level bindings in Node's source (in lib/tty.js).

The strategy to use for adapting a stream's output is to create a new `stream.Writable` class and set an internal property based on `isTTY`. Then add a listener to the `pipe` event, which changes `isTTY` based on the newly piped stream that's passed as the first argument to the listener callback.

Listing 5.13 demonstrates this by using two classes. The first, `MemoryStream`, inherits from `stream.Readable` and generates data based on Node's memory usage. The

second, `OutputStream`, monitors the stream it's bound to so it can tell the readable stream about what kind of output it expects.

Listing 5.13 Using `isTTY` to adapt stream behavior

```
var stream = require('stream');
var util = require('util');

util.inherits(MemoryStream, stream.Readable);
util.inherits(OutputStream, stream.Writable);

function MemoryStream() {
  this.isTTY = process.stdout.isTTY;
  stream.Readable.call(this);
}

MemoryStream.prototype._read = function() {
  var text = JSON.stringify(process.memoryUsage()) + 'n';
  if (this.isTTY) {
    this.push('u001b[32m' + text + 'u001b[39m');
  } else {
    this.push(text);
  }
};

// A simple writable stream
function OutputStream() {
  stream.Writable.call(this);
  this.on('pipe', function(dest) {
    dest.isTTY = this.isTTY;
  }.bind(this));
}

OutputStream.prototype._write = function(chunk, encoding, cb) {
  util.print(chunk.toString());
  cb();
};

var memoryStream = new MemoryStream();

// Switch the desired output stream by commenting one of these lines:
//memoryStream.pipe(new OutputStream);
memoryStream.pipe(process.stdout);
```

1. **Set an internal flag to record what kind of output is expected.**
2. **Use ANSI colors when printing to a terminal.**
3. **When the writable stream is bound with a pipe, change the destination's isTTY state.**

Internally, Node uses `isTTY` to adapt the behavior of the `repl` module and the `readline` interface. The example in listing 5.13 tracks the state of `process.stdout.isTTY` 1 to determine what the original output stream was, and then copies that value to subsequent destinations 3. When the terminal is a TTY, colors are used 2; otherwise plain text is output instead.

Streams, like anything else, should be tested. The next technique presents a method for writing unit tests for your own stream classes.

TECHNIQUE 38 Testing streams

Just like anything else you write, it's strongly recommended that you test your streams. This technique explains how to use Node's built-in `assert` module to test a class that inherits from `stream.Readable`.

Problem

You've written your own stream class and you want to write a unit test for it.

Solution

Use some suitable sample data to drive your stream class, and then call `read()` or `write()` to gather the results and compare them to the expected output.

Discussion

The common pattern for testing streams, used in Node's source itself and by many open source developers, is to drive the stream being tested using sample data and then compare the end results against expected values.

The most difficult part of this can be coming up with suitable data to test. Sometimes it's easy to create a text file, or a *fixture* in testing nomenclature, that can be used to drive the stream by piping it. If you're testing a network-oriented stream, then you should consider using Node's `net` or `http` modules to create "mock" servers that generate suitable test data.

Listing 5.14 is a modified version of the CSV parser from technique 34; it has been turned into a module so we can easily test it. Listing 5.15 is the associated test that creates an instance of `CSVParser` and then pushes some values through it.

Listing 5.14 The `CSVParser` stream

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

module.exports = CSVParser;

CSVParser.prototype = Object.create(stream.Transform.prototype, {
  constructor: { value: CSVParser }
});

function CSVParser(options) {
  options = options || {};
  options.objectMode = true;
  stream.Transform.call(this, options);

  this.value = '';
  this.headers = [];
  this.values = [];
  this.line = 0;
}

CSVParser.prototype._transform = function(chunk, encoding, done) {
  var c;
  var i;
```

1 Export the class so it can be easily tested.

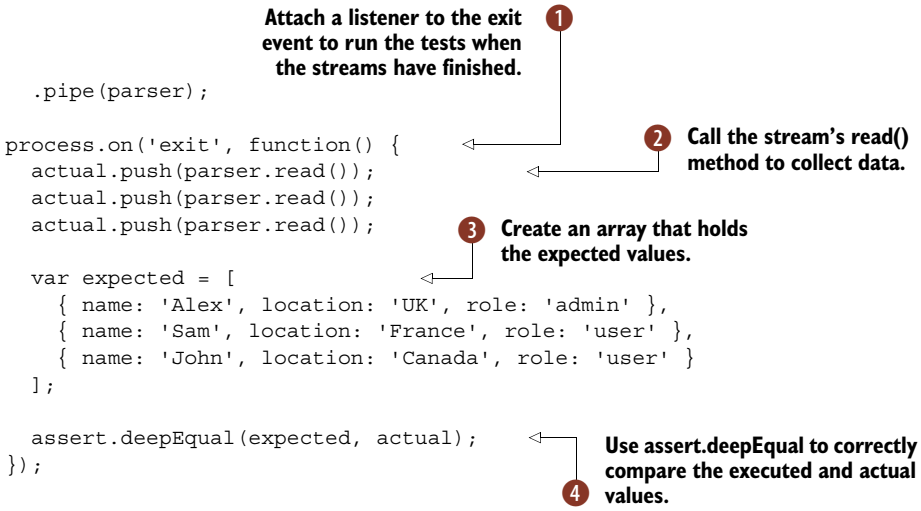
2 We can test this method by calling .push() on an instance of the class.

```
  chunk = chunk.toString();

  for (i = 0; i < chunk.length; i++) {
    c = chunk.charAt(i);

    if (c === ',') {
      this.addValue();
    } else if (c === 'n') {
      this.addValue();
      if (this.line > 0) {
        this.push(this.toObject());
      }
      this.values = [];
      this.line++;
    } else {
      this.value += c;
    }
  }

  done();
};

CSVParser.prototype.toObject = function() {
  var i;
  var obj = {};
  for (i = 0; i < this.headers.length; i++) {
    obj[this.headers[i]] = this.values[i];
  }
  return obj;
};

CSVParser.prototype.addValue = function() {
  if (this.line === 0) {
    this.headers.push(this.value);
  } else {
    this.values.push(this.value);
  }
  this.value = '';
};
```

The `CSVParser` class is exported using `module.exports` so it can be loaded by the unit test ❶. The `_transform` method ❷ will run later when `push` is called on an instance of this class. Next up is a simple unit test for this class.

Listing 5.15 Testing the `CSVParser` stream

```
var assert = require('assert');
var fs = require('fs');
var CSVParser = require('./csvparser');

var parser = new CSVParser();
var actual = [];

fs.createReadStream(__dirname + '/sample.csv')
```



A fixture file, sample.csv, has been used to pipe data to the `CSVParser` instance. Then the `assert.deepEqual` method has been used to make it easy to compare the expected array with the actual array.

A listener is attached to `exit` (1) because we want to wait for the streams to finish processing the data before running the assertion. Then data is `read` (2) from the parser and pushed to an array to examine with assertions (4)—the expected values are defined first (3). This pattern is used in Node's own streams tests, and is a lightweight version of what test frameworks like Mocha and node-tap provide.

5.6 Summary

In this chapter you've seen how the built-in streamable APIs work, how to create new and novel streams using the base classes provided by Node, and how to use some more advanced techniques to structure programs with streams. As you saw in technique 36, building new streams starts with correctly inheriting from the base classes—and don't forget to test those streams! For more on testing, refer back to technique 38.

As you saw, there are some novel uses of streams, like substack's `baudio` module (https://github.com/substack/baudio) that speaks in streams of sound waves. There are also *two* streams APIs: the original Node 0.8 and below API, and the newer streams2 API. Forward compatibility is supported through the `readable-stream` module (https://github.com/isaacs/readable-stream), and backward compatibility is made possible by wrapping streams (technique 36).

A big part of working with streams is handling files. In the next chapter we'll look at Node's file system handling in detail.

6 File system: Synchronous and asynchronous approaches to files

This chapter covers

- Understanding the `fs` module and its components
- Working with configuration files and file descriptors
- Using file-locking techniques
- Recursive file operations
- Writing a file database
- Watching files and directories

As we've noted in previous chapters, Node's core modules typically stick to a low-level API. This allows for various (even competing) ideas and implementations of higher-level concepts like web frameworks, file parsers, and command-line tools to exist as third-party modules. The `fs` (or file system) module is no different.

The `fs` module allows the developer to interact with the file system by providing

- POSIX file I/O primitives
- File streaming
- Bulk file I/O
- File watching

The `fs` module is unique compared with other I/O modules (like `net` and `http`) in that it has both asynchronous and synchronous APIs. That means that it provides a mechanism to perform blocking I/O. The reason the file system also has a synchronous API is largely because of the internal workings of Node itself, namely, the module system and the synchronous behavior of `require`.

The goal of this chapter is to show you a number of techniques, of varying complexity, to use when working with the file system module. We'll look at

- Asynchronous and synchronous approaches for loading configuration files
- Working with the file descriptors
- Advisory file-locking techniques
- Recursive file operations
- Writing a file database
- Watching for file and directory changes

But before we get to the techniques, let's first take a high-level view of all you can do with the file system API in order to capture the functionality and provide some insight into what tool may be the best for the job.

6.1 An overview of the fs module

The fs module includes wrappers for common POSIX file operations, as well as bulk, stream, and watching operations. It also has synchronous APIs for many of the operations. Let's take a high-level walk through the different components.

6.1.1 POSIX file I/O wrappers

At a bird's-eye view, the majority of methods in the file system API are wrappers around standard POSIX file I/O calls (http://mng.bz/7EKM). These methods will have a similar name. For example, the `readdir` call (http://linux.die.net/man/3/readdir) has an `fs.readdir` counterpart in Node:

```
var fs = require('fs');
fs.readdir('/path/to/dir', function (err, files) {
  console.log(files); // [ 'fileA', 'fileB', 'fileC', 'dirA', 'etc' ]
});
```

Table 6.1 shows a list of the supported POSIX file methods in Node, including a description of their functionality.

Table 6.1 Supported POSIX file methods in Node

POSIX method	`fs` method	Description
`rename(2)`	`fs.rename`	Changes the name of a file
`truncate(2)`	`fs.truncate`	Truncates or extends a file to a specified length
`ftruncate(2)`	`fs.ftruncate`	Same as `truncate` but takes a file descriptor
`chown(2)`	`fs.chown`	Changes file owner and group
`fchown(2)`	`fs.fchown`	Same as `chown` but takes a file descriptor
`lchown(2)`	`fs.lchown`	Same as `chown` but doesn't follow symbolic links
`chmod(2)`	`fs.chmod`	Changes file permissions
`fchmod(2)`	`fs.fchmod`	Same as `chmod` but takes a file descriptor
`lchmod(2)`	`fs.lchmod`	Same as `chmod` but doesn't follow symbolic links
`stat(2)`	`fs.stat`	Gets file status
`lstat(2)`	`fs.lstat`	Same as `stat` but returns information about link if provided rather than what the link points to
`fstat(2)`	`fs.fstat`	Same as `stat` but takes a file descriptor
`link(2)`	`fs.link`	Makes a hard file link
`symlink(2)`	`fs.symlink`	Makes a symbolic link to a file
`readlink(2)`	`fs.readlink`	Reads value of a symbolic link
`realpath(2)`	`fs.realpath`	Returns the canonicalized absolute pathname
`unlink(2)`	`fs.unlink`	Removes directory entry
`rmdir(2)`	`fs.rmdir`	Removes directory
`mkdir(2)`	`fs.mkdir`	Makes directory
`readdir(2)`	`fs.readdir`	Reads contents of a directory
`close(2)`	`fs.close`	Deletes a file descriptor
`open(2)`	`fs.open`	Opens or creates a file for reading or writing
`utimes(2)`	`fs.utimes`	Sets file access and modification times
`futimes(2)`	`fs.futimes`	Same as `utimes` but takes a file descriptor
`fsync(2)`	`fs.fsync`	Synchronizes file data with disk
`write(2)`	`fs.write`	Writes data to a file
`read(2)`	`fs.read`	Reads data from a file

The POSIX methods provide a low-level API to many common file operations. For example, here we use a number of synchronous POSIX methods to write data to a file and then retrieve that data:

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

var fd = fs.openSync('./file.txt', 'w+');
var writeBuf = new Buffer('some data to write');
fs.writeSync(fd, writeBuf, 0, writeBuf.length, 0);

var readBuf = new Buffer(writeBuf.length);
fs.readSync(fd, readBuf, 0, writeBuf.length, 0);
assert.equal(writeBuf.toString(), readBuf.toString());

fs.closeSync(fd);
```

Open or create file.txt for writing and reading (w+).

Create a buffer with data to write.

Write the buffer to the file.

Create an empty read buffer the same size as what was written.

Fill the read buffer with the data stored in the file.

Assert the written and read data are indeed the same.

Close the file.

When it comes to reading and writing files, typically you won't need a level this low, but rather can use a streaming or bulk approach.

6.1.2 Streaming

The `fs` module provides a streaming API with `fs.createReadStream` and `fs.createWriteStream`. `fs.createReadStream` is a `Readable` stream, whereas `fs.createWriteStream` is a `Writeable`. The streaming APIs can connect to other streams with `pipe`. For example, here's a simple application that copies a file using streams:

```
var fs = require('fs');
var readable = fs.createReadStream('./original.txt');
var writeable = fs.createWriteStream('./copy.txt');
readable.pipe(writeable);
```

Open original.txt to start reading.

Create or overwrite copy.txt with new data.

While reading in original.txt, write it out to copy.txt.

File streaming is beneficial when you want to deal with bits and pieces of data at a time or want to chain data sources together. For a more in-depth look at streams, check out chapter 5.

6.1.3 Bulk file I/O

The file system API also includes a few bulk methods for reading (`fs.readFile`), writing (`fs.writeFile`), or appending (`fs.appendFile`).

The bulk methods are good when you want to load a file into memory or write one out completely in one shot:

```
var fs = require('fs');
fs.readFile('/path/to/file', function (err, buf) {
  console.log(buf.toString());
});
```

Entire file is buffered and provided in the buf variable

6.1.4 File watching

The `fs` module also provides a couple of mechanisms for watching files (`fs.watch` and `fs.watchFile`). This is useful when you want to know if a file has changed in some way. `fs.watch` uses the underlying operating system's notifications, making it very efficient. But `fs.watch` can be finicky or simply not work on network drives. For those situations, the less-efficient `fs.watchFile` method, which uses `stat` polling, can be used.

We'll look more at file watching later on in this chapter.

6.1.5 Synchronous alternatives

Node's synchronous file system API sticks out like a sore thumb. With a big `Sync` tacked onto the end of each synchronous method, it's hard to miss its purpose. Synchronous methods are available for all the POSIX and bulk API calls. Some examples include `readFileSync`, `statSync`, and `readdirSync`. `Sync` tells you that this method will block your single-threaded Node process until it has finished. As a general rule, synchronous methods should be used when first setting up your application, and not within a callback:

```
var fs = require('fs');
var http = require('http');
fs.readFileSync('./output.dat');

http.createServer(function (req, res) {
  fs.readFileSync('./output.dat');
}).listen(3000);
```

A good spot for a synchronous method, top level, gets called on initialization of the application

A bad spot for a synchronous method, halts the server until the file is read on every request

Of course there are exceptions to the rule, but what's important is understanding the performance implications of using synchronous methods.

Testing server performance

How do we know synchronous execution within the request handling of a web server is slower? A great way to test this is using ApacheBench (http://en.wikipedia.org/wiki/ApacheBench). Our earlier example showed a ~2x drop in performance when serving a 10 MB file synchronously on every request rather than cached during application setup. Here's the command used in this test:

```
ab -n 1000 -c 100 "http://localhost:3000"
```

With our quick overview out of the way, we're now ready to get into some of the techniques you'll use when working with the file system.

TECHNIQUE 39 Loading configuration files

Keeping configuration in a separate file can be handy, especially for applications that run in multiple environments (like development, staging, and production). In this technique, you'll learn the ins and outs of how to load configuration files.

Problem

Your application stores configuration in a separate file and it depends on having that configuration when it starts up.

Solution

Use a synchronous file system method to pull in the configuration on initial setup of your application.

Discussion

A common use of synchronous APIs is for loading configuration or other data used in the application on startup. Let's say we have a simple configuration file stored as JSON that looks like the following:

```
{
  "site title": "My Site",
  "site base url": "http://mysite.com",
  "google maps key": "92asdfase8230232138asdfasd",
  "site aliases": [ "http://www.mysite.com", "http://mysite.net" ]
}
```

Let's first look at how we could do this asynchronously so you can see the difference. For example, say `doThisThing` depends on information from our configuration file. Asynchronously we could write it this way:

```
var fs = require('fs');
fs.readFile('./config.json', function (err, buf) {
  if (err) throw er;
  var config = JSON.parse(buf.toString());
  doThisThing(config);
})
```

Since the application can't run without this config file, we'll just throw the error so the Node process will exit with a stack trace.

We get a Buffer back, convert to a string, and then parse the JSON.

This will work and may be desirable for some setups, but will also have the effect of having everything that depends on the configuration nested in one level. This can get ugly. By using a synchronous version, we can handle things more succinctly:

```
var fs = require('fs');
var config = JSON.parse(fs.readFileSync('./config.json').toString());
doThisThing(config);
```

Synchronous methods will automatically throw if there's an error.

One of the characteristics of using `Sync` methods is that whenever an error occurs, it will be thrown:

```
var fs = require('fs');
try {
  fs.readFileSync('./some-file');
```

Synchronous errors can be caught using a standard try/catch block.

```
}
catch (err) {
  console.error(err);
}
```

Handle the error.

A note about require

We can `require` JSON files as modules in Node, so our code could even be shortened further:

```
var config = require('./config.json');
doThisThing(config);
```

But there's one caveat with this approach. Modules are cached globally in Node, so if we have another file that also requires `config.json` and we modify it, it's modified everywhere that module is used in our application. Therefore, using `readFileSync` is recommended when you want to tamper with the objects. If you choose to use `require` instead, treat the object as frozen (read-only); otherwise you can end up with hard-to-track bugs. You can explicitly freeze an object by using `Object.freeze`.

This is different from asynchronous methods, which use an error argument as the first parameter of the callback:

```
fs.readFile('./some-file', function (err, data) {
  if (err) {
    console.error(err);
  }
});
```

Asynchronous errors are handled as the first parameter in the callback function.

Handle the error.

In our example of loading a configuration file, we prefer to crash the application since it can't function without that file, but sometimes you may want to handle synchronous errors.

TECHNIQUE 40 Using file descriptors

Working with file descriptors can be confusing at first if you haven't dealt with them. This technique serves as an introduction and shows some examples of how you use them in Node.

Problem

You want to access a file descriptor to do writes or reads.

Solution

Use Node's `fs` file descriptor methods.

Discussion

File descriptors (FDs) are integers (indexes) associated with open files within a process managed by the operating system. As a process opens files, the operating system keeps track of these open files by assigning each a unique integer that it can then use to look up more information about the file.

Although it has *file* in the name, it covers more than just regular files. File descriptors can point to directories, pipes, network sockets, and regular files, to name a few.

Node can get at these low-level bits. Most processes have a standard set of file descriptors, as shown in table 6.2.

Table 6.2 Common file descriptors

Stream	File descriptor	Description
stdin	0	Standard input
stdout	1	Standard output
stderr	2	Standard error

In Node, we typically are used to the `console.log` sugar when we want to write to stdout:

```
console.log('Logging to stdout')
```

If we use the stream objects available on the `process` global, we can accomplish the same thing more explicitly:

```
process.stdout.write('Logging to stdout')
```

But there's another, far less used way to write to stdout using the `fs` module. The `fs` module contains a number of methods that take an FD as its first argument. We can write to file descriptor 1 (or stdout) using `fs.writeSync`:

```
fs.writeSync(1, 'Logging to stdout')
```

SYNCHRONOUS LOGGING `console.log` and `process.stdout.write` are actually synchronous methods under the hood, provided the TTY is a file stream

A file descriptor is returned from the `open` and `openSync` calls as a number:

```
var fd = fs.openSync('myfile','a');
console.log(typeof fd == 'number');     <-- Returns true
```

There are a variety of methods that deal with file descriptors specified in the file system documentation.

Typically more interesting uses of file descriptors happen when you're inheriting from a parent process or spawning a child process where descriptors are shared or passed. We'll discuss this more when we look at child processes in a later chapter.

TECHNIQUE 41 Working with file locking

File locking is helpful when cooperating processes need access to a common file where the integrity of the file is maintained and data isn't lost. In this technique, we'll explore how to write your own file locking module.

Problem

You want to lock a file to prevent processes from tampering with it.

Solution

Set up a file-locking mechanism using Node's built-ins.

Discussion

In a single-threaded Node process, file locking is typically something you won't need to worry about. But you may have situations where other processes are accessing the same file, or a cluster of Node processes are accessing the same file.

In these cases, there's the possibility that races and data loss may occur (more about this at http://mng.bz/yTLV). Most operating systems provide mandatory locks (those enforced at a kernel level) and advisory locks (not enforced; these only work if processes involved subscribe to the same locking scheme). Advisory locks are generally preferred if possible, as mandatory locks are heavy handed and may be difficult to unlock (https://kernel.org/doc/Documentation/filesystems/mandatory-locking.txt).

> **FILE LOCKING WITH THIRD-PARTY MODULES** Node has no built-in support for locking a file directly (either mandatory or advisory). But advisory locking of files can be done using syscalls such as `flock` (http://linux.die.net/man/2/flock), which is available in a third-party module (http://github.com/baudehlo/node-fs-ext).

Instead of locking a file directly with something like `flock`, you can use a *lockfile*. Lockfiles are ordinary files or directories whose *existence* indicates some other resource is currently in use and not to be tampered with. The creation of a lockfile needs to be atomic (no races) to avoid collisions. Being advisory, all the participating processes would have to play by the same rules agreed on when the lockfile is present. This is illustrated in figure 6.1.

Let's say we had a file called config.json that could potentially be updated by any number of processes at any time. To avoid data loss or corruption, a `config.lock` file could be created by the process making the updates and removed when the process is finished. Each process would agree to check for the existence of the lockfile before making any updates.

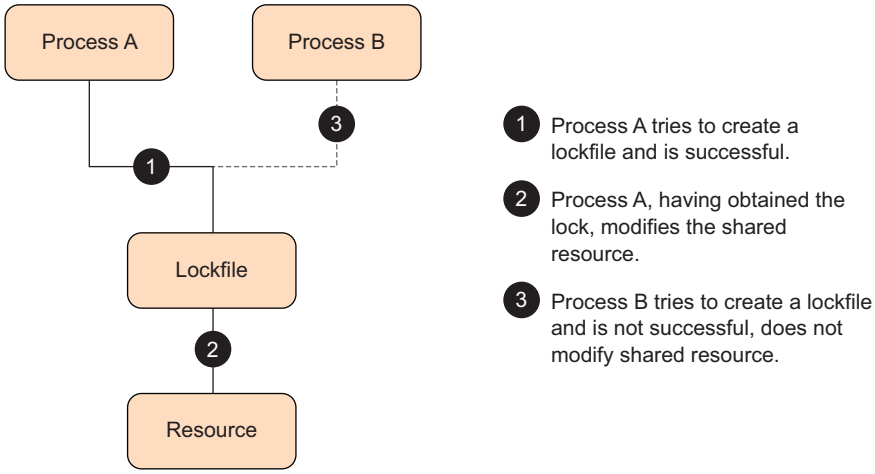


Figure 6.1 Advisory locking using a lockfile between cooperating processes

Node provides a few ways to perform this out of the box. We'll look at a couple of options:

- Creating a lockfile using the exclusive flag
- Creating a lockfile using `mkdir`

Let's look at using the exclusive flag first.

Creating lockfiles using the exclusive flag

The `fs` module provides an `x` flag for any methods that involve opening a file (like `fs.writeFile`, `fs.createWriteStream`, and `fs.open`). This flag tells the operating system the file should be opened in an exclusive mode (`O_EXCL`). When used, the file will fail to open if it already exists:

```
fs.open('config.lock', 'wx', function (err) {
   if (err) return console.error(err);

});
```

Open in an exclusive write mode.

Any failure including if file exists.

Safely modify config.json.

FLAG COMBINATIONS WHEN OPENING FILES There are a variety of flag combinations you can pass when opening files; for a list of all of them consult the `fs.open` documentation: http://nodejs.org/api/fs.html#fs_fs_open_path_flags_mode_callback.

We want to fail if another process has already created a lockfile. We fail because we don't want to tamper with the resource behind the lockfile while another process is using it. Thus, having the exclusive flag mechanism turns out to be useful in our case. But instead of writing an empty file, it's a good idea to throw the PID (process ID) inside of this file so if something bad happens, we'll know what process had the lock last:

Any failure including if file exists.

```
fs.writeFile('config.lock', process.pid, { flags: 'wx' },
  function (err) {
    if (err) return console.error(err);

  });
```

Write PID to lockfile if it doesn't exist.

Safely modify config.json.

Creating lockfiles with mkdir

Exclusive mode may not work well if the lockfile exists on a network drive, since some systems don't honor the `O_EXCL` flag on network drives. To circumvent this, another strategy is creating a lockfile as a directory. `mkdir` is an atomic operation (no races), has excellent cross-platform support, and works well with network drives. `mkdir` will fail if a directory exists. In this case, the PID could be stored as a file inside of that directory:

```
fs.mkdir('config.lock', function (err) {
  if (err) return console.error(err);
  fs.writeFile('config.lock/'+process.pid, function (err) {
    if (err) return console.error(err);

  });
});
```

Unable to create directory.

Indicate which PID has the lock for debugging.

Safely modify config.json.

Making a lockfile module

So far we've discussed a couple ways to create lockfiles. We also need a mechanism to remove them when we're done. In addition, to be good lockfile citizens, we should remove any lockfiles created whenever our process exits. A lot of this functionality can be wrapped up in a simple module:

```
var fs = require('fs');
var hasLock = false;
var lockDir = 'config.lock';

exports.lock = function (cb) {
  if (hasLock) return cb();

  fs.mkdir(lockDir, function (err) {
    if (err) return cb(err);
    fs.writeFile(lockDir+'/'+process.pid, function (err) {
      if (err) console.error(err);
      hasLock = true;
      return cb();
    });
  });
}

exports.unlock = function (cb) {
  if (!hasLock) return cb();

  fs.unlink(lockDir+'/'+process.pid, function (err) {
    if (err) return cb(err);
    fs.rmdir(lockDir, function (err) {
      if (err) return cb(err);
      hasLock = false;
      cb();
    });
  });
}

process.on('exit', function () {
  if (hasLock) {
    fs.unlinkSync(lockDir+'/'+process.pid);
    fs.rmdirSync(lockDir);
    console.log('removed lock');
  }
});
```

A lock is already obtained.

Define a method for obtaining a lock.

Write PID in directory for debugging.

Unable to create a lock.

Lock created.

If unable to write PID, not the end of the world: log and keep going.

No lock to unlock.

Define a method for releasing a lock.

If we still have a lock, remove it synchronously before exit.

Here's an example usage:

```
var locker = require('./locker');

locker.lock(function (err) {
  if (err) throw err;

  // Do modifications here.

  locker.unlock(function () {  });
})
```

Try to attain a lock.

Release lock when finished.

For a more full-featured implementation using exclusive mode, check out the lockfile third-party module (https://github.com/isaacs/lockfile).

TECHNIQUE 42 Recursive file operations

Ever need to remove a directory and all subdirectories (akin to `rm -rf`)? Create a directory and any intermediate directories given a path? Search a directory tree for a particular file? Recursive file operations are helpful and hard to get right, especially when done asynchronously. But understanding how to perform them is a good exercise in mastering evented programming with Node. In this technique, we'll dive into recursive file operations by creating a module for searching a directory tree.

Problem

You want to search for a file within a directory tree.

Solution

Use recursion and combine file system primitives.

Discussion

When a task spans multiple directories, things become more interesting, especially in an asynchronous world. You can mimic the command-line functionality of `mkdir` with a single call to `fs.mkdir`, but for fancier things like `mkdir -p` (helpful for creating intermediate directories), you have to think recursively. This means the solution to our problem will depend on "solutions to smaller instances of the same problem" ("Recursion (computer science)": http://en.wikipedia.org/wiki/Recursion_(computer_science)).

In our example we'll write a finder module. Our finder module will recursively look for matching files at a given start path (akin to `find /start/path -name='file-in-question'`) and provide the paths to those files in an array.

Let's say we had the following directory tree:

```
root
├── dir-a
│   ├── dir-b
│   │   ├── dir-c
│   │   │   └── file-e.png
│   │   ├── file-c.js
│   │   └── file-d.txt
│   ├── file-a.js
│   └── file-b.txt
```

A search for the pattern `/file.*/` from the root would give us the following:

```
[ 'dir-a/dir-b/dir-c/file-e.png',
  'dir-a/dir-b/file-c.js',
  'dir-a/dir-b/file-d.txt',
  'dir-a/file-a.js',
  'dir-a/file-b.txt' ]
```

So how do we build this? To start, the `fs` module gives us some primitives we'll need:

- `fs.readdir/fs.readdirSync`—List all the files (including directories), given a path.
- `fs.stat/fs.statSync`—Give us information about a file at the specified path, including whether the path is a directory.

Our module will expose synchronous (`findSync`) and asynchronous (`find`) implementations. `findSync` will block execution like other `Sync` methods, will be slightly faster than its asynchronous counterpart, and may fail on excessively large directory trees (since JavaScript doesn't have proper tail calls yet: https://people.mozilla.org/~jorendorff/es6-draft.html#sec-tail-position-calls).

> **Why are synchronous functions slightly faster?**
>
> Synchronous functions aren't deferred until later, even though the asynchronous counterparts happen very quickly. Synchronous functions happen right away while you're already on the CPU and you're guaranteed to wait only exactly as long as necessary for the I/O to complete. But synchronous functions will block other things from happening during the wait period.

On the other hand, `find` will be slightly slower, but won't fail on large trees (since the stack is regularly cleared due to the calls being asynchronous). `find` won't block execution.

Let's take a look at the code for `findSync` first:

```
var fs = require('fs');
var join = require('path').join;

exports.findSync = function (nameRe, startPath) {
  var results = [];

  function finder (path) {
    var files = fs.readdirSync(path);

    for (var i = 0; i < files.length; i++) {
      var fpath = join(path,files[i]);
      var stats = fs.statSync(fpath);

      if (stats.isDirectory()) finder(fpath);
      if (stats.isFile() && nameRe.test(files[i])) results.push(fpath);
    }
  }

  finder(startPath);
  return results;
}
```

Takes a regular expression for the file we're searching for and a starting path.

Collection to store matches.

Read a list of files (including directories).

Get path to current file.

Get stats for current file

If it's a directory, call finder again with the new path.

If it's a file and matches search, add it to results.

Start initial file lookup.

Return results.

Since everything is synchronous, we can use `return` at the end to get all our results, as it'll never reach there until all the recursion has finished. The first error to occur would throw and could be caught, if desired, in a try/catch block. Let's look at a sample usage:

```
var finder = require('./finder');
try {
  var results = finder.findSync(/file.*/, '/path/to/root');
  console.log(results);
} catch (err) {
  console.error(err);
}
```

Success! List files found.

Oh no! Something bad happened; log error.

Let's switch now and take a look at how to tackle this problem asynchronously with the `find` implementation:

```
var fs = require('fs');
var join = require('path').join;

exports.find = function (nameRe, startPath, cb) {
  var results = [];
  var asyncOps = 0;
  var errored = false;

  function error (err) {
    if (!errored) cb(err);
    errored = true;
  }

  function finder (path) {
    asyncOps++;
    fs.readdir(path, function (err, files) {
      if (err) return error(err);

      files.forEach(function (file) {
        var fpath = join(path,file);

        asyncOps++;
        fs.stat(fpath, function (err, stats) {
          if (err) return error(err);

          if (stats.isDirectory()) finder(fpath);
          if (stats.isFile() && nameRe.test(file)) results.push(fpath);

          asyncOps--;
          if (asyncOps == 0) cb(null, results);
        })
      })

      asyncOps--;
      if (asyncOps == 0) cb(null, results);
    });
  }

  finder(startPath);
}
```

In order to know when we've completed our traversal, we'll need a counter.

Find now takes a third callback parameter.

In order to avoid multiple error calls if we're unsuccessful, we'll track when an error occurs.

Error handler to ensure callback is only called once if multiple errors.

Increment counter before each asynchronous operation.

Need a closure here so we don't lose our file reference later on.

Decrement counter after an asynchronous operation has completed.

If we're back to zero, we're done and had no errors, and can call back with the results.

We can't just `return` our results, like in the synchronous version; we need to call back with them when we know we're finished. To know that we're finished, we use a counter (`asyncOps`). We also have to be aware whenever we have callbacks to ensure we have a closure around any variables we expect to have around when any asynchronous call completes (this is why we switched from a standard `for` loop to a `forEach` call—more about this at http://mng.bz/rqEA).

Our counter (`asyncOps`) increments right before we do an asynchronous operation (like `fs.readdir` or `fs.stat`). The counter decrements in the callback for the asynchronous operation. Specifically it decrements *after* any other asynchronous calls have been made (otherwise we'll get back to `0` too soon). In a successful scenario, `asyncOps` will reach `0` when all the recursive asynchronous work has completed, and we can call back with the results (`if (asyncOps == 0) cb(null, results)`). In a failure scenario, `asyncOps` will never reach `0`, and one of the error handlers would've been triggered and have already called back with the error.

Also, in our example, we can't be sure that `fs.stat` will be the last thing to be called, since we may have a directory with no files in our chain, so we check at both spots. We also have a simple `error` wrapper to ensure we never call back with more than one error. If your asynchronous operation returns one value like in our example or one error, it's important to ensure you'll never call the callback more than once, as it leads to hard-to-track bugs later down the road.

> **ALTERNATIVES TO COUNTERS** The counter isn't the only mechanism that can track the completion of a set of asynchronous operations. Depending on the requirements of the application, recursively passing the original callback could work. For an example look at the third-party `mkdirp` module (https://github.com/substack/node-mkdirp).

Now we have an asynchronous version (`find`) and can handle the result of that operation with the standard Node-style callback signature:

```
var finder = require('./finder');
finder.find(/file*/, '/path/to/root', function (err, results) {
  if (err) return console.error(err);
  console.log(results);
});
```

> **THIRD-PARTY SOLUTIONS TO PARALLEL OPERATIONS** Parallel operations can be hard to keep track of, and can easily become bug-prone, so you may want to use a third-party library like `async` (https://github.com/caolan/async) to help. Another alternative is using a promises library like `Q` (https://github.com/kriskowal/q).

TECHNIQUE 43 Writing a file database

Node's core `fs` module gives you the tools to build complexity like the recursive operations you saw in the last technique. It also enables you to do other complex tasks such

as creating a file database. In this technique we'll write a file database in order to look at other pieces in the `fs` module, including streaming, working together.

Problem

You want a simple and fast data storage structure with some consistency guarantees.

Solution

Use an in-memory database with append-only journaling.

Discussion

We'll write a simple key/value database module. The database will provide in-memory access to the current state for speed and use an append-only storage format on disk for persistence. Using append-only storage will provide us the following:

- *Efficient disk I/O performance*—We're always writing to the end of the file.
- *Durability*—The previous state of the file is never changed in any way.
- *A simple way to create backups*—We can just copy the file at any point to get the state of the database at that point.

Each line in the file is a record. The record is simply a JSON object with two properties, a `key` and a `value`. A `key` is a string representing a lookup for the `value`. The `value` can be anything JSON-serializable, which includes strings and numbers. Let's look at some sample records:

```
{"key":"a","value":23}
{"key":"b","value":["a","list","of","things"]}
{"key":"c","value":{"an":"object"}}
{"key":"d","value":"a string"}
```

If a record is updated, a new version of the record will be found later in the file with the same key:

```
{"key":"d","value":"an updated string"}
```

If a record has been removed, it'll also be found later in the file with a `null` value:

```
{"key":"b","value":null}
```

When the database is loaded, the journal will be streamed in from top to bottom, building the current state of the database in memory. Remember, data isn't deleted, so it's possible to store the following data:

```
{"key":"c","value":"my first value"}
...
{"key":"c","value":null}
...
{"key":"c","value":{"my":"object"}}
```

In this case, at some point we saved `"my first value"` as the key `c`. Later on we deleted the key. Then, most recently, we set the key to be `{"my":"object"}`. The most recent entry will be loaded in memory, as it represents the current state of the database.

We talked about how data will be persisted to the file system. Let's talk about the API we'll expose next:

```
var Database = require('./database');
var client = new Database('./test.db');

client.on('load', function () {
  var foo = client.get('foo');

  client.set('bar', 'my sweet value', function (err) {
    if (err) return console.error(err);
    console.log('write successful');
  });

  client.del('baz');
});
```

Load our Database module.

Provide the path to the database file we want to load and/or create.

Load is triggered when data is loaded into memory.

Get the value stored at key foo.

Set a value for key bar.

An error occurred when persisting to disk.

Delete key baz; optionally take afterWrite callback.

Let's dive into the code to start putting this together. We'll write a `Database` module to store our logic. It'll inherit from `EventEmitter` so we can emit events back to the consumer (like when the database has loaded all its data and we can start using it):

```
var fs = require('fs')
var EventEmitter = require('events').EventEmitter

var Database = function (path) {
  this.path = path

  this._records = Object.create(null)
  this._writeStream = fs.createWriteStream(this.path, {
    encoding: 'utf8',
    flags: 'a'
  })

  this._load()
}

Database.prototype = Object.create(EventEmitter.prototype)
```

Set the path to the database storage.

Create an internal mapping of all the records in memory.

Create a write stream in append-only mode to handle writes to disk.

Load the database.

Inherit from EventEmitter.

We want to stream the data stored and emit a "load" event when that's completed. Streaming will enable us to handle data as it's being read in. Streaming also is asynchronous, allowing the host application to do other things while the data is being loaded:

```
Database.prototype._load = function () {
  var stream = fs.createReadStream(this.path, { encoding: 'utf8' });
  var database = this;

  var data = '';
  stream.on('readable', function () {
    data += stream.read();
    var records = data.split('\n');
    data = records.pop();
```

Split records on newlines.

Read the available data.

Pop the last record as it may be incomplete.

```
    for (var i = 0; i < records.length; i++) {
      try {
        var record = JSON.parse(records[i]);
        if (record.value == null)
          delete database._records[record.key];
        else
          database._records[record.key] = record.value;
      } catch (e) {
          database.emit('error', 'found invalid record:', records[i]);
      }
    }
  });
  stream.on('end', function () {
    database.emit('load');
  });
}
```

If the record has a null value, delete the record if stored.

Otherwise, for all non-null values, store the value for that key.

Emit an error if an invalid record was found.

Emit a load event when data is ready to be used.

As we read in data from the file, we find all the complete records that exist.

STRUCTURING OUR WRITES TO STRUCTURE OUR READS What do we do with the data we just `pop()`ed the last time a `readable` event is triggered? The last record turns out to always be an empty string (`''`) because we end each line with a newline (`\n`) character.

Once we've loaded the data and emitted the `load` event, a client can start interacting with the data. Let's look at those methods next, starting with the simplest—the `get` method:

```
Database.prototype.get = function (key) {
  return this._records[key] || null;
}
```

Return value for key or null if no key exists.

Let's look at storing updates next:

```
Database.prototype.set = function (key, value, cb) {
  var toWrite = JSON.stringify({ key: key, value: value }); + '\n'

  if (value == null)
    delete this._records[key];
  else
    this._records[key] = value;

  this._writeStream.write(toWrite, cb);
}
```

Stringify JSON storage object, and then add newline.

If deleting, remove record from in-memory storage.

Otherwise, set key to value in memory.

Write out record to disk with callback if provided.

Now we add some sugar for deleting a key:

```
Database.prototype.del = function (key, cb) {
  return this.set(key, null, cb);
}
```

Call set for key with null as its value (storing a delete record).

There we have a simple database module. Last thing: we need to export the constructor:

```
module.exports = Database;
```

There are various improvements that could be made on this module, like flushing writes (http://mng.bz/2g19) or retrying on failure. For examples of more full-featured Node-based database modules, check out `node-dirty` (https://github.com/felixge/node-dirty) or `nstore` (https://github.com/creationix/nstore).

TECHNIQUE 44 Watching files and directories

Ever need to process a file when a client adds one to a directory (through FTP, for instance) or reload a web server after a file is modified? You can do both by watching for file changes.

Node has *two* implementations for file watching. We'll talk about both in this technique in order to understand when to use one or the other. But at the core, they enable the same thing: watching files (and directories).

Problem

You want to watch a file or directory and perform an action when a change is made.

Solution

Use `fs.watch` and `fs.watchFile`.

Discussion

It's rare to see multiple implementations for the same purpose in Node core. Node's documentation recommends that you prefer `fs.watch` over `fs.watchFile` if possible, as it's considered more reliable. But `fs.watch` isn't consistent across operating systems, whereas `fs.watchFile` is. Why the madness?

The story about fs.watch

Node's event loop taps into the operating system in order to juggle asynchronous I/O in its single-threaded environment. This also provides a performance benefit, as the OS can let the process know immediately when some new piece of I/O is ready to be handled. Operating systems have different ways of notifying a process about events (that's why we have `libuv`). The culmination of that work for file watching is the `fs.watch` method.

`fs.watch` combines all these different types of event systems into one method with a common API to provide the following:

- A more reliable implementation in terms of file change events always getting fired
- A faster implementation, as notifications get passed to Node immediately when they occur

Let's look at the older method next.

The story about fs.watchFile

There's another, older implementation of file watching called `fs.watchFile`. It doesn't tap into the notification system but instead polls on an interval to see if changes have occurred.

`fs.watchFile` isn't as full-fledged in the changes it can detect, nor as fast. But the advantage of using `fs.watchFile` is that it's *consistent* across platforms and it works more reliably on network file systems (like SMB and NFS).

Which one is right for me?

The preferred is `fs.watch`, but since it's inconsistent across platforms, it's a good idea to test whether it does what you want (and better to have a test suite).

Let's write a program to help us play around file watching and see what each API provides. First, create a file called watcher.js with the following contents:

```
var fs = require('fs');
fs.watch('./watchdir', console.log);
fs.watchFile('./watchdir', console.log);
```

Now create a directory called `watchdir` in the same directory as your watcher.js file:

```
mkdir watchdir
```

Then, open a couple terminals. In the first terminal, run

```
node watcher
```

and in the second terminal, change to `watchdir`:

```
cd watchdir
```

With your two terminals open (preferably side by side), we'll make changes in `watchdir` and see Node pick them up. Let's create a new file:

```
touch file.js
```

We can see the Node output:

```
rename file.js
change file.js
{ dev: 64512,
   mode: 16893,
  nlink: 2,
  ... } { dev: 64512,
  mode: 16893,
  nlink: 2,
  ... }
```

A couple of fs.watch events come quickly (rename and change). These are the only two events fs.watch will emit. The second argument, file.js, is the file that received the event.

The fs.watchFile event comes later and has a different response. It includes two fs.Stats objects for the current and previous state of the file. They're the same here because the file was just created.

All right, so now we have a file created; let's update its modification time with the same command:

```
touch file.js
```

Now when we look at our Node output, we see that only `fs.watch` picked up this change:

```
change file.js
```

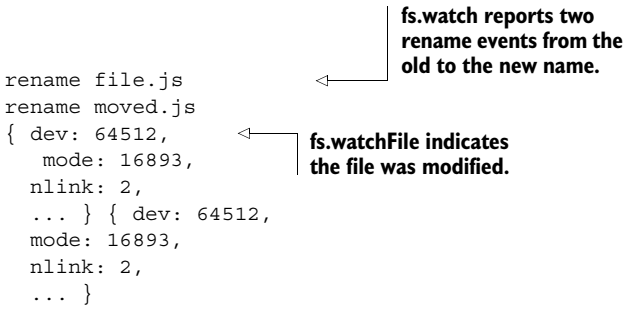
So if using `touch` to update a file when watching a directory is important to your application, `fs.watch` has support.

FS.WATCHFILE AND DIRECTORIES Many updates to files while watching a directory won't be picked up by `fs.watchFile`. If you want to get this behavior with `fs.watchFile`, watch the individual file.

Let's try moving our file:

```
mv file.js moved.js
```

In our Node terminal, we see the following output indicating both APIs picked up the change:



The main point here is to test the APIs using the exact use case you want to utilize. Hopefully, this API will get more stable in the future. Read the documentation to get the latest development (http://nodejs.org/api/fs.html#fs_fs_watch_filename_options_listener). Here are some tips to help navigate:

- Run your test case, preferring `fs.watch`. Are events getting triggered as you expect them to be?
- If you intend to watch a single file, don't watch the directory it's in; you may end up with more events being triggered.
- If comparing file stats is important between changes, `fs.watchFile` provides that out of the box. Otherwise, you'll need to manage stats manually using `fs.watch`.
- Just because `fs.watch` works on your Mac doesn't mean it will work exactly the same way on your Linux server. Ensure development and production environments are tested for the desired functionality.

Go forth and watch wisely!

6.2 Summary

In this chapter we talked through a number of techniques using the `fs` module. We covered asynchronous and synchronous usage while looking at configuration file loading and recursive file handling. We also looked at file descriptors and file locking. Lastly we implemented a file database.

Hopefully this has expanded your understanding of some of the concepts possible with using the `fs` module. Here are a few takeaways:

- Synchronous methods can be a nicer, simpler way to do things over their asynchronous counterparts, but beware of the performance issues, especially if you're writing a server.
- Advisory file locking is a helpful mechanism for resources shared across multiple processes as long as all processes follow the same contract.
- Parallel asynchronous operations that require some sort of response after completion need to be tracked. Although it's helpful to understand how to use counters or recursive techniques, consider using a well-tested third-party module like `async`.
- Look at how you'll use a particular file to determine which course of action to follow. If it's a large file or can be dealt with in chunks, consider using a streaming approach. If it's a smaller file or something you can't use until you have the entire file loaded, consider a bulk method. If you want to change a particular part of a file, you probably want to stick with the POSIX file methods.

In the next chapter we'll look at the other main form of I/O in Node: networking.

7 Networking: Node's true "Hello, World"

This chapter covers

- Networking concepts and how they relate to Node
- TCP, UDP, and HTTP clients and servers
- DNS
- Network encryption

The Node.js platform itself is billed as a solution for writing fast and scalable network applications. To write network-oriented software, you need to understand how networking technologies and protocols interrelate. Over the course of the next section, we explain how networks have been designed around technology stacks with clear boundaries; and furthermore, how Node implements these protocols and what their APIs look like.

In this chapter you'll learn about how Node's networking modules work. This includes the `dgram`, `dns`, `http`, and `net` modules. If you're unsure about network terminology like *socket*, *packet*, and *protocol*, then don't worry: we also introduce key networking concepts to give you a solid foundation in network programming.

7.1 Networking in Node

This section is an introduction to networking. You'll learn about network layers, packets, sockets—all of the stuff that networks are made of. These ideas are critical to understanding Node's networking APIs.

7.1.1 Networking terminology

Networking jargon can quickly become overwhelming. To get everyone on the same page, we've included table 7.1, which summarizes the main concepts that will form the basis of this chapter.

To understand Node's networking APIs, it's crucial to learn about layers, packets, sockets, and all the other things that networks are made of. If you don't learn about the difference between TCP (Transmission Control Protocol) and UDP (User Datagram Protocol), then it would be difficult for you to know when to use these protocols. In this section we introduce the terms you need to know and then explore the concepts a bit more so you leave the section with a solid foundation.

Table 7.1 Networking concepts

Term	Description
Layer	A slice of related networking protocols that represents a logical group. The application layer, where we work, is the highest level; physical is the lowest.
HTTP	Hypertext Transfer Protocol—An application-layer client-server protocol built on TCP.
TCP	Transmission Control Protocol—Allows communication in both directions from the client to the server, and is built on to create application-layer protocols like HTTP.
UDP	User Datagram Protocol—A lightweight protocol, typically chosen where speed is desired over reliability.
Socket	The combination of an IP address and a port number is generally referred to as a socket.
Packet	TCP packets are also known as *segments*—the combination of a chunk of data along with a header.
Datagram	The UDP equivalent of a packet.
MTU	Maximum Transmission Unit—The largest size of a protocol data unit. Each layer can have an MTU: IPv4 is at least 68 bytes, and Ethernet v2 is 1,500 bytes.

If you're responsible for implementing high-level protocols that run on top of HTTP or even low-latency game code that uses UDP, then you should understand each of these concepts. We break each of these concepts down into more detail over the next few sections.

LAYERS

The stack of protocols and standards that make up the internet and internet technology in general can be modeled as layers. The lowest layers represent physical media—Ethernet, Bluetooth, fiber optics—the world of pins, voltages, and network adapters.

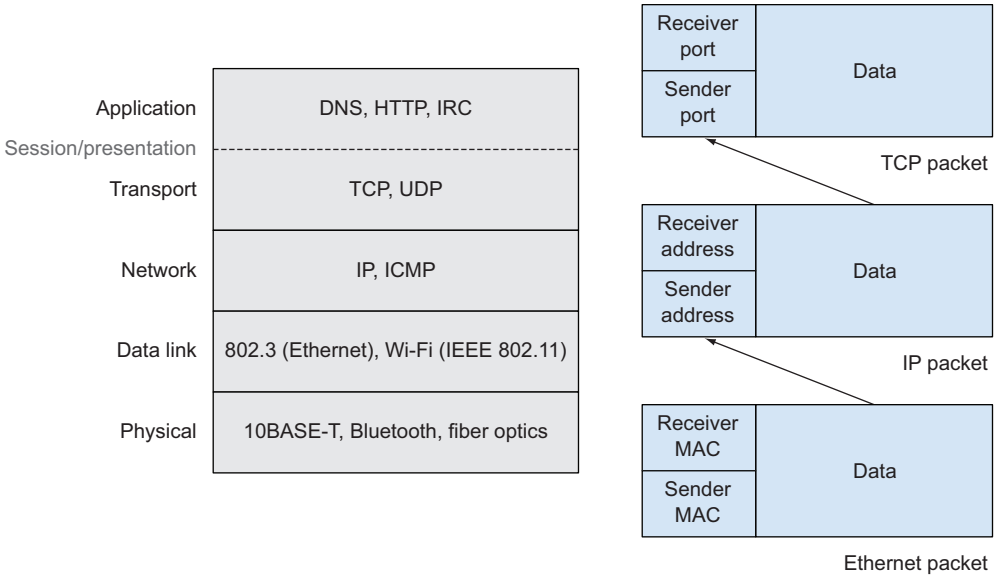


Figure 7.1 Protocols are grouped into seven logical layers. Packets are wrapped by protocols at consecutive layers.

As software developers, we work at a higher level than lower-level hardware. When talking to networks with Node, we're concerned with the *application* and *transport* layers of the Internet Protocol (IP) suite.

Layers are best represented visually. Figure 7.1 relates logical network layers to packets. The lower-level physical and data-link layer protocols wrap higher-level protocols.

Packets are wrapped by protocols at consecutive layers. A TCP packet, which could represent part of a series of packets from an HTTP request, is contained in the data section of an IP packet, which in turn is wrapped by an Ethernet packet. Going back to figure 7.1, TCP packets from HTTP requests cut through the transport and application layers: TCP is the transport layer, used to create the higher-level HTTP protocol. The other layers are also involved, but we don't always know which specific protocols are used at each layer: HTTP is always transmitted over TCP/IP, but beyond that, Wi-Fi or Ethernet can be used—your programs won't know the difference.

Figure 7.2 shows how network layers are wrapped by each protocol. Notice that data is never seen to move more than one step between layers—we don't talk about transport layer protocols interacting with the network layer.

When writing Node programs, you should appreciate that HTTP is implemented using TCP because Node's `http` module is built on the underlying TCP implementation found in the `net` module. But you don't need to understand how Ethernet, 10BASE-T, or Bluetooth works.

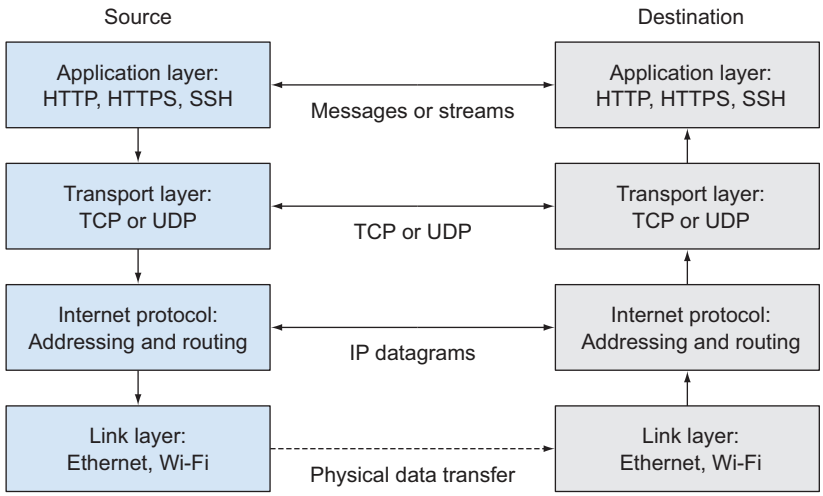


Figure 7.2 **Network layer wrapping**

TCP/IP

You've probably heard of TCP/IP—this is what we call the *Internet Protocol suite* because the Transmission Control Protocol (TCP) and the Internet Protocol (IP) are the most important and earliest protocols defined by this standard.

In Internet Protocol, a host is identified by an IP address. In IPv4, addresses are 32-bit, which limits the available address space. IP has been at the center of controversy over the last decade because addresses are running out. To fix this, a new version of the protocol known as IPv6 was developed.

You can make TCP connections with Node by using the `net` module. This allows you to implement application layer protocols that aren't supported by the core modules: IRC, POP, and even FTP could be implemented with Node's core modules. If you find yourself needing to talk to nonstandard TCP protocols, perhaps something used internally in your company, then `net.Socket` and `net.createConnection` will make light work of it.

Node supports both IPv4 and IPv6 in several ways: the `dns` module can query IPv4 and IPv6 records, and the `net` module can transmit and receive data to hosts on IPv4 and IPv6 networks.

The interesting thing about IP is it doesn't guarantee data integrity or delivery. For reliable communication, we need a transport layer protocol like TCP. There are also times when delivery isn't *always* required, although of course it's preferred—in these situations a lighter protocol is needed, and that's where UDP comes in. The next section examines TCP and UDP in more detail.

UDP AND HOW IT COMPARES TO TCP

Datagrams are the basic unit of communication in UDP. These messages are self-contained, holding a source, destination, and some user data. UDP doesn't guarantee

delivery or message order, or offer protection against duplicated data. Most protocols you'll use with Node programs will be built on TCP, but there are times when UDP is useful. If delivery isn't critical, but performance is desired, then UDP may be a better choice. One example is a streaming video service, where occasional glitches are an acceptable trade-off to gain more throughput.

TCP and UDP both use the same network layer—IP. Both provide services to application layer protocols. But they're very different. TCP is a connect-oriented and reliable byte stream service, whereas UDP is based around *datagrams*, and doesn't guarantee the delivery of data.

Contrast this to TCP, which is a full-duplex[1] connection-oriented protocol. In TCP, there are only ever two endpoints for a given connection. The basic unit of information passed between endpoints is known as a *segment*—the combination of a chunk of data along with a header. When you hear the term *packet*, a TCP segment is generally being referred to.

Although UDP packets include checksums that help detect corruption, which can occur as a datagram travels across the internet, there's no automatic retransmission of corrupt packets—it's up to your application to handle this if required. Packets with invalid data will be effectively silently discarded.

Every packet, whether it's TCP or UDP, has an origin and destination address. But the source and destination *programs* are also important. When your Node program connects to a DNS server or accepts incoming HTTP connections, there has to be a way to map between the packets traveling along the network and the programs that generated them. To fully describe a connection, you need an extra piece of information. This is known as a *port number*—the combination of a port number and an address is known as a socket. Read on to learn more about ports and how they relate to sockets.

SOCKETS

The basic unit of a network, from a programmer's perspective, is the socket. A socket is the combination of an IP address and a port number—and there are both TCP and UDP sockets. As you saw in the previous section, a TCP connection is full-duplex—opening a connection to a given host allows communication to flow *to* and *from* that host. Although the term *socket* is correct, historically "socket" meant the Berkeley Sockets API.

> **THE BERKELEY SOCKETS API** Berkeley Sockets, released in 1983, was an API for working with internet sockets. This is the original API for the TCP/IP suite. Although the origins lie in Unix, Microsoft Windows includes a networking stack that closely follows Berkeley Sockets.

There are well-known port numbers for standard TCP/IP services. They include DNS, HTTP, SSH, and more. These port numbers are usually odd numbers due to historical reasons. TCP and UDP ports are distinct so they can overlap. If an application layer

[1] Full-duplex: messages can be sent and received in the same connection.

protocol requires both TCP *and* UDP connections, then the convention is to use the same port number for both connections. An example of a protocol that uses both UDP and TCP is DNS.

In Node, you can create TCP sockets with the `net` module, and UDP is supported by the `dgram` module. Other networking protocols are also supported—DNS is a good example.

The following sections look at the application layer protocols included in Node's core modules.

7.1.2 Node's networking modules

Node has a suite of networking modules that allows you to build web and other server applications. Over the next few sections we'll cover DNS, TCP, HTTP, and encryption.

DNS

The Domain Name System (DNS) is the naming system for addressing resources connected to the internet (or even a private network). Node has a core module called `dns` for looking up and resolving addresses. Like other core modules, `dns` has asynchronous APIs. In this case, the implementation is also asynchronous, apart from certain methods that are backed by a thread pool. This means DNS queries in Node are fast, but also have a friendly API that is easy to learn.

You don't often have to use this module, but we've included techniques because it's a powerful API that can come in handy for network programming. Most application layer protocols, HTTP included, accept hostnames rather than IP addresses.

Node also provides modules for networking protocols that we're more familiar with—for example, HTTP.

HTTP

HTTP is important to most Node developers. Whether you're building web applications or calling web services, you're probably interacting with HTTP in some way. Node's `http` core module is built on the `net`, `stream`, `buffer`, and `events` modules. It's low-level, but can be used to create simple HTTP servers and clients without too much effort.

Due to the importance of the web to Node development, we've included several techniques that explore Node's `http` module. Also, when we're working with HTTP we often need to use encryption—Node also supports encryption through the `crypto` and `tls` modules.

ENCRYPTION

You should know the term *SSL*—Secure Sockets Layer—because it's how secure web pages are served to web browsers. Not just HTTP traffic gets encrypted, though—other services, like email, encrypt messages as well. Encrypted TCP connections use TLS: Transport Layer Security. Node's `tls` module is implemented using OpenSSL.

This type of encryption is called *public key cryptography*. Both clients and servers must have private keys. The server can then make its public key available so clients can

encrypt messages. To decrypt these messages, access to the server's *private* key is required.

Node supports TLS by allowing TCP servers to be created that support several ciphers. The TCP server itself inherits from `net.Server`—once you've got your head around TCP clients and servers in Node, encrypted connections are just an extension of these principles.

A solid understanding of TLS is important if you want to deploy web applications with Node. People are increasingly concerned with security and privacy, and unfortunately SSL/TLS is designed in such a way that programmer error can cause security weaknesses.

There's one final aspect of networking in Node that we'd like to introduce before we move on to the techniques for this chapter: how Node is able to give you asynchronous APIs to networking technologies that are sometimes blocking at the system level.

7.1.3 Non-blocking networking and thread pools

This section delves into Node's lower-level implementation to explore how networking works under the hood. If you're confused about what exactly "asynchronous" means in the context of networking, then read on for some background information on what makes Node's networking APIs tick.

Remember that in Node, APIs are said to be *asynchronous* when they accept a callback and return immediately. At the operating system level, I/O operations can also be asynchronous, or they can be synchronous and wrapped with threads to appear asynchronous.

Node employs several techniques to provide asynchronous network APIs. The main ones are non-blocking system calls and thread pools to wrap around blocking system calls.

Behind the scenes, most of Node's networking code is written in C and C++—the JavaScript code in Node's source gives you an asynchronous binding to features provided by `libuv` and `c-ares`.

Figure 7.3 shows Apple's Instruments tool recording the activity of a Node program that makes 50 HTTP requests. HTTP requests are non-blocking—each takes place using callbacks that are run on the main thread. The BSD sockets library, which is used by `libuv`, can make non-blocking TCP and UDP connections.

For HTTP and other TCP connections, Node is able to access the network using a system-level non-blocking API.

When writing networking or file system code, the Node code *looks* asynchronous: you pass a function to a method that will execute the function when the I/O operation has reached the desired state. But for file operations, the underlying implementation is not asynchronous: thread pools are used instead.

When dealing with I/O operations, understanding the difference between non-blocking I/O, thread pools, and asynchronous APIs is important if you want to truly understand how Node works.

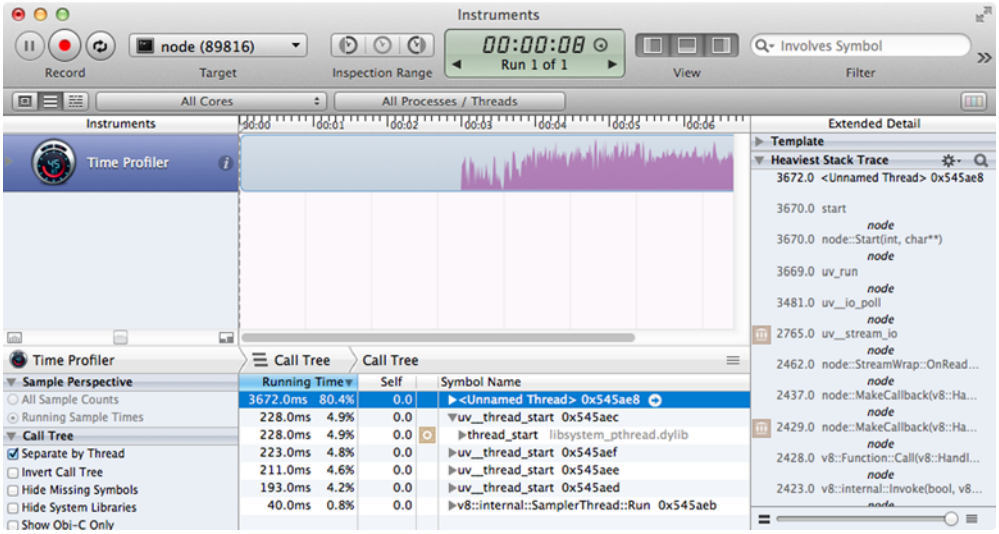


Figure 7.3 Node's threads when making HTTP requests

For those interested in reading more about `libuv` and networking, the freely available book, *An Introduction to libuv* (http://nikhilm.github.io/uvbook/networking.html#tcp) has a section on networking that covers TCP, DNS, and UDP.

Now on to the first set of networking techniques: TCP clients and servers.

7.2 TCP clients and servers

Node has a simple API for creating TCP connections and servers. Most of the lowest-level classes and methods can be found in the `net` module. In the next technique, you'll learn how to create a TCP server and track the clients that connect to it. The cool thing about this is that higher-level protocols like HTTP are built on top of the TCP API, so once you've got the hang of TCP clients and servers, you can really start to exploit some of the more subtle features of the HTTP API as well.

TECHNIQUE 45 Creating a TCP server and tracking clients

The `net` module forms the foundation of many of Node's networking features. This technique demonstrates how to create a TCP server.

Problem

You want to start your own TCP server, bind to a port, and send data over the network.

Solution

Use `net.createServer` to create a server, and then call `server.listen` to bind it to a port. To connect to the server, either use the command-line tool `telnet` or create an in-process client connection with its client counterpart, `net.connect`.

Discussion

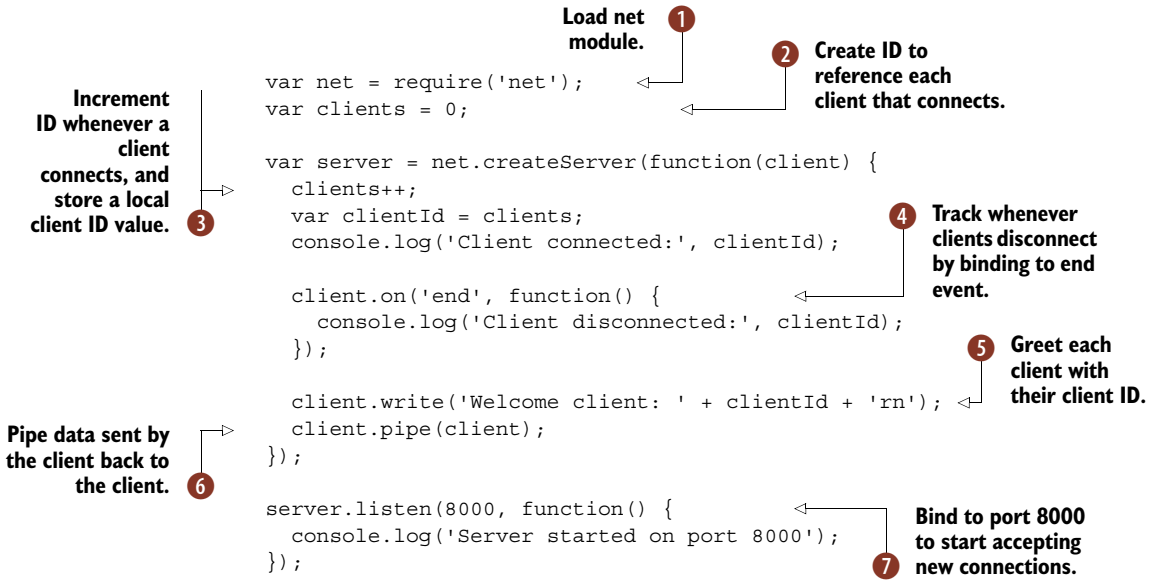
The `net.createServer` method returns an object that can be used to listen on a given TCP port for incoming connections. When a client makes a new connection, the callback

passed to `net.createServer` will run. This callback receives a connection object which extends `EventEmitter`.

The server object itself is an instance of `net.Server`, which is just a wrapper around the `net.Socket` class. It's interesting to note that `net.Socket` is implemented using a duplex stream—for more on streams, see chapter 5.

Before going into more theory, let's look at an example that you can run and connect to with `telnet`. The following listing shows a simple TCP server that accepts connections and echoes data back to the client.

Listing 7.1 A simple TCP server




```
var net = require('net');
var clients = 0;

var server = net.createServer(function(client) {
  clients++;
  var clientId = clients;
  console.log('Client connected:', clientId);

  client.on('end', function() {
    console.log('Client disconnected:', clientId);
  });

  client.write('Welcome client: ' + clientId + 'rn');
  client.pipe(client);
});

server.listen(8000, function() {
  console.log('Server started on port 8000');
});
```

To try out this example, run `node server.js` to start a server, and then run `telnet localhost 8000` to connect to it with `telnet`. You can connect several times to see the ID incremented. If you disconnect, a message should be printed that contains the correct client ID.

Most programs that use TCP clients and servers load the `net` module 1. Once it has been loaded, TCP servers can be created using `net.createServer`, which is actually just a shortcut for `new net.Server` with a `listener` event listener. After a server has been instantiated, it can be set to listen for connections on a given port using `server.listen` 7.

To echo back data sent by the client, `pipe` is used 6. Sockets are streams, so you can use the standard stream API methods with them as you saw in chapter 5.

In this example, we track each client that has connected using a numerical ID by incrementing a "global" value 2 that tracks the number of clients 3. The total number

of connected clients is stored in the callback's scope by creating a local variable in the connection callback called `clientId`.

This value is displayed whenever a client connects ❺ or disconnects ❹. The client argument passed to the server's callback is actually a socket—you can write to it with `client.write` and data will be sent over the network.

The important thing to note is any event listener added to the socket in the server's callback will share the same scope—it will create closures around any variables inside this callback. That means the client ID is unique to each connection, and you can also store other values that clients might need. This forms a common pattern employed by client-server applications in Node.

The next technique builds on this example by adding client connections in the same process.

TECHNIQUE 46 Testing TCP servers with clients

Node makes creating TCP servers *and* clients in the same process a breeze—it's an approach particularly useful for testing your network programs. In this technique you'll learn how to make TCP clients, and use them to test a server.

Problem

You want to test a TCP server.

Solution

Use `net.connect` to connect to the server's port.

Discussion

Due to how TCP and UDP ports work, it's entirely possible to create multiple servers and clients in the same process. For example, a Node HTTP server could also run a simple TCP server on another port that allows `telnet` connections for remote administration.

In technique 45, we demonstrated a TCP server that can track client connections by issuing each client a unique ID. Let's write a test to ensure this worked correctly.

Listing 7.2 shows how to create client connections to an in-process server, and then run assertions on the data sent over the network by the server. Of course, technically this isn't running over a real network because it all happens in the same process, but it could easily be adapted to work that way; just copy the program to a server and specify its IP address or hostname in the client.

Listing 7.2 Creating TCP clients to test servers

```
var assert = require('assert');
var net = require('net');
var clients = 0;
var expectedAssertions = 2;

var server = net.createServer(function(client) {
  clients++;
  var clientId = clients;
  console.log('Client connected:', clientId);
```

```
  client.on('end', function() {
    console.log('Client disconnected:', clientId);
  });

  client.write('Welcome client: ' + clientId + '\r\n');
  client.pipe(client);
});

server.listen(8000, function() {
  console.log('Server started on port 8000');

  runTest(1, function() {
    runTest(2, function() {
      console.log('Tests finished');
      assert.equal(0, expectedAssertions);
      server.close();
    });
  });
});

function runTest(expectedId, done) {
  var client = net.connect(8000);

  client.on('data', function(data) {
    var expected = 'Welcome client: ' + expectedId + '\r\n';
    assert.equal(data.toString(), expected);
    expectedAssertions--;
    client.end();
  });

  client.on('end', done);
}
```

1 runTest function accepts a callback so additional tests can be scheduled.

2 After tests have finished, a counter is checked to see if tests were executed.

3 Once tests and assertions have run, server can be closed.

4 runTest function connects to server, checks that it displays expected client ID, and then disconnects.

5 net.connect is used to connect to server; it returns an EventEmitter object that can be used to listen for events.

6 Client's data event is used to grab the message the server displays after the client connects.

7 Disconnect client when data has been sent.

8 When client has finished sending data, run callback.

This is a long example, but it centers around a relatively simple method: `net.connect`. This method accepts some optional arguments to describe the remote host. Here we've just specified a port number, but the second argument can be a hostname or IP address—`localhost` is the default 5. It also accepts a callback, which can be used to write data to the other end once the client has connected. Remember that TCP servers are full-duplex, so both ends can receive and send data.

The `runTest` function in this example will run once the server has started listening 1. It accepts an expected client ID, and a callback called `done` 4. The callback will be triggered once the client has connected, received some data by subscribing to the `data` event 6, and then disconnected.

Whenever clients are disconnected, the `end` event will be emitted. We bind the `done` callback to this event 8. When the test has finished in the `data` callback, we call `client.end` to disconnect the socket manually, but `end` events will be triggered when servers close connections, as well.

The `data` event is where the main test is performed ❼. The expected message is passed to `assert.equal` with the data passed to the event listener. The data is a buffer, so `toString` is called for the assertion to work. Once the test has finished, and the `end` event has been triggered ❼, the callback passed to `runTest` will be executed.

> **Error handling**
>
> If you need to collect errors generated by TCP connections, just subscribe to the `error` event on the `EventEmitter` objects returned by `net.connect`. If you don't, an exception will be raised; this is standard behavior in Node.
>
> Unfortunately, this isn't easy to work with when dealing with sets of distinct network connections. In such cases, a better technique is to use the `domain` module. Creating a new domain with `domain.create()` will cause error events to be sent to the domain; you can then handle them in a centralized error handler by subscribing to `error` events on the domain.
>
> For more about domains, refer to technique 21.

We've used two calls to `runTest` here by calling one inside the callback. Once both have run, the number of expected assertions is checked ❷, and the server is shut down ❸.

This example highlights two important things: clients and servers can be run together in-process, and Node TCP clients and servers are easy to unit test. If the server in this example were a remote service that we had no control over, then we could create a "mock" server for the express purpose of testing our client code. This forms the basis of how most developers write tests for web applications written with Node.

In the next technique we'll dig deeper into TCP networking by looking at Nagle's algorithm and how it can affect the performance characteristics of network traffic.

TECHNIQUE 47 Improve low-latency applications

Although Node's `net` module is relatively high-level, it does provide access to some low-level functionality. One example of this is control over the `TCP_NODELAY` flag, which determines whether Nagle's algorithm is used. This technique explains what Nagle's algorithm is, when you should use it, and how to turn it off for specific sockets.

Problem

You want to improve connection latency in a real-time application.

Solution

Use `socket.setNoDelay()` to enable `TCP_NODELAY`.

Discussion

Sometimes it's more efficient to move batches of things together, rather than separately. Every day millions of products are shipped around the globe, but they're not carried one at a time—instead they're grouped together in shipping containers, based

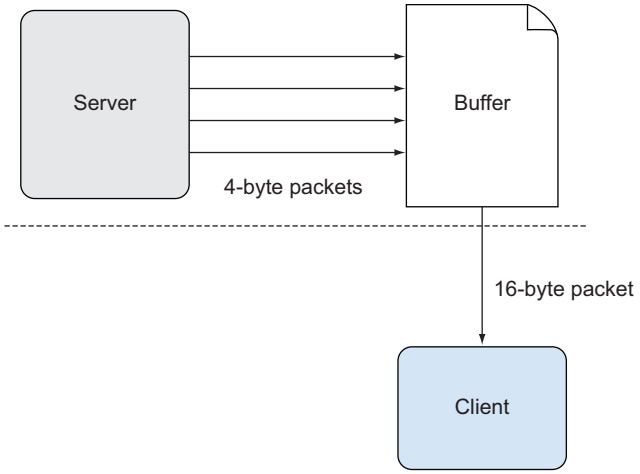


Figure 7.4 When Nagle's algorithm is used, smaller packets are collected into a larger payload.

on their final destination. TCP works exactly the same way, and this feature is made possible by Nagle's algorithm.

Nagle's algorithm says that when a connection has data that hasn't yet been acknowledged, small segments should be retained. These small segments will be batched into larger segments that can be transmitted when sufficient data has been acknowledged by the recipient.

In networks where many small packets are transmitted, it can be desirable to reduce congestion by combining small outgoing messages, and sending them together. But sometimes latency is desired over all else, so transmitting small packets is important.

This is particularly true for interactive applications, like `ssh`, or the X Window System. In these applications, small messages should be delivered without delay to create a sense of real-time feedback. Figure 7.4 illustrates the concept.

Certain classes of Node programs benefit from turning off Nagle's algorithm. For example, you may have created a REPL that transmits a single character at a time as the user types messages, or a game that transmits location data of players. The next listing shows a program that disables Nagle's algorithm.

Listing 7.3 Turning off Nagle's algorithm

```
var net = require('net');
var server = net.createServer(function(c) {
  c.setNoDelay(true);
  c.write('377375042377373001', 'binary');
  console.log('server connected');
  c.on('end', function() {
    console.log('server disconnected');
    server.unref();
  });
  c.on('data', function(data) {
    process.stdout.write(data.toString());
    c.write(data.toString());
```

1 Turn off Nagle's algorithm.

2 Force client to use character mode.

3 Call unref() so that when last client disconnects, program exits.

4 Print out characters from client to the server's terminal.

```
  });
});
server.listen(8000, function() {
  console.log('server bound');
});
```

To use this example, run the program in a terminal with `node nagle.js`, and then connect to it with `telnet 8000`. The server turns off Nagle's algorithm ❶, and then forces the client to use character mode ❷. Character mode is part of the Telnet Protocol (RFC 854), and will cause the Telnet client to send a packet whenever a key is pressed.

Next, `unref` is used ❸ to cause the program to exit when there are no more client connections. Finally, the `data` event is used to capture characters sent by the client and print them to the server's terminal ❹.

This technique could form the basis for creating low-latency applications where data integrity is important, which therefore excludes UDP. If you really want to get more control over the transmission of data, then read on for some techniques that use UDP.

7.3 UDP clients and servers

Compared to TCP, UDP is a much simpler protocol. That can mean more work for you: rather than being able to rely on data being sent and received, you have to cater to UDP's more volatile nature. UDP is suitable for query-response protocols, which is why it's used for the Domain Name System (DNS). It's also stateless—if you want to transfer data and you value lower latency over data integrity, then UDP is a good choice. That might sound unusual, but there are applications that fit these characteristics: media streaming protocols and online games generally use UDP.

If you wanted to build a video streaming service, you could transfer video over TCP, but each packet would have a lot of overhead for ensuring delivery. With UDP, it would be possible for data to be lost with no simple means of discovery, but with video you don't care about occasional glitches—you just want data as fast as possible. In fact, some video and image formats can survive a small amount of data loss: the JPEG format is resilient to corrupt bytes to a certain extent.

The next technique combines Node's file streams with UDP to create a simple server that can be used to transfer files. Although this can potentially result in data loss, it can be useful when you care about speed over all else.

TECHNIQUE 48 Transferring a file with UDP

This technique is really about sending data from a stream to a UDP server rather than creating a generalized file transfer mechanism. You can use it to learn the basics of Node's datagram API.

Problem

You want to transfer data from a client to a server using datagrams.

Solution

Use the `dgram` module to create datagram sockets, and then send data with `socket.send`.

Discussion

Sending datagrams is similar to using TCP sockets, but the API is slightly different, and datagrams have their own rules that reflect the actual structure of UDP packets. To set up a server, use the following snippet:

```
var dgram = require('dgram');
var socket = dgram.createSocket('udp4');
socket.bind(4000);
```

1 Create UDP socket

2 Bind it to a port

This example creates a socket that will act as the server 1, and then binds it to a port 2. The port can be anything you want, but in both TCP and UDP the first 1,023 ports are privileged.

The client API is different from TCP sockets because UDP is a *stateless* protocol. You must write data a packet at a time, and packets (datagrams) must be relatively small—under 65,507 bytes. The maximum size of a datagram depends on the Maximum Transmission Unit (MTU) of the network. 64 KB is the upper limit, but isn't usually used because large datagrams may be silently dropped by the network.

Creating a client socket is the same as servers—use `dgram.createSocket`. Sending a datagram requires a buffer for the payload, an offset to indicate where in the buffer the message starts, the message length, the server port, the remote IP, and an optional callback that will be triggered when the message has been sent:

```
var message = 'Sample message';
socket.send(new Buffer(message), 0, message.length, port, remoteIP);
```

Listing 7.4 combines a client and a server into a single program. To run it, you must issue two commands: `node udp-client-server.js server` to run the server, and then `node udp-client-server.js client remoteIP` to start a client. The `remoteIP` option can be omitted if you run both locally; we designed this example to be a single file so you can easily copy it to another computer to test sending things over the internet or a local network.

Listing 7.4 A UDP client and server

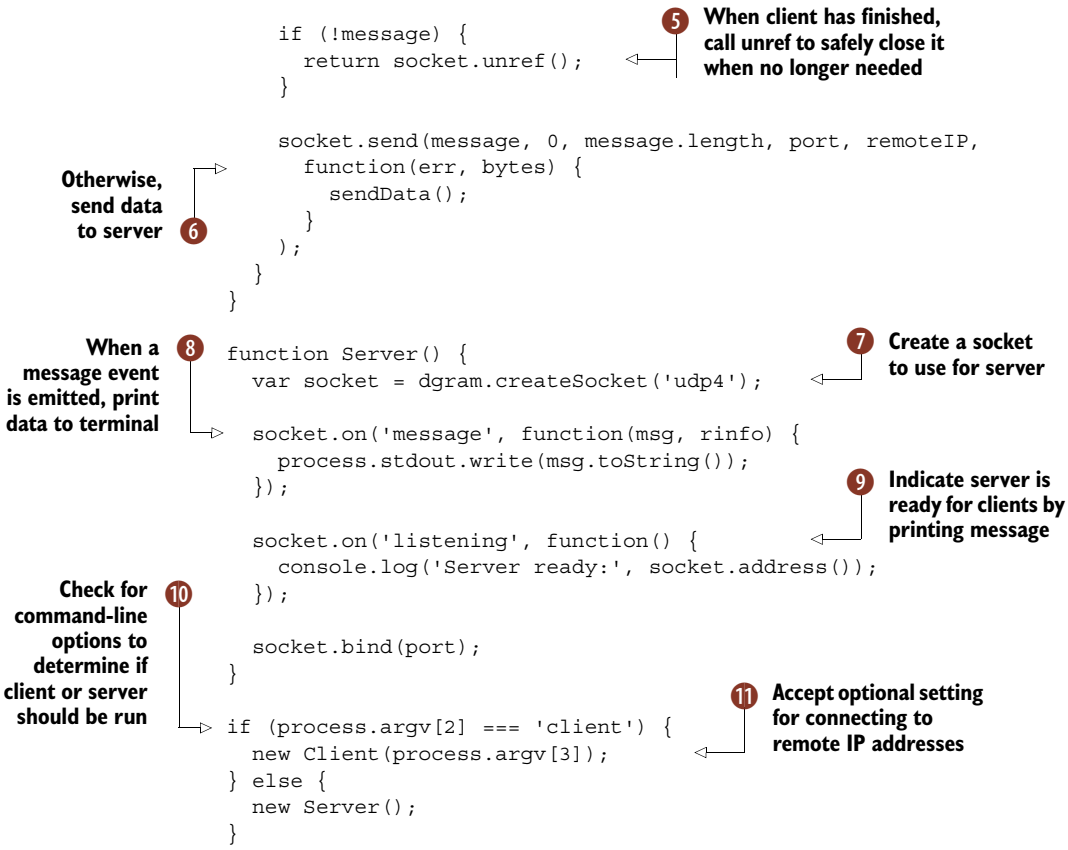
```
var dgram = require('dgram');
var fs = require('fs');
var port = 41230;
var defaultSize = 16;

function Client(remoteIP) {
  var inStream = fs.createReadStream(__filename);
  var socket = dgram.createSocket('udp4');

  inStream.on('readable', function() {
    sendData();
  });

  function sendData() {
    var message = inStream.read(defaultSize);
```

1 Create a readable stream for current file

2 Make a new datagram socket to use as client

3 When readable stream is ready, start sending its data to the server

4 Use stream.read(size) to read chunks of data



When you run this example, it starts by checking the command-line options to see if the client or server is required ⑩. It also accepts an optional argument for clients so you can connect to remote servers ⑪.

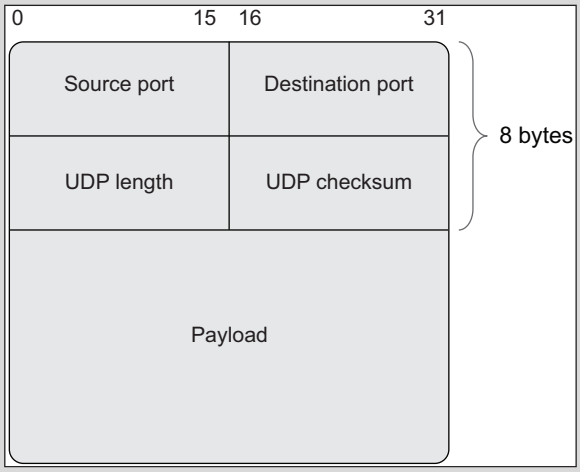
If the client was specified, then a new client will be created by making a new datagram socket ②. This involves using a read stream from the `fs` module so we have some data to send to the server ①—we've used `__filename` to make it read the current file, but you could make it send any file.

Before sending any data, we need to make sure the file has been opened and is ready for reading, so the `readable` event is subscribed to ③. The callback for this event executes the `sendData` function. This will be called repeatedly for each chunk of the file—files are read in small chunks at a time using `inStream.read` ④, because UDP packets can be silently dropped if they're too large. The `socket.send` method is used to push the data to the server ⑥. The `message` object returned when reading the file is an instance of `Buffer`, and it can be passed straight to `socket.send`.

When all of the data has been read, the last chunk is set to `null`. The `socket.unref` ⑤ method is called to cause the program to exit when the socket is no longer required—in this case, once it has sent the last message.

Datagram packet layout and datagram size

UDP packets are comparatively simple. They're composed of a source port, the destination port, datagram length, checksum, and the payload data. The length is the total size of the packet—the header size added to the payload's size. When deciding on your application's buffer size for UDP packets, you should remember that the length passed to `socket.send` is only for the buffer (payload), and the overall packet size must be under the MTU on the network. The structure of a datagram looks like the following.



The UDP header is 8 bytes, followed by an optional payload of up to 65,507 bytes for IPv4 and 65,527 bytes for IPv6.

The server is simpler than the client. It sets up a socket in the same way ❼, and then subscribes to two events. The first event is `message`, which is emitted when a datagram is received ❽. The data is written to the terminal by using `process.stdout.write`. This looks better than using `console.log` because it won't automatically add newlines.

The `listening` event is emitted when the server is ready to accept connections ❾. A message is displayed to indicate this so you know it's safe to try connecting a client.

Even though this is a simple example, it's immediately obvious how UDP is different from TCP—you need to pay attention to the size of the messages you send, and realize that it's possible for messages to get lost. Although datagrams have a checksum, lost or damaged packets aren't reported to the application layer, which means data loss is possible. It's generally best to use UDP for sending data where assured integrity is second place to low latency and throughput.

In the next technique you'll see how to build on this example by sending messages back to the client, essentially setting up bidirectional communication channels with UDP.

TECHNIQUE 49 UDP client server applications

UDP is often used for query-response protocols, like DNS and DHCP. This technique demonstrates how to send messages back to the client.

Problem

You've created a UDP server that responds to requests, but you want to send messages back to the client.

Solution

Once you've created a server and it has received a message, create a datagram connection *back* to the client based on the `rinfo` argument that's passed to `message` events. Optionally create a unique reference by combining the client port and IP address to send subsequent messages.

Discussion

Chat servers are the classic network programming example for new Node programmers, but this one has a twist—it uses UDP instead of TCP or HTTP.

TCP connections are different from UDP, and this is apparent in the design of Node's networking API. TCP connections are represented as a stream of bidirectional events, so sending a message back to the sender is straightforward—once a client has connected you can write messages to it at any time using `client.write`. UDP, on the other hand, is *connectionless*—messages are received without an active connection to the client.

There are some protocol-level similarities that enable you to respond to messages from clients, however. Both TCP and UDP connections use source and destination ports. Given a suitable network setup, it's possible to open a connection back to the client based on this information. In Node the `rinfo` object that's included with every `message` event contains the relevant details. Figure 7.5 shows how messages flow between two clients using this scheme.

Listing 7.5 presents a client-server program that allows clients to connect to a central server over UDP and message each other. The server keeps details of each client in

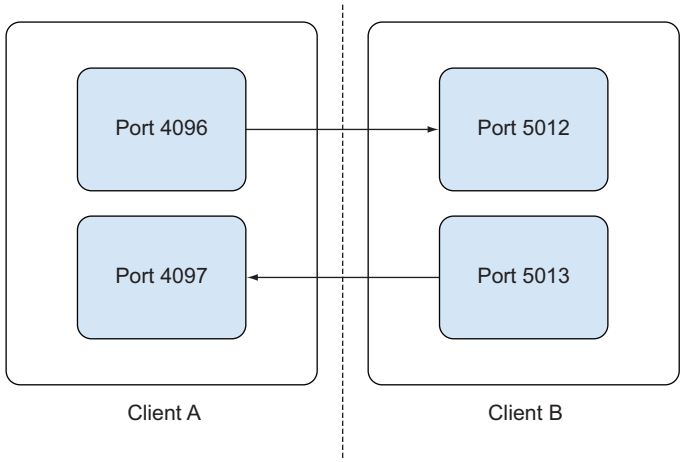


Figure 7.5 Even though UDP isn't full-duplex, it's possible to create connections in two directions given a port number at both sides.

an array, so it can refer to each one uniquely. By storing the client's address and port, you can even run multiple clients on the same machine—it's safe to run this program several times on the same computer.

Listing 7.5 Sending messages back to clients

```
var assert = require('assert');
var dgram = require('dgram');
var fs = require('fs');
var defaultSize = 16;
var port = 41234;

function Client(remoteIP) {
  var socket = dgram.createSocket('udp4');
  var readline = require('readline');
  var rl = readline.createInterface(process.stdin, process.stdout);

  socket.send(new Buffer('<JOIN>'), 0, 6, port, remoteIP);

  rl.setPrompt('Message> ');
  rl.prompt();

  rl.on('line', function(line) {
    sendData(line);
  }).on('close', function() {
    process.exit(0);
  });

  socket.on('message', function(msg, rinfo) {
    console.log('\n<' + rinfo.address + '>', msg.toString());
    rl.prompt();
  });

  function sendData(message) {
    socket.send(new Buffer(message), 0, message.length, port, remoteIP,
      function(err, bytes) {
        console.log('Sent:', message);
        rl.prompt();
      }
    );
  }
}

function Server() {
  var clients = [];
  var server = dgram.createSocket('udp4');

  server.on('message', function(msg, rinfo) {
    var clientId = rinfo.address + ':' + rinfo.port;

    msg = msg.toString();

    if (!clients[clientId]) {
      clients[clientId] = rinfo;
```

1. Use readline module to handle user input.
2. Whenever a client first joins, send special join message.
3. Send messages to server when the user types a message and presses Return.
4. Listen for messages from other users.
5. Take user's message and create a new buffer that can then be sent as UDP message to server.
6. Listen for new messages from clients.
7. Combine client's port and address to make a unique reference to it.
8. If client hasn't been seen before, keep a record of its connection details.

```
    }

    if (msg.match(/^</)) {
      console.log('Control message:', msg);
      return;
    }

    for (var client in clients) {
      if (client !== clientId) {
        client = clients[client];
        server.send(
          new Buffer(msg), 0,
          msg.length, client.port, client.address,
          function(err, bytes) {
            if (err) console.error(err);
            console.log('Bytes sent:', bytes);
          }
        );
      }
    }
  });

  server.on('listening', function() {
    console.log('Server ready:', server.address());
  });

  server.bind(port);
}

module.exports = {
  Client: Client,
  Server: Server
};

if (!module.parent) {
  switch (process.argv[2]) {
    case 'client':
      new Client(process.argv[3]);
      break;

    case 'server':
      new Server();
      break;

    default:
      console.log('Unknown option');
  }
}
```

9 If message is wrapped in angled brackets, treat it as a control message.

10 Send message to every other client.

This example builds on technique 48—you can run it in a similar way. Type `node udp-chat.js server` to start a server, and then `node udp-chat.js client` to connect a client. You should run more than one client for it to work; otherwise messages won't get routed anywhere.

The `readline` module has been used to capture user input in a friendly manner ❶. Like most of the other core modules you've seen, this one is event-based. It'll emit the `line` event whenever a line of text is entered ❸.

Before messages can be sent by the user, an initial `join` message is sent ❷. This is just to let the server know it has connected—the server code uses it to store a unique reference to the client ❽.

The `Client` constructor wraps `socket.send` inside a function called `sendData` ❺. This is so messages can be easily sent whenever a line of text is typed. Also, when a client itself receives a message, it'll print it to the console and create a new prompt ❹

Messages received by the server ❻ are used to create a unique reference to the client by combining the port and remote address ❼. We get all of this information from the `rinfo` object, and it's safe to run multiple clients on the same machine because the port will be the client's port rather than the port the server listens on (which doesn't change). To understand how this is possible, recall that UDP headers include a source and destination port, much like TCP.

Finally, whenever a message is seen that isn't a control message ❾, each client is iterated over and sent the message ❿. The client that has sent the message won't receive a copy. Because we've stored references to each `rinfo` object in the `clients` array, messages can be sent back to clients.

Client-server networking is the basis of HTTP. Even though HTTP uses TCP connections, it's slightly different from the type of protocols you've seen so far: it's stateless. That means you need different patterns to model it. The next section has more details on how to make HTTP clients and servers.

7.4 HTTP clients and servers

Today most of us work with HTTP—whether we're producing or consuming web services, or building web applications. The HTTP protocol is stateless and built on TCP, and Node's HTTP module is similarly built on top of its TCP module.

You could, of course, use your own protocol built with TCP. After all, HTTP is built on top of TCP. But due to the prevalence of web browsers and tools for working with web-based services, HTTP is a natural fit for many problems that involve communicating between remote systems.

In the next section you'll learn how to write a basic HTTP server using Node's core modules.

TECHNIQUE 50 HTTP servers

In this technique you'll learn how to create HTTP servers with Node's `http` module. Although this is more work than using a web framework built on top of Node, popular web frameworks generally use the same techniques internally, and the objects they expose are derived from Node's standard classes. Understanding the underlying modules and classes is therefore useful for working extensively with HTTP.

■ **Problem**

You want to run HTTP servers and test them.

■ **Solution**

Use `http.createServer` and `http.createClient`.

■ **Discussion**

The `http.createServer` method is a shortcut for creating a new `http.Server` object that descends from `net.Server`. The HTTP server is extended to handle various elements of the HTTP protocol—parsing headers, dealing with response codes, and setting up various events on sockets. The major focus in Node's HTTP handling code is parsing; a C++ wrapper around Joyent's own C parser library is used. This library can extract header fields and values, Content-Length, request method, response status code, and more.

The following listing shows a small "Hello World" web server that uses the `http` module.

Listing 7.6 A simple HTTP server

```
var assert = require('assert');
var http = require('http');

var server = http.createServer(function(req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.write('Hello, world.\r\n');
  res.end();
});

server.listen(8000, function() {
  console.log('Listening on port 8000');
});

var req = http.request({
  port: 8000
}, function(res) {
  console.log('HTTP headers:', res.headers);
  res.on('data', function(data) {
    console.log('Body:', data.toString());
    assert.equal('Hello, world.\r\n', data.toString());
    assert.equal(200, res.statusCode);
    server.unref();
  });
});

req.end();
```

1 Load HTTP module

2 Create new HTTP server and pass a callback that will run when there's a new request

3 Write some sensible headers for text-based response

4 Write message back to the client

5 Set server to listen on port 8000

6 Make request using http.request

7 Add listener to the data event and make sure response is what was expected

The `http` module contains both Node's client and server HTTP classes 1. The `http.createServer` creates a new server object and returns it. The argument is a callback that receives `req` and `res` objects—request and response, respectively 2. You may be familiar with these objects if you've used higher-level Node web frameworks like Express and restify.

The interesting thing about the listener callback passed to `http.createServer` is that it behaves much like the listener passed to `net.createServer`. Indeed, the mechanism is the same—we're creating TCP sockets, but layering HTTP on top. The main conceptual difference between the HTTP protocol and TCP socket communication is a question of state: HTTP is a stateless protocol. It's perfectly acceptable and in fact typical to create and tear down TCP sockets *per request*. This partly explains why Node's underlying HTTP implementation is low-level C++ and C: it needs to be fast and use as little memory as possible.

In listing 7.6, the listener runs for every request. In the TCP example from technique 45, the server kept a connection open as long as the client was connected. Because HTTP connections are just TCP sockets, we can use `res` and `req` like the sockets in listing 7.6: `res.write` will write to the socket (4), and headers can be written back with `res.writeHead` (3), which is where the socket connection and HTTP APIs visibly diverge—the underlying socket will be closed as soon as the response has been written.

After the server has been set up, we can set it to listen on a port with `server.listen` (5).

Now that we can create servers, let's look at creating HTTP requests. The `http.request` method will create new connections (6), and accepts an `options` argument object and a callback that will be run when a connection is made. This means we still need to attach a `data` listener to the *response* passed to the callback to slurp down any sent data.

The `data` callback ensures the response from the server has the expected format: the body content and status code (7) are checked. The server is told to stop listening for connections when the last client has disconnected by calling `server.unref`, which means the script exits cleanly. This makes it easy to see if any errors were encountered.

One small feature of the HTTP module is the `http.STATUS_CODES` object. This allows human-readable messages to be generated by looking up the integer status code: `http.STATUS_CODES[302]` will evaluate to `Moved Temporarily`.

Now that you've seen how to create HTTP servers, in the next technique we'll look at the role state plays in HTTP clients—despite HTTP being a stateless protocol—by implementing HTTP redirects.

TECHNIQUE 51 Following redirects

Node's `http` module provides a convenient API for handling HTTP requests. But it doesn't follow redirects, and because redirects are so common on the web, it's an important technique to master. You could use a popular third-party module that handles redirection, like the popular *request* module by Mikeal Rogers,[2] but you'll learn much more about Node by looking at how it can be implemented with the core modules.

[2] https://npmjs.org/package/request

In this technique we'll look at how to use straightforward JavaScript to maintain state across several requests. This allows a redirect to be followed correctly without creating redirect loops or other issues.

Problem

You want to download pages and follow redirects if necessary.

Solution

Handling redirection is fairly straightforward once the basics of the protocol are understood. The HTTP standard defines status codes that denote when redirection has occurred, and it also states that clients should detect infinite redirect loops. To satisfy these requirements, we'll use a simple prototype class to retain the state of each request, redirecting if needed and detecting redirect loops.

Discussion

In this example we'll use Node's core `http` module to make a `GET` request to a URL that we know will generate a redirection. To determine if a given response is a redirect, we need to check whether the returned status code begins with a 3. All of the status codes in the 3xx family of responses indicate that a redirect of some kind has occurred.

According to the specification, this is the full set of status codes that we need to deal with:

- *300*—Multiple choices
- *301*—Moved permanently
- *302*—Found
- *303*—See other
- *304*—Not modified
- *305*—See proxy
- *307*—Temporary redirect

Exactly how each of these status codes is handled depends on the application. For example, it might be extremely important for a search engine to identify responses that return a 301, because it means the search engine's list of URLs should be permanently updated. For this technique we simply need to follow redirects, which means a single statement is sufficient to check whether the request is being redirected: `if (response.statusCode >= 300 && response.statusCode < 400)`.

Testing for redirection loops is more involved. A request can no longer exist in isolation—we need to track the state of several requests. The easiest way to model this is by using a class that includes an instance variable for counting how many redirects have occurred. When the counter reaches a limit, an error is raised. Figure 7.6 shows how HTTP redirects are handled.

Before writing any code, it's important to consider what kind of API we need. Since we've already determined a "class" should be used to manage state, then users of our module will need to instantiate an instance of this class. Node's `http` module is asynchronous, and our code should be as well. That means that to get a result back, we'll have to pass a callback to a method.

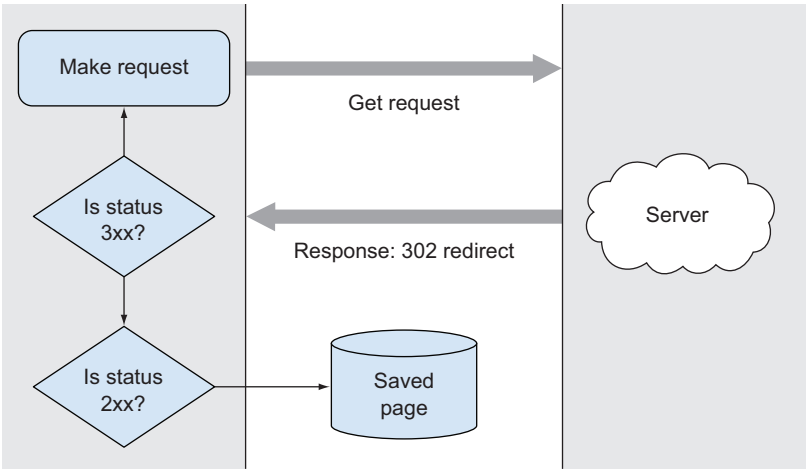


Figure 7.6 Redirection is cyclical, and requests will be made until a 200 status is encountered.

The signature for this callback should use the same format as Node's core modules, where an error variable is the first parameter. Designing the API in this way has the advantage of making error handling straightforward. Making an HTTP request can result in several errors, so it's important to handle them correctly.

The following listing puts all of this together to successfully follow redirects

Listing 7.7 Making an HTTP `GET` request that follows redirects

```
var http = require('http');
var https = require('https');
var url = require('url');
var request;

function Request() {
  this.maxRedirects = 10;
  this.redirects = 0;
}

Request.prototype.get = function(href, callback) {
  var uri = url.parse(href);
  var options = { host: uri.host, path: uri.path };
  var httpGet = uri.protocol === 'http:' ? http.get : https.get;

  console.log('GET:', href);

  function processResponse(response) {
    if (response.statusCode >= 300 && response.statusCode < 400) {
      if (this.redirects >= this.maxRedirects) {
        this.error = new Error('Too many redirects for: ' + href);
      } else {
        this.redirects++;
        href = url.resolve(options.host, response.headers.location);
```

1 url module has useful methods for parsing URLs

2 Define constructor to manage request state

3 Parse URLs into format used by Node's http module, and determine if HTTPS should be used

4 Check to see if statusCode is in the range for HTTP redirects

5 Increment redirection counter, and use url.resolve to ensure relative URLs are expanded to absolute URLs

```
        return this.get(href, callback);
      }
    }

    response.url = href;
    response.redirects = this.redirects;

    console.log('Redirected:', href);

    function end() {
      console.log('Connection ended');
      callback(this.error, response);
    }

    response.on('data', function(data) {
      console.log('Got data, length:', data.length);
    });

    response.on('end', end.bind(this));
  }

  httpGet(options, processResponse.bind(this))
    .on('error', function(err) {
      callback(err);
    });
};

request = new Request();
request.get('http://google.com/', function(err, res) {
  if (err) {
    console.error(err);
  } else {
    console.log('Fetched URL:', res.url,
      'with', res.redirects, 'redirects');
    process.exit();
  }
});
```

6 Use Fuction.prototype.bind to bind callback to Request instance so this points to correct object

7 Instantiate Request and fetch a URL

Running this code will display the last-fetched URL, and the number of times the request was redirected. Try it with a few URLs to see what happens: even nonexistent URLs that result in DNS errors should cause error information to be printed to `stderr`.

After loading the necessary modules 1, the `Request` 2 constructor function is used to create an object that models the lifetime of a request. Using a class in this way keeps implementation details neatly encapsulated from the user. Meanwhile, the `Request.prototype.get` method does most of the work. It sets up a standard HTTP request, or HTTPS if necessary, and then calls itself recursively whenever a redirect is encountered. Note that the URL has to be parsed 3 into an object that we use to create the `options` object that is compatible with Node's `http` module.

The request protocol (HTTP or HTTPS) is checked to ensure we use the right method from Node's `http` or `https` module. Some servers are configured to always

redirect HTTP traffic to HTTPS. Without checking for the protocol, this method would repeatedly fetch the original HTTP URL until `maxRedirects` is hit—this is a trivial mistake that's easily avoided.

Once the response has been received, the `statusCode` is checked ❹. The number of redirects is incremented as long as `maxRedirects` hasn't been reached ❺. This process is repeated until there's no longer a status in the 300 range, or too many redirects have been encountered.

When the final request has finished (or the first if there were no redirects), the user-supplied `callback` function is run. The standard Node API signature of `error, result` has been used here to stay consistent with Node's core modules. An error is generated when `maxRedirects` is reached, or when creating the HTTP request by listening for an `error` event.

The user-supplied callback runs after the last request has finished, allowing the callback to access the requested resource. This is handled by running the callback after the `end` event for the last request has been triggered, and by binding the event handler to the current `Request` instance ❻. Binding the event handler means it'll have access to any useful instance variables that the user might need—including errors that are stored in `this.error`.

Lastly, we create an instance of `Request` ❼ to try out the class. You can use it with other URLs if you like.

This technique illustrates an important point: state is important, even though HTTP is technically a stateless protocol. Some misconfigured web applications and servers can create redirect loops, which would cause a client to fetch URLs forever until it's forcibly stopped.

Though listing 7.7 showcases some of Node's HTTP- and URL-handling features, it isn't a complete solution. For a more advanced HTTP API, take a look at Request by Mikeal Rogers (https://github.com/mikeal/request), a widely used simplified Node HTTP API.

In the next technique we'll dissect a simple HTTP proxy. This expands on the client and server techniques discussed here, and could be expanded to create numerous useful applications.

TECHNIQUE 52 HTTP proxies

HTTP proxies are used more often than you might expect—ISPs use transparent proxies to make networks more efficient, corporate systems administrators use caching proxies to reduce bandwidth, and web application DevOps use them to improve the performance of their apps. This technique only scratches the surface of proxies—it catches HTTP requests and responses, and then mirrors them to their intended destinations.

Problem

You want to capture and retransmit HTTP requests.

Solution

Use Node's built-in HTTP module to act as a simple HTTP proxy.

Discussion

A proxy server offers a level of redirection, which facilitates a variety of useful applications: caching, logging, and security-related software. This technique explores how to use the core `http` module to create HTTP proxies. Fundamentally all that's required is an HTTP server that catches requests, and then an HTTP client to clone them.

The `http.createServer` and `http.request` methods can catch and retransmit requests. We'll also need to interpret the original request so we can safely copy it—the `url` core module has an ideal URL-parsing method that can help do this.

The next listing shows how simple it is to create a working proxy in Node.

Listing 7.8 Using the `http` module to create a proxy

```
var http = require('http');
var url = require('url');

http.createServer(function(req, res) {
  console.log('start request:', req.url);
  var options = url.parse(req.url);
  options.headers = req.headers;
  var proxyRequest = http.request(options, function(proxyResponse) {
    proxyResponse.on('data', function(chunk) {
      console.log('proxyResponse length:', chunk.length);
      res.write(chunk, 'binary');
    });

    proxyResponse.on('end', function() {
      console.log('proxied request ended');
      res.end();
    });

    res.writeHead(proxyResponse.statusCode, proxyResponse.headers);
  });

  req.on('data', function(chunk) {
    console.log('in request length:', chunk.length);
    proxyRequest.write(chunk, 'binary');
  });

  req.on('end', function() {
    console.log('original request ended');
    proxyRequest.end();
  });
}).listen(8080);
```

1. Create standard HTTP server instance
2. Create request that copies the original request
3. Listen for data; then write it back to browser
4. Track when proxied request has finished
5. Send headers to the browser
6. Capture data sent from browser to the server
7. Track when original request ends
8. Listen for connections from local browsers

To use this example, your computer will need a bit of configuration. Find your system's internet options, and then look for HTTP proxies. From there you should be able to enter `localhost:8080` as the proxy. Alternatively, add the proxy in a browser's settings if possible. Some browsers don't support this; Google Chrome will open the system proxy dialog.

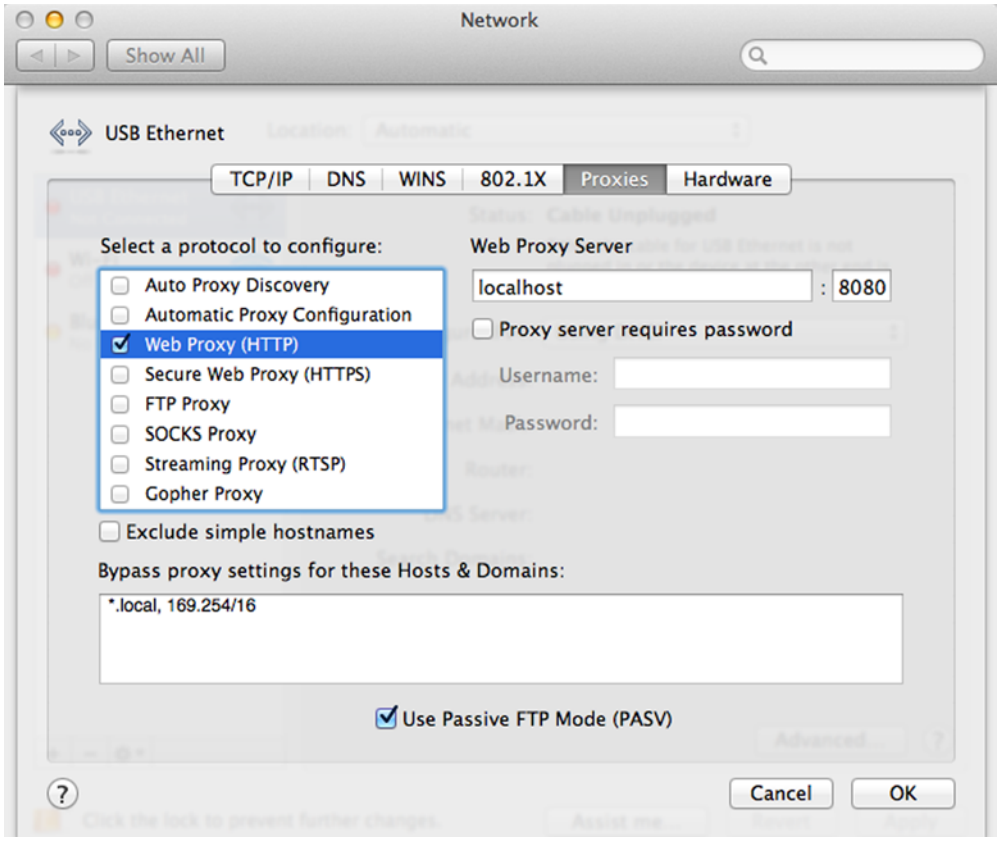


Figure 7.7 To use the Node proxy we've created, set `localhost:8080` as the Web Proxy Server.

Figure 7.7 shows how to configure the proxy on a Mac. Make sure you click OK and then Apply in the main *Network* dialog to save the setting. And remember to disable the proxy once you're done!

Once your system is set up to use the proxy, start the Node process up with `node listings/network/proxy.js` in a shell. Now when you visit web pages, you should see the successive requests and responses logged to the console.

This example works by first creating a server ❶ using the `http` module. The callback will be triggered when a browser makes a request. We've used `url.parse` (`url` is another core module) to separate out the URL's various parts so they can be passed as arguments to `http.request`. The parsed URL object is compatible with the arguments that `http.request` expects, so this is convenient ❷.

From within the request's callback, we can subscribe to events that need to be repeated back to the browser. The `data` event is useful because it allows us to capture the response from the server and pass it back to the client with `res.write` ❸. We also respond to the end of the server's connection by closing the connection to the

browser ❹. The status code is also written back to the client based on the server's response ❺.

Any data sent by the client is also proxied to the remote server by subscribing to the browser's `data` events ❻. Similarly, the browser's original request is watched for an `end` event so it can be reflected back to the proxied request ❼.

Finally, the HTTP server used as the proxy is set up to listen on port 8080 ❽.

This example creates a special server that sits between the browser and the server the browser wants to talk to. It could be extended to do lots of interesting things. For example, you could cache image files and compress them based on the remote client, sending mobile browsers heavily compressed images. You could even strip out certain content based on rules; some ad-blocking and parental filters work this way.

We've been using the DNS so far without really thinking about it too much. DNS uses TCP *and* UDP for its request/response-based protocol. Fortunately, Node hides this complexity for us with a slick asynchronous DNS module. The next section demonstrates how to make DNS requests using Node's `dns` module.

7.5 Making DNS requests

Node's DNS module lives outside of the `net` module, in `dns`. When the `http` or `net` modules are used to connect to remote servers, Node will look up IP addresses using `dns.lookup` internally.

TECHNIQUE 53 Making a DNS request

Node has multiple methods for making DNS requests. In this technique you'll learn how and why you should use each to resolve a domain name to an IP address.

When you query a DNS record, the results may include answers for different record types. The DNS is a distributed database, so it isn't used purely for resolving IP addresses—some records like `TXT` are used to build features off the back of the DNS itself.

Table 7.2 includes a list of each type along with the associated `dns` module method.

Table 7.2 DNS record types

Type	Method	Description
`A`	`dns.resolve`	An `A` record stores the IP address. It can have an associated time-to-live (TTL) field to indicate how often the record should be updated.
`TXT`	`dns.resolveTxt`	Text values that can be used by other services for additional features built on top of DNS.
`SRV`	`dns.resolveSrv`	Service records define "location" data for a service; this usually includes the port number and hostname.
`NS`	`dns.resolveNs`	Used for name servers themselves.
`CNAME`	`dns.resolveCname`	Canonical name records. These are set to domain names rather than IP addresses.

Problem

You want to look up a single or multiple domain names quickly.

Solution

The `dns.lookup` method can be used to look up either IPv4 or IPv6 addresses. When looking up multiple addresses, it can be faster to use `dns.resolve` instead.

Discussion

According to Node's documentation, `dns.lookup` is backed by a thread pool, whereas `dns.resolve` uses the `c-ares` library, which is faster. The `dns.lookup` API is a little friendlier—it uses `getaddrinfo`, which is more consistent with the other programs on your system. Indeed, the `Socket.prototype.connect` method, and any of Node's core modules that inherit from the objects in the `net` module, all use `dns.lookup` for consistency:

```
var dns = require('dns');                                    ← 1 Load dns module

dns.lookup('www.manning.com', function(err, address) {       ← 2 Look up IP address of the given domain
  if (err) {
    console.error('Error:', err);
  }
  console.log('Addresses:', address);
});
```

This example loads the `dns` module 1, and then looks up the IP address using `dns.lookup` 2. The API is asynchronous, so we have to pass a callback to receive the IP address and any errors that were raised when looking up the address. Note that the domain name has to be provided, rather than a URL—don't include `http://` here.

If everything runs correctly, then you should see `68.180.151.75` printed as the IP address. Conversely, if the previous example is run when you're offline, then a rather interesting error should be printed instead:

```
Error: {
  [Error: getaddrinfo ENOTFOUND]
  code: 'ENOTFOUND',                    ← 1 Error code
  errno: 'ENOTFOUND',
  syscall: 'getaddrinfo'                ← 2 System call where the error originated
}
```

The error object includes a standard error code 1 alongside the system call that raised the error 2. You can use the error code in your programs to detect when this kind of error was raised and handle it appropriately. The `syscall` property, meanwhile, is useful to us as programmers: it shows that the error was generated by a service outside of our Node code that is provided by the operating system.

Now compare this to the version that uses `dns.resolve`:

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

dns.resolve('www.manning.com', function(err, addresses) {   ← 1 Resolve domain name asynchronously.
  if (err) {
```

```
    console.error(err);
  }

  console.log('Addresses:', addresses);
});
```

The API looks similar to the previous example, apart from `dns.resolve` (1). You'll still see an error object that includes `ECONNREFUSED` if the DNS server couldn't be reached, but this time the result is different: we receive an array of addresses instead of a single result. In this example you should see `[ '68.180.151.75' ]`, but some servers may return more than one address.

Node's `dns` module is flexible, friendly, and fast. It can scale up well from infrequent single requests to making batches of requests.

The last part of Node's networking suite left to look at is perhaps the hardest to learn, yet paradoxically the most important to get right: encryption. The next section introduces SSL/TLS with the `tls` and `https` modules.

7.6 Encryption

Node's encryption module, `tls`, uses OpenSSL Transport Layer Security/Secure Socket Layer (TLS/SSL). This is a public key system, where each client and server both have a private key. The server makes its public key available so clients can encrypt subsequent communications in a way that only that server can decrypt again.

The `tls` module is used as the basis for the `https` module—this allows HTTP servers and clients to communicate over TLS/SSL. Unfortunately, TLS/SSL is a world of potential pitfalls. Node potentially supports different cyphers based on what version of OpenSSL it has been linked against. You can specify what cyphers you want to use when creating servers with `tls.createServer`, but we recommend using the defaults unless you have specific expertise in this area.

In the following technique you'll learn how to start a TCP server that uses SSL and a self-signed certificate. After that, we end the chapter with a technique that shows how encrypting web server communication works in Node.

TECHNIQUE 54 A TCP server that uses encryption

TLS can be used to encrypt servers made with `net.createServer`. This technique demonstrates how to do this by first creating the necessary certificates and then starting a client and server.

Problem

You want to encrypt communication sent and received over a TCP connection.

Solution

Use the `tls` module to start a client and server. Set up the required certificate files using OpenSSL.

Discussion

The main thing to master when working with encryption, whether it's web servers, mail servers, or any TCP-based protocol, is how to properly set up the key and certificate files.

Public key cryptography is dependent on public-private key pairs—a pair is required for both clients and servers. But an additional file is needed: the public key of the Certificate Authority (CA).

Our goal in this technique is to create a TLS client and server that both report `authorized` after the TLS handshake. This state is reported when *both* parties have verified each other's identity. When working with web server certificates, your CA will be the well-known organizations that commercially distribute certificates. But for the purposes of testing, you can become your own CA and sign certificates. This is also useful for secure communication between your own systems that don't need publicly verifiable certificates.

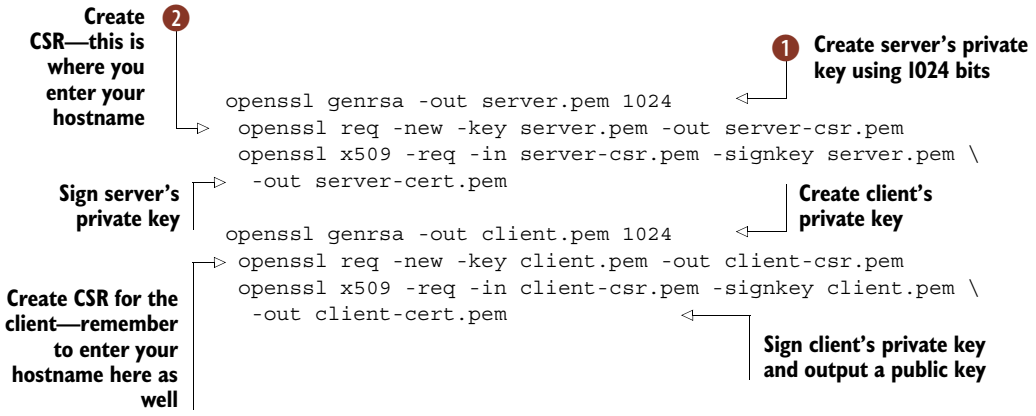
That means before you can run any Node examples, you'll need certificates. The OpenSSL command-line tools are required for this. If you don't have them, you should be able to install them with your operating system's package manager, or by visiting www.openssl.org.

The `openssl` tool takes a command as the first argument, and then options as subsequent arguments. For example, `openssl req` is used for X.509 Certificate Signing Request (CSR) management. To make a certificate signed by an authority you control, you'll need to issue the following commands:

- *`genrsa`*—Generate an RSA certificate; this is our private key.
- *`req`*—Create a CSR.
- *`x509`*—Sign the private key with the CSR to produce a public key.

When the process is broken down like this, it's fairly easy to understand: certificates require an authority and must be signed, and we need a public and private key. The process is similar when creating a public and private key signed against a commercial certificate authority, which you'll do if you want to buy certificates to use with public web servers.

The full command list for creating a public and private key is as follows:



After creating a private key 1, you'll create a CSR. When prompted for the "Common Name" 2, enter your computer's hostname, which you can find by typing `hostname` in

the terminal on a Unix system. This is important, because when your code sends or receives certificates, it'll check the name value against the `servername` property passed to the `tls.connect` method.

The next listing reads the server's keys and starts a server running using `tls.createServer`.

Listing 7.9 A TCP server that uses TLS for encryption

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

var options = {
  key: fs.readFileSync('server.pem'),
  cert: fs.readFileSync('server-cert.pem'),
  ca: [ fs.readFileSync('client-cert.pem') ],
  requestCert: true
};

var server = tls.createServer(options, function(cleartextStream) {
  var authorized = cleartextStream.authorized ?
    'authorized' : 'unauthorized';
  console.log('Connected:', authorized);
  cleartextStream.write('Welcome!\n');
  cleartextStream.setEncoding('utf8');
  cleartextStream.pipe(cleartextStream);
});

server.listen(8000, function() {
  console.log('Server listening');
});
```

1 Private key

Public key 2

3 Client as a certificate authority

Ensure client certificates are always checked 4

5 Whenever a client connects, show if server was able to verify the certificates

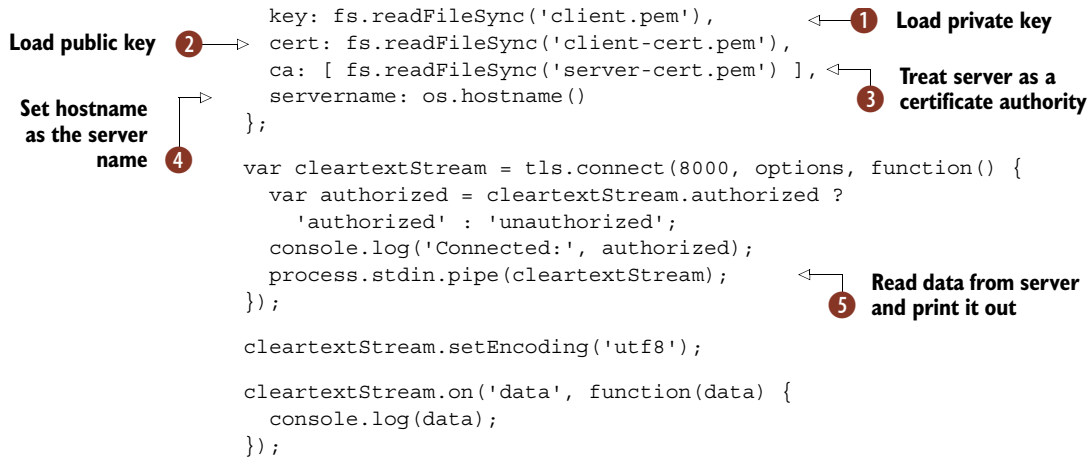
The network code in listing 7.9 is very similar to the `net.createServer` method—that's because the `tls` module inherits from it. The rest of the code is concerned with managing certificates, and unfortunately this process is left to us to handle and is often the cause of programmer errors, which can compromise security. First we load the private 1 and public 2 keys, passing them to `tls.createServer`. We also load the client's public key as a certificate authority 3—when using a commercially obtained certificate, this stage isn't usually required.

When clients connect, we want to send them some data, but for the purposes of this example we really just want to see if the client was authorized 5. Client authorization has been forced by setting the `requestCert` option 4.

This server can be run with `node tls.js`—but there's something missing: a client! The next listing contains a client that can connect to this server.

Listing 7.10 A TCP client that uses TLS

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

var options = {
```




```
  key: fs.readFileSync('client.pem'),
  cert: fs.readFileSync('client-cert.pem'),
  ca: [ fs.readFileSync('server-cert.pem') ],
  servername: os.hostname()
};

var cleartextStream = tls.connect(8000, options, function() {
  var authorized = cleartextStream.authorized ?
    'authorized' : 'unauthorized';
  console.log('Connected:', authorized);
  process.stdin.pipe(cleartextStream);
});

cleartextStream.setEncoding('utf8');

cleartextStream.on('data', function(data) {
  console.log(data);
});
```

The client is similar to the server: the private 1 and public keys 2 are loaded, and this time the server is treated as the CA 3. The server's name is set to the same value as the Common Name in the CSR by using `os.hostname` 4—you could type in the name manually if you set it to something else. After that the client connects, displays whether it was able to authorize the certificates, and then reads data sent by the server and pipes it to the standard output 5.

Testing SSL/TLS

When testing secure certificates, it can be hard to tell whether the problem lies in your code or elsewhere. One way around this is to use the `openssl` command-line tool to simulate a client or server. The following command will start a client that connects to a server with the given certificate file:

```
openssl s_client -connect 127.0.0.1:8000 \
  -CAfile ./server-cert.pem
```

The `openssl` tool will display a lot of extra information about the connection. When we wrote the example in this technique, we used it to figure out that the certificate we'd generated had the wrong value for its Common Name.

An instance of `tls.Server` is instantiated when you call `tls.createServer`. This constructor calls `net.Server`—there's a clear inheritance chain between each networking module. That means the events emitted by `net.Server` are the same for TLS servers.

In the next technique you'll see how to use HTTPS, and how this is also related to the `tls` and `net` modules.

TECHNIQUE 55 Encrypted web servers and clients

Though it's possible to host Node applications behind other web servers like Apache and nginx, there are times when you'll want to run your own HTTPS servers. This technique introduces the `https` module and shows how it's related to the `tls` module.

■ **Problem**

You want to run a server that supports SSL/TLS.

■ **Solution**

Use the `https` module and `https.createServer`.

■ **Discussion**

To run the examples in this technique, you'll need to have followed the steps to create suitable self-signed certificates, as found in technique 54. Once you've set up some public and private keys, you'll be able to run the examples.

The following listing shows an HTTPS server.

Listing 7.11 A basic HTTP server that uses TLS for encryption

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

var options = {
  key: fs.readFileSync('server.pem'),                    // 1 Private key
  cert: fs.readFileSync('server-cert.pem'),              // Public key 2
  ca: [ fs.readFileSync('client-cert.pem') ],
  requestCert: true                                      // Ensure client certificates are always checked
};

var server = https.createServer(options, function(req, res) {
  var authorized = req.socket.authorized                 // When a browser requests a page, show if server was able to verify the certificates
    ? 'authorized' : 'unauthorized';
  res.writeHead(200);
  res.write('Welcome! You are ' + authorized + '\n');
  res.end();
});

server.listen(8000, function() {
  console.log('Server listening');
});
```

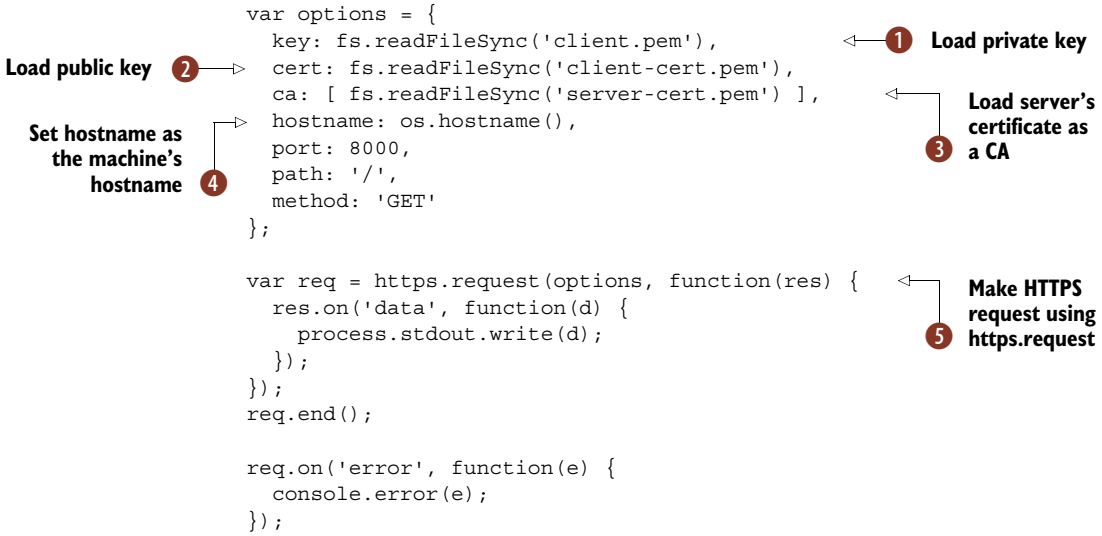
The server in listing 7.11 is basically the same as the one in technique 54. Again, the private 1 and public 2 keys are loaded and passed to `https.createServer`.

When browsers request a page, we check the `req.socket.authorized` property to see if the request was authorized. This status is returned to the browser. If you want to try this out with a browser, ensure you type `https://` into the address bar; otherwise it won't work. You'll see a warning message because the browser won't be able to verify the server's certificate—that's OK; you know what's going on because you created the server. The server will respond saying that you're *unauthorized* because it won't be able to authorize you, either.

To make a client that can connect to this server, follow the code shown next.

Listing 7.12 An example HTTPS client

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




```
var options = {
  key: fs.readFileSync('client.pem'),
  cert: fs.readFileSync('client-cert.pem'),
  ca: [ fs.readFileSync('server-cert.pem') ],
  hostname: os.hostname(),
  port: 8000,
  path: '/',
  method: 'GET'
};

var req = https.request(options, function(res) {
  res.on('data', function(d) {
    process.stdout.write(d);
  });
});
req.end();

req.on('error', function(e) {
  console.error(e);
});
```

This example sets the private 1 and public 2 keys for the client, which is what your browser does transparently when making secure requests. It also sets the server as a certificate authority 3, which wouldn't usually be required. The hostname used for the HTTP request is the machine's current hostname 4.

Once all of this setup is done, the HTTPS request can be made. This is done using `https.request` 5. The API is identical to the `http` module. In this example the server will ensure the SSL/TLS authorization procedure was valid, so the server will return text to indicate if the connection was fully authorized.

In real HTTPS code, you probably wouldn't make your own CA. This can be useful if you have internal systems that you want to communicate with using HTTPS—perhaps for testing or for API requests over the internet. When making HTTPS requests against public web servers, Node will be able to verify the server's certificates for you, so you won't need to set the `key`, `cert`, and `ca` options.

The `https` module has some other features—there's an `https.get` convenience method for making `GET` requests more easily. Otherwise, that wraps up our set of techniques on encryption in Node.

Secure pairs

Before moving off encryption for greener pastures, there's one patch of delicious turf left to chew: `SecurePair`. This is a class in the `tls` module that can be used to create a secure pair of streams: one reads and writes encrypted data, and the other reads and writes clear text. This potentially allows you to stream anything to an encrypted output.

There's a convenience method for this: `tls.createSecurePair`. When a `SecurePair` establishes a secure connection, it'll emit a `secure` event, but you'll still need to check for `cleartext.authorized` to ensure the certificates were properly authorized.

7.7 Summary

This chapter has been long, but that's because networking in Node is important. Node is built on excellent foundations for network programming; buffers, streams, and asynchronous I/O all contribute to an environment that is perfect for writing the next generation of network-oriented programs.

With this chapter you should be able to appreciate how Node fits into the wider world of network software. Whether you're developing Unix daemons, Windows-based game servers, or the next big web app, you should now know where to start.

It goes without saying that networking and encryption are closely related. With Node's `tls` and `https` modules, you should be able to write network clients and servers that can talk to other systems without fear of eavesdroppers.

The next chapter is the last on Node's core modules, `child_process`, and looks at techniques for interfacing with other command-line programs.

8 Child processes: Integrating external applications with Node

This chapter covers

- Executing external applications
- Detaching a child process
- Interprocess communication between Node processes
- Making Node programs executable
- Creating job pools
- Synchronous child processes

No platform is an island. Although it would be fun to write everything in JavaScript, we'd miss out on valuable applications that already exist in other platforms. Take GraphicsMagick, for instance (http://www.graphicsmagick.org/): a full-featured image manipulation tool, great for resizing that massively large profile photo that was just uploaded. Or take wkhtmltopdf (http://wkhtmltopdf.org/), a headless webkit PDF generator, perfect for turning that HTML report into a PDF download.

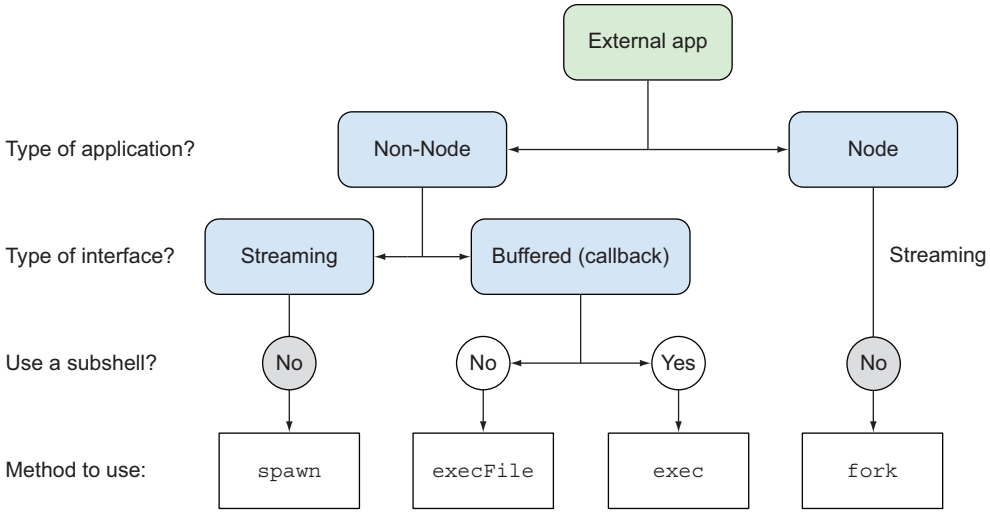


Figure 8.1 Choosing the right method

In Node, the `child_process` module allows us to execute these applications and others (including Node applications) to use with our programs. Thankfully, we don't have to re-invent the wheel.

The `child_process` module provides four different methods for executing external applications. All methods are asynchronous. The right method will depend on what you need, as shown in figure 8.1.

- *`execFile`*—Execute an external application, given a set of arguments, and callback with the buffered output after the process exits.
- *`spawn`*—Execute an external application, given a set of arguments, and provide a streaming interface for I/O and events for when the process exits.
- *`exec`*—Execute one or more commands inside a shell and callback with the buffered output after the process exits.
- *`fork`*—Execute a Node module as a separate process, given a set of arguments, provide a streaming and evented interface like `spawn`, and also set up an interprocess communication (IPC) channel between the parent and child process.

Throughout this chapter we'll dive into how to get the most out of these methods, giving practical examples of where you'd want to use each. Later on, we'll look into some other techniques to use when working with child processes: detaching processes, interprocess communication, file descriptors, and pooling.

8.1 Executing external applications

In this first section, we will look at all the ways you can work asynchronously with an external program.

TECHNIQUE 56 Executing external applications

Wouldn't it be great to run some image processing on a user's uploaded photo with ImageMagick, or validate an XML file with xmllint? Node makes it easy to execute external applications.

Problem

You want to execute an external application and get the output.

Solution

Use `execFile` (see figure 8.2).

Discussion

If you want to run an external application and get the result, using `execFile` makes it simple and straightforward. It'll buffer the output for you and provide the results and any errors in a callback. Let's say we want to run the `echo` program given the parameters `hello world`. With `execFile`, we would do the following:

```
var cp = require('child_process');

cp.execFile('echo', ['hello', 'world'],
   function (err, stdout, stderr) {
     if (err) console.error(err);
     console.log('stdout', stdout);
     console.log('stderr', stderr);
   });
```

Provide command as first parameter and any command arguments as an array for second parameter

Callback includes any error executing the command and buffered output from stdout and stderr

How does Node know where to find the external application? To answer that, we need to look at how paths work in the underlying operating system.

8.1.1 Paths and the PATH environment variable

Windows/UNIX has a `PATH` environment variable (envvar: http://en.wikipedia.org/wiki/PATH_(variable)). `PATH` contains a list of directories where executable programs exist. If a program exists in one of the listed directories, it can be located without needing an absolute or relative path to the application.

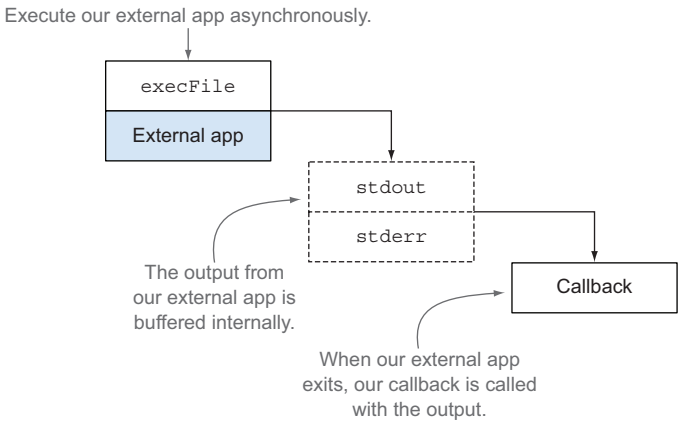


Figure 8.2 The `execFile` method buffers the result and provides a callback interface.

Node, using execvp behind the scenes, will search for applications using PATH when no absolute or relative location is provided. We can see this in our earlier example, since directories to common system applications like echo usually exist in PATH already.

If the directory containing the application isn't in PATH, you'll need to provide the location explicitly like you would on the command line:

```
cp.execFile('./app-in-this-directory' ...
cp.execFile('/absolute/path/to/app' ...
cp.execFile('../relative/path/to/app' ...
```

To see what directories are listed in PATH, you can run a simple one-liner in the Node REPL:

```
$ node
> console.log(process.env.PATH.split(':').join('\n'))
/usr/local/bin
/usr/bin/bin
…
```

If you want to avoid including the location to external applications not in PATH, one option is to add any new directories to PATH inside your Node application. Just add this line before any execFile calls:

```
process.env.PATH += ':/a/new/path/to/executables';
```

Now any applications in that new directory will be accessible without providing a path to execFile.

8.1.2 Errors when executing external applications

If your external application doesn't exist, you'll get an ENOENT error. Often this is due to a typo in the application name or path with the result that Node can't find the application, as shown in figure 8.3.

If the external application does exist but Node can't access it (typically due to insufficient permissions), you'll get an EACCES or EPERM error. This can often be mitigated by either running your Node program as a user with sufficient permissions or changing the external application permissions themselves to allow access.

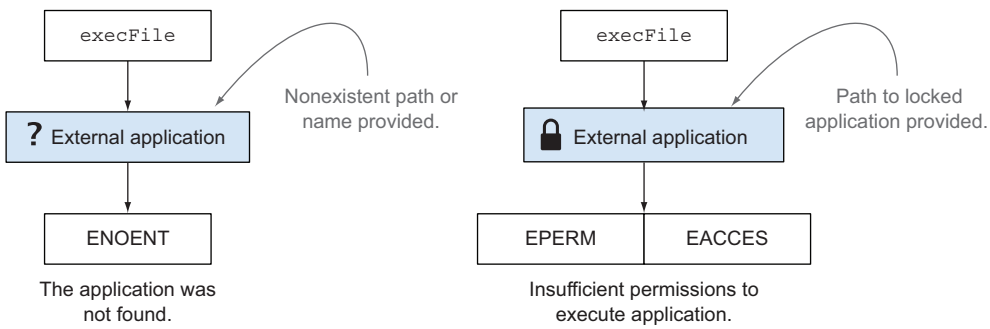


Figure 8.3 Common child process errors

You'll also get an error if the external application has a non-zero exit status (http://mng.bz/MLXP), which is used to indicate that an application couldn't perform the task it was given (on both UNIX and Windows). Node will provide the exit status as part of the error object and will also provide any data that was written to stdout or stderr:

```
var cp = require('child_process');
cp.execFile('ls', ['non-existent-directory-to-list'],
  function (err, stdout, stderr) {
    console.log(err.code);
     console.log(stderr);
 });
```

Output exit code, which is 1 in this case, indicating command failed

Output error details stored in stderr

Having `execFile` is great for when you want to just execute an application and get the output (or discard it), for example, if you want to run an image-processing command with ImageMagick and only care if it succeeds or not. But if an application has a lot of output or you want to do more real-time analysis of the data returned, using streams is a better approach.

TECHNIQUE 57 Streaming and external applications

Imagine a web application that uses the output from an external application. As that data is being made available, you can at the same time be pushing it out to the client. Streaming enables you to tap into the data from a child process as it's being outputted, versus having the data buffered and then provided. This is good if you expect the external application to output large amounts of data. Why? Buffering a large set of data can take up a lot of memory. Also, this enables data to be consumed as it's being made available, which improves responsiveness.

Problem

You want to execute an external application and stream the output.

Solution

Use `spawn` (see figure 8.4).

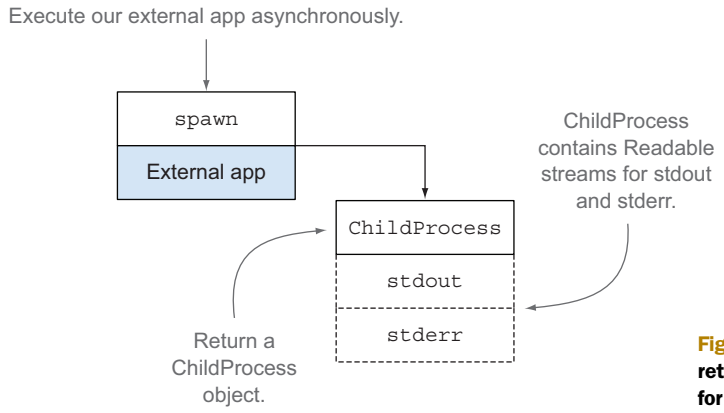


Figure 8.4 The `spawn` method returns a streaming interface for I/O.

Discussion

The `spawn` method has a function signature similar to `execFile`:

```
cp.execFile('echo', ['hello', 'world'], ...);
cp.spawn('echo', ['hello', 'world'], ...);
```

The application is the first argument, and an array of parameters/flags for the application is the second. But instead of taking a callback providing the output already buffered, `spawn` relies on streams:

```
var cp = require('child_process');

var child = cp.spawn('echo', ['hello', 'world']);
child.on('error', console.error);
child.stdout.pipe(process.stdout);
child.stderr.pipe(process.stderr);
```

Spawn method returns a ChildProcess object containing stdin, stdout, and stderr stream objects

Errors are emitted on error event

Output from stdout and stderr can be read as it's available

Since `spawn` is stream-based, it's great for handling large outputs or working with data as it's read in. All other benefits of streams apply as well. For example, `child.stdin` is a `Writeable` stream, so you can hook that up to any `Readable` stream to get data. The reverse is true for `child.stdout` and `child.stderr`, which are `Readable` streams that can be hooked into any `Writeable` stream.

> **API SYMMETRY** The `ChildProcess` API (`child.stdin`, `child.stdout`, `child.stderr`) share a nice symmetry with the parent `process` streams (`process.stdin`, `process.stdout`, `process.stderr`).

8.1.3 Stringing external applications together

A large part of UNIX philosophy is building applications that do one thing and do it well, and then communicating between those applications with a common interface (that being plain text).

Let's make a Node program that exemplifies this by taking three simple applications that deal with text streams and sticking them together using `spawn`. The `cat` application will read a file and output its contents. The `sort` application will take in the file as input and provide the lines sorted as output. The `uniq` application will take the sorted file as input, and output the sorted file with all the duplicate lines removed. This is illustrated in figure 8.5.

Let's look at how we can do this with `spawn` and streams:

```
var cp = require('child_process');
var cat = cp.spawn('cat', ['messy.txt']);
var sort = cp.spawn('sort');
var uniq = cp.spawn('uniq');

cat.stdout.pipe(sort.stdin);
sort.stdout.pipe(uniq.stdin);
uniq.stdout.pipe(process.stdout);
```

Call spawn for each command we want to chain together

Stream result to the console with process.stdout

Output of each command becomes input for next command

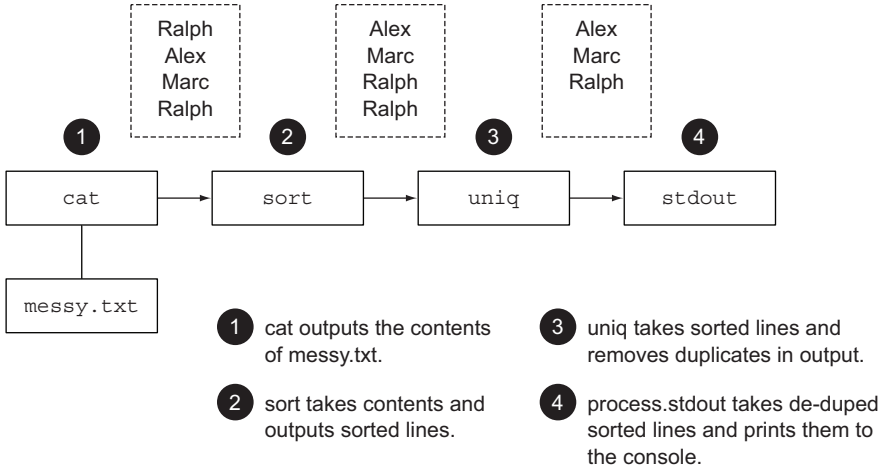


Figure 8.5 Stringing external applications together with `spawn`

Using `spawn`'s streaming interfaces allows a seamless way to work with any stream objects in Node, including stringing external applications together. But sometimes we need the facilities of our underlying shell to do powerful composition of external applications. For that, we can use `exec`.

> **APPLYING WHAT YOU'VE LEARNED** Can you think of a way to avoid using the `cat` program based on what you learned with the `fs` module and streaming in chapter 6?

TECHNIQUE 58 Executing commands in a shell

Shell programming is a common way to build utility scripts or command-line applications. You could whip up a Bash or Python script, but with Node, you can use JavaScript. Although you could execute a subshell manually using `execFile` or `spawn`, Node provides a convenient, cross-platform method for you.

Problem

You need to use the underlying shell facilities (like pipes, redirects, file blobs) to execute commands and get the output.

Solution

Use `exec` (see figure 8.6).

Discussion

If you need to execute commands in a shell, you can use `exec`. The `exec` method runs the commands with `/bin/sh` or `cmd.exe` (on Windows). Running commands in a shell means you have access to all the functionality provided by your particular shell (like pipes, redirects, and backgrounding).

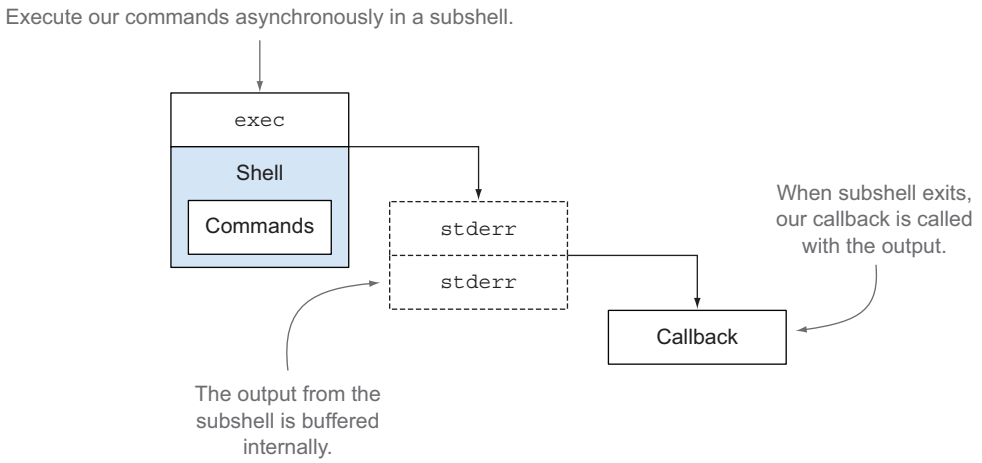


Figure 8.6 The `exec` method runs our commands in a subshell.

A SINGLE COMMAND ARGUMENT Unlike `execFile` and `spawn`, the `exec` method doesn't have a separate argument for command parameters/flags, since you can run more than one command on a shell.

As an example, let's pipe together the same three applications we did in the last technique to generate a sorted, unique list of names. But this time, we'll use common UNIX shell facilities rather than streams:

```
cp.exec('cat messy.txt | sort | uniq',
    function (err, stdout, stderr) {
      console.log(stdout);
    });
```

If successful, stdout will contain sorted, de-duped version of messy.txt

Pipe cat, sort, and uniq together like we would on command line

ABOUT SHELLS UNIX users should keep in mind that Node uses whatever is mapped to `/bin/sh` for execution. This typically will be Bash on most modern operating systems, but you have the option to remap it to another shell of your liking. Windows users who need a piping facility can use streams and spawn as discussed in technique 57.

8.1.4 Security and shell command execution

Having access to a shell is powerful and convenient, but it should be used cautiously, especially with a user's input.

Let's say we're using xmllint (http://xmlsoft.org/xmllint.html) to parse and detect errors in a user's uploaded XML file where the user provides a schema to validate against:

```
cp.exec('xmllint --schema '+req.query.schema+' the.xml');
```

If a user provided "http://site.com/schema.xsd," it would be replaced and the following command would run:

```
xmllint --schema http://site.com/schema.xsd the.xml
```

But since the argument has user input, it can easily fall prey to command (or shell) injection attacks (https://golemtechnologies.com/articles/shell-injection)—for example, a malicious user provides "; rm -rf / ;" causing the following comment to run (*please don't run this in your terminal!*):

```
xmllint --schema ; rm -rf / ; the.xml
```

If you haven't guessed already, this says, "Start new command (`;`), remove forcibly and recursively all files/directories at root of the file system (`rm -rf /`), and end the command (`;`) in case something follows it."

In other words, this injection could potentially delete all the files the Node process has permission to access on the entire operating system! And that's just one of the commands that can be run. Anything your process user has access to (files, commands, and so on) can be exploited.

If you need to run an application and don't need shell facilities, it's safer (and slightly faster) to use `execFile` instead:

```
cp.execFile('xmllint', ['--schema', req.query.schema, 'the.xml']);
```

Here this malicious injection attack would fail since it's not run in a shell and the external application likely wouldn't understand the argument and would raise an error.

TECHNIQUE 59 Detaching a child process

Node can be used to kick off external applications and then allow them to run on their own. For example, let's say you have an administrative web application in Node that allows you to kick off a long-running synchronization process with your cloud provider. If that Node application were to crash, your synchronization process would be halted. To avoid this, you detach your external application so it'll be unaffected.

Problem

You have a long-running external application that you want Node to start but then be able to exit with the child process still running.

Solution

Detach a spawned child process (see figure 8.7).

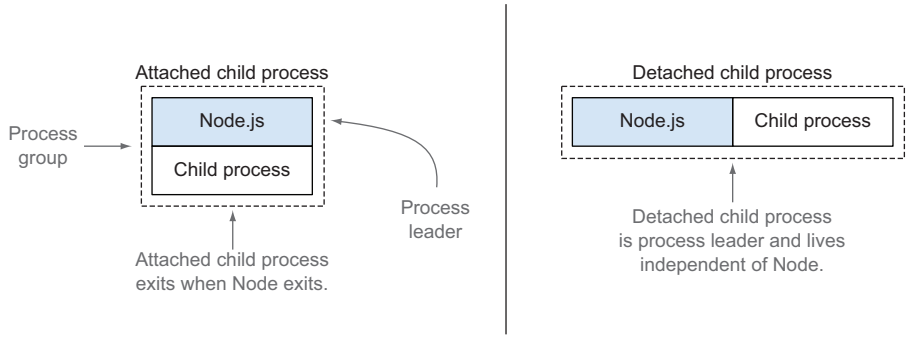


Figure 8.7 Detached child process exists independent of the Node process

Discussion

Normally, any child process will be terminated when the parent Node process is terminated. Child processes are said to be *attached* to the parent process. But the `spawn` method includes the ability to *detach* a child process and promote it to be a process group leader. In this scenario, if the parent is terminated, the child process will continue until finished.

This scenario is useful when you want Node to set up the execution of a long-running external process and you don't need Node to babysit it after it starts.

This is the `detached` option, configurable as part of a third options parameter to `spawn`:

```
var child = cp.spawn('./longrun', [], { detached: true });
```

In this example, `longrun` will be promoted to a process group leader. If you were to run this Node program and forcibly terminate it (Ctrl-C), `longrun` would continue executing until finished.

If you didn't forcibly terminate, you'd notice that the parent stays alive until the child has completed. This is because I/O of the child process is connected to the parent. In order to disconnect the I/O, you have to configure the `stdio` option.

8.1.5 Handing I/O between the child and parent processes

The `stdio` option defines where the I/O from a child process will be redirected. It takes either an array or a string as a value. The string values are simply shorthands that will expand to common array configurations.

The array is structured such that the *indexes* correspond to file descriptors in the child process and the *values* indicate where the I/O for the particular file descriptor (FD) should be redirected.

> **WHAT ARE FILE DESCRIPTORS?** If you're confused about file descriptors, check out technique 40 in chapter 6 for an introduction.

By default, `stdio` is configured as

```
stdio: 'pipe'
```

which is a shorthand for the following array values:

```
stdio: [ 'pipe', 'pipe', 'pipe' ]
```

This means that file descriptors 0-2 will be made accessible on the `ChildProcess` object as streams (`child.stdio[0]`, `child.stdio[1]`, `child.stdio[2]`). But since FDs 0-2 often refer to `stdin`, `stdout`, and `stderr`, they're also made available as the now familiar `child.stdin`, `child.stdout`, and `child.stderr` streams.

The `pipe` value connects the parent and child processes because these streams stay open, waiting to write or read data. But for this technique, we want to disconnect the two in order to exit the Node process. A brute-force approach would be to simply destroy all the streams created:

```
child.stdin.destroy();
child.stdout.destroy();
child.stderr.destroy();
```

Although this would work, given our intent to not use them, it's better to not create the streams in the first place. Instead, we can assign a file descriptor if we want to direct the I/O elsewhere or use `ignore` to discard it completely.

Let's look at a solution that uses both options. We want to `ignore` FD 0 (`stdin`) since we won't be providing any input to the child process. But let's capture any output from FDs 1 and 2 (`stdout`, `stderr`) just in case we need to do some debugging later on. Here's how we can accomplish that:

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

var outFd = fs.openSync('./longrun.out', 'a');
var errFd = fs.openSync('./longrun.err', 'a');

var child = cp.spawn('./longrun', [], {
  detached: true,
  stdio: [ 'ignore', outFd, errFd ]
});
```

Open two log files, one for stdout and one for stderr

Ignore FD 0; redirect output from FDs I and 2 to the log files

This will disconnect the I/O between the child and parent processes. If we run this application, the output from the child process will end up in the log files.

8.1.6 Reference counting and child processes

We're almost there. The child process will live on because it's detached and the I/O is disconnected from the parent. But the parent still has an internal reference to the child process and won't exit until the child process has finished and the reference has been removed.

You can use the `child.unref()` method to tell Node not to include this child process reference in its count. The following complete application will now exit after spawning the child process:

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

var outFd = fs.openSync('./longrun.out', 'a');
var errFd = fs.openSync('./longrun.err', 'a');

var child = cp.spawn('./longrun', [], {
  detached: true,
  stdio: [ 'ignore', outFd, errFd ]
});

child.unref();
```

Remove reference of child in the parent process

To review, detaching a process requires three things:

- The `detached` option must be set to `true` so the child becomes its own process leader.
- The `stdio` option must be configured so the parent and child are disconnected.
- The reference to the child must be severed in the parent using `child.unref()`.

8.2 Executing Node programs

Any of the prior techniques can be used to execute Node applications. However, in the techniques to follow, we will focus on making the most out of Node child processes.

TECHNIQUE 60 Executing Node programs

When writing shell scripts, utilities, or other command-line applications in Node, it's often handy to make executables out of them for ease of use and portability. If you publish command-line applications to npm, this also comes in handy.

Problem

You want to make a Node program an executable script.

Solution

Set up the file to be executable by your underlying platform.

Discussion

A Node program can be run as a child process with any of the means we've already described by simply using the `node` executable:

```
var cp = require('child_process');
cp.execFile('node', ['myapp.js', 'myarg1', 'myarg2' ], ...
```

But there are many cases where having a standalone executable is more convenient, where you can instead use your app like this:

```
myapp myarg1 myarg2
```

The process for making an executable will vary depending on whether you're on Windows or UNIX.

Executables on Windows

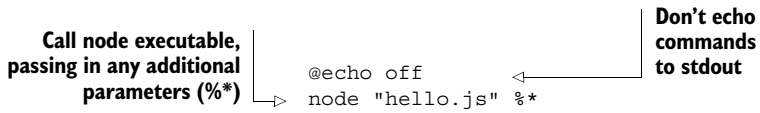
Let's say we have a simple one-liner `hello.js` program that echoes the first argument passed:

```
console.log('hello', process.argv[2]);
```

To run this program, we type

```
$ node hello.js marty
hello marty
```

To make a Windows executable, we can make a simple batch script calling the Node program. For consistency, let's call it `hello.bat`:



Now we can execute our hello.js program by simply running the following:

```
$ hello tom
hello tom
```

Running it as a child process requires the `.bat` extension:

```
var cp = require('child_process');
cp.execFile('hello.bat', ['billy'], function (err, stdout) {
  console.log(stdout); // hello billy
});
```

Executables on UNIX

To turn a Node program into an executable script on most UNIX systems, we don't need a separate batch file like in Windows; we simply modify `hello.js` itself by adding the following to the top of the file:

```
#!/usr/bin/env node
console.log('hello', process.argv[2]);
```

Execute node command wherever it's found in user's environment

Then to actually make the file executable, we run the following command:

```
$ chmod +x hello.js
```

We can then run the command like this:

```
$ ./hello.js jim
hello jim
```

The file can be renamed as well to look more like a standalone program:

```
$ mv hello.js hello
$ ./hello jane
hello jane
```

Executing this program as a child process will look the same as its command-line counterpart:

```
var cp = require('child_process');
cp.execFile('./hello', ['bono'], function (err, stdout) {
  console.log(stdout); // hello bono
});
```

PUBLISHING EXECUTABLE FILES IN NPM For publishing packages that contain executable files, use the UNIX conventions, and npm will make the proper adjustments for Windows.

TECHNIQUE 61 Forking Node modules

Web workers (http://mng.bz/UG63) provide the browser and JavaScript an elegant way to run computationally intense tasks off the main thread with a built-in communication stream between the parent and worker. This removes the painful work of breaking up computation into pieces in order to not upset the user experience. In Node, we have the same concept, with a slightly different API with `fork`. This helps

us break out any heavy lifting into a separate process, keeping our event loop running smoothly.

■ **Problem**

You want to manage separate Node processes.

■ **Solution**

Use `fork` (see figure 8.8).

■ **Discussion**

Sometimes it's useful to have separate Node processes. One such case is computation. Since Node is single-threaded, computational tasks directly affect the performance of the whole process. This may be acceptable for certain jobs, but when it comes to network programming, it'll severely affect performance since requests can't be serviced when the process is tied up. Running these types of tasks in a forked process allows the main application to stay responsive. Another use of forking is for sharing file descriptors, where a child can accept an incoming connection received by the parent process.

Node provides a nice way to communicate between other Node programs. Under the hood, it sets up the following `stdio` configuration:

```
stdio: [ 0, 1, 2, 'ipc' ]
```

This means that, by default, all output and input are directly inherited from the parent; there's no `child.stdin`, `child.stdout`, or `child.stderr`:

```
var cp = require('child_process');
var child = cp.fork('./myChild');
```

If you want to provide an I/O configuration that behaves like the `spawn` defaults (meaning you get a `child.stdin`, and so on), you can use the `silent` option:

```
var cp = require('child_process');
var child = cp.fork('./myChild', { silent: true });
```

INTERNALS OF INTERPROCESS COMMUNICATION Although a number of mechanisms exist to provide interprocess communication (IPC; see http://mng.bz/LGKD), Node IPC channels will use either a UNIX domain socket (http://mng.bz/1189) or a Windows named pipe (http://mng.bz/262Q).

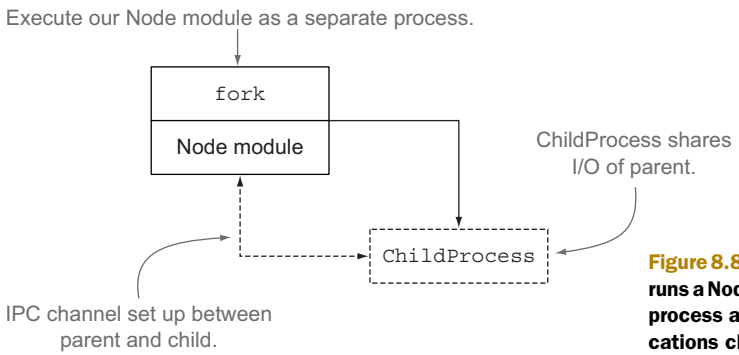
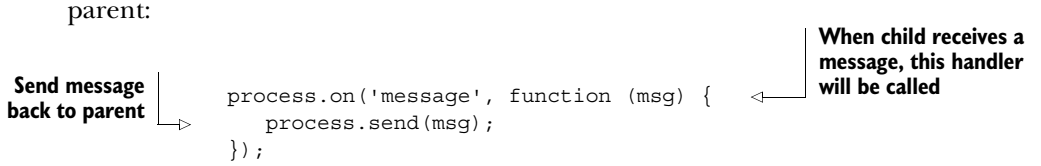


Figure 8.8 The `fork` command runs a Node module in a separate process and sets up a communications channel.

Communicating with forked Node modules

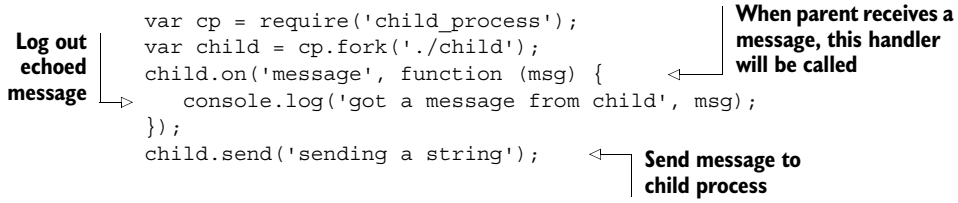
The `fork` method opens up an IPC channel that allows message passing between Node processes. On the child side, it exposes `process.on('message')` and `process.send()` as mechanisms for receiving and sending messages. On the parent side, it provides `child.on('message')` and `child.send()`.

Let's make a simple echo module that sends back any message received from the parent:




```
process.on('message', function (msg) {
  process.send(msg);
});
```

An application can now consume this module using `fork`:




```
var cp = require('child_process');
var child = cp.fork('./child');
child.on('message', function (msg) {
  console.log('got a message from child', msg);
});
child.send('sending a string');
```

Sending data between the processes maintains the type information, which means you can send any valid JSON value over the wire and it retains the type:

```
child.send(230);
child.send('a string');
child.send(true);
child.send(null);
child.send({ an: 'object' });
```

Disconnecting from forked Node modules

Since we're opening an IPC channel between the parent and child, both stay alive until the child is disconnected (or exits some other way). If you need to disconnect the IPC channel, you can do that explicitly from the parent process:

```
child.disconnect();
```

TECHNIQUE 62 Running jobs

When you need to run *routine* computational jobs, forking processes on demand will quickly eat up your CPU resources. It's better to keep a job pool of available Node processes ready for work. This technique takes a look at that.

- **Problem**

  You have routine jobs that you don't want to run on the main event loop.

- **Solution**

  Use `fork` and manage a pool of workers.

- **Discussion**

  We can use the IPC channel built into `fork` to create a pattern for handling computationally intensive tasks (or jobs). It builds upon our last technique, but adds an important

constraint: when the parent sends a task to the child, it expects to receive exactly one result. Here's how this works in the parent process:

```
function doWork (job, cb) {
  var child = cp.fork('./worker');
  child.send(job);
  child.once('message', function (result) {
     cb(null, result);
  });
}
```

Send job to the child process

Expect child to respond with exactly one message providing the result

But receiving a result is only one of the possible outcomes. To build resilience into our `doWork` function, we'll account for

- The child exiting for any reason
- Unexpected errors (like a closed IPC channel or failure to fork)

Handling those in code will involve a couple more listeners:

```
child.once('error', function (err) {
  cb(err);
  child.kill();
 });
child.once('exit', function (code, signal) {
  cb(new Error('Child exited with code: ' + code));
});
```

Unexpected error; kill the process as it's likely unusable

This is a good start, but we run the risk of calling our callback more than once in the case where the worker finished the job but then later exited or had an error. Let's add some state and clean things up a bit:

```
function doWork (job, cb) {
  var child = cp.fork('./worker');
  var cbTriggered = false;

  child
    .once('error', function (err) {
      if (!cbTriggered) {
        cb(err);
        cbTriggered = true;
      }
      child.kill();
    })
    .once('exit', function (code, signal) {
      if (!cbTriggered)
        cb(new Error('Child exited with code: ' + code));
    })
    .once('message', function (result) {
      cb(null, result);
      cbTriggered = true;
    })
    .send(job);
}
```

Track if callback was called

Message will never be triggered if an error or exit happens, so we don't need cbTriggered check

So far we've only looked at the parent process. The child worker takes in a job, and sends exactly one message back to the parent when completed:

```
process.on('message', function (job) {
  // do work
  process.send(result);
});
```

8.2.1 Job pooling

Currently, our `doWork` function will spin up a new child process every time we need to do some work. This isn't free, as the Node documentation states:

> *These child Nodes are still whole new instances of V8. Assume at least 30ms startup and 10mb memory for each new Node. That is, you cannot create many thousands of them.*

A performant way to work around this is not to spin off a new process whenever you want to do something computationally expensive, but rather to maintain a pool of long-running processes that can handle the load.

Let's expand our `doWork` function, creating a module for handling a worker pool. Here are some additional constraints we'll add:

- Only fork up to as many worker processes as CPUs on the machine.
- Ensure new work gets an available worker process and not one that's currently in-process.
- When no worker processes are available, maintain a queue of tasks to execute as processes become available.
- Fork processes on demand.

Let's take a look at the code to implement this:

```
var cp = require('child_process');
var cpus = require('os').cpus().length; )

module.exports = function (workModule) {
  var awaiting = [];
  var readyPool = [];
  var poolSize = 0;

  return function doWork (job, cb) {
    if (!readyPool.length && poolSize > cpus)
      return awaiting.push([ doWork, job, cb ]);

    var child = readyPool.length
      ? readyPool.shift()
      : (poolSize++, cp.fork(workModule));
    var cbTriggered = false;

    child
```

Grab number of CPUs

Keep list of tasks that are queued to run when all processes are in use

Keep list of worker processes that are ready for work

Keep track of how many worker processes exist

If no worker processes are available and we've reached our limit, queue work to be run later

Grab next available child, or fork a new process (incrementing the poolSize)

```
      .removeAllListeners()
      .once('error', function (err) {
        if (!cbTriggered) {
          cb(err);
          cbTriggered = true;
        }
        child.kill();
      })
      .once('exit', function () {
        if (!cbTriggered)
          cb(new Error('Child exited with code: ' + code));
        poolSize--;
        var childIdx = readyPool.indexOf(child);
        if (childIdx > -1) readyPool.splice(childIdx, 1);
      })
      .once('message', function (msg) {
        cb(null, msg);
        cbTriggered = true;
        readyPool.push(child);
        if (awaiting.length) setImmediate.apply(null, awaiting.shift());
      })
      .send(job);
  }
}
```

Remove any listeners that exist on child, ensuring that a child process will always have only one listener attached for each event at a time

If child exits for any reason, ensure it's removed from the readyPool

Child is ready again; add back to readyPool and run next awaiting task (if any)

APPLYING WHAT YOU'VE LEARNED Other constraints may apply depending on the needs of the pool, for example, retrying jobs on failure or killing long-running jobs. How would you implement a retry or timeout using the preceding example?

8.2.2 Using the pooler module

Let's say we want to run a computationally intensive task based on a user's request to our server. First, let's expand our child worker process to simulate an intensive task:

```
process.on('message', function (job) {
  for (var i = 0; i < 1000000000; i++);
  process.send('finished: ' + job);
});
```

Actual work happens here; in our case, we'll simply generate a CPU load on the child

Receive task from the parent

Send result of task back to the parent

Now that we have a sample child process to run, let's put this all together with a simple application that uses the pooler module and worker modules:

```
var http = require('http');
var makePool = require('./pooler');
var runJob = makePool('./worker');

http.createServer(function (req, res) {
  runJob('some dummy job', function (er, data) {
```

Create job pool around the worker module

Include pooler module to make job pools

Run job on every request to the server, responding with the result

```
    if (er) return res.end('got an error:' + er.message);
    res.end(data);
  });
}).listen(3000);
```

Pooling saves the overhead of spinning up and destroying child processes. It makes use of the communications channels built into `fork` and allows Node to be used effectively for managing jobs across a set of child processes.

GOING FURTHER To further investigate job pools, check out the third-party `compute-cluster` module (https://github.com/lloyd/node-compute-cluster).

We've discussed asynchronous child process execution, which is when you need to juggle multiple points of I/O, like servers. But sometimes you just want to execute commands one after another without the overhead. Let's look at that next.

8.3 Working synchronously

Non-blocking I/O is important for keeping the event loop humming along without having to wait for an unwieldy child process to finish. However, it has extra coding overhead that isn't pleasant when you want things to block. A good example of this is writing shell scripts. Thankfully, synchronous child processes are also available.

TECHNIQUE 63 Synchronous child processes

Synchronous child process methods are recent additions to the Node scene. They were first introduced in Node 0.12 to address a very real problem in a performant and familiar manner: shell scripting. Before Node 0.12, clever but nonperformant hacks were used to get synchronous-like behavior. Now, synchronous methods are a first-class citizen.

In this technique we'll cover all the synchronous methods available in the child process modules.

Problem

You want to execute commands synchronously.

Solution

Use `execFileSync`, `spawnSync`, and `execFile`.

Discussion

By now, we hope these synchronous methods look extremely familiar. In fact, they're the same in their function signatures and purpose as we've discussed previously in this chapter, with one important distinction—*they block and run to completion* when called.

If you just want to execute a single command and get output synchronously, use `execFileSync`:

```
var ex = require('child_process').execFileSync;
var stdout = ex('echo', ['hello']).toString();
console.log(stdout);
```

Extract execFileSync method as ex for a shorthand way to refer to it

execFileSync returns a Buffer of the output, which is then converted to a UTF-8 string and assigned to stdout

Outputs "hello"

If you want to execute multiple commands synchronously and programmatically where the input of one depends on the output of another, use `spawnSync`:

```
var sp = require('child_process').spawnSync;
var ps = sp('ps', ['aux']);
var grep = sp('grep', ['node'], {
  input: ps.stdout,
  encoding: 'utf8'
});
console.log(grep);
```

Run ps aux and grep node synchronously

Extract spawnSync method as sp for a shorthand way to refer to it

Indicate all resulting stdio should be in UTF-8

Pass stdout Buffer from ps aux as input to grep node

The resulting synchronous child process contains a lot of detail of what happened, which is another advantage of using `spawnSync`:

```
{ status: 0,
  signal: null,
  output:
    [ null,
     'wavded 4376 ... 9:03PM 0:00.00 (node)\n
      wavded 4400 ... 9:11PM 0:00.10 node spawnSync.js\n',
     '' ],
  pid: 4403,
  stdout: 'wavded ... 9:03PM 0:00.00 (node)\n
    wavded 4400 ... 9:11PM 0:00.10 node spawnSync.js\n',
  stderr: '',
  envPairs:
    [ 'USER=wavded',
     'EDITOR=vim',
     'NODE_PATH=/Users/wavded/.nvm/v0.11.12/lib/node_modules:',
     ... ],
  options:
    { input: <Buffer 55 53 45 52 20 20 20 ... >,
      encoding: 'utf8',
      file: 'grep',
      args: [ 'grep', 'node' ],
      stdio: [ [Object], [Object], [Object] ] },
  args: [ 'grep', 'node' ],
  file: 'grep' } [
```

Exit status of the process

Signal used to end the process

Output of all the stdio streams used in the child process; we can see that index l (stdout) has data

PID of the process

stdout output for the process

stderr output for the process

Environment variables present when the process ran

Options used to create the process; we can see our input buffer from ps aux here

Arguments used to execute the process

Executable file

Lastly, there's `execSync`, which executes a subshell synchronously and runs the commands given. This can be handy when writing shell scripts in JavaScript:

```
var ex = require('child_process').execSync;
var stdout = ex('ps aux | grep').toString();
console.log(stdout);
```

Extract execSync method as ex for a shorthand way to refer to it

Execute shell command synchronously and return the output as a string

This will output the following:

```
wavded 4425 29.7 0.2 ... 0:00.10 node execSync.js
wavded 4427 1.5 0.0 ... /bin/sh -c ps aux | grep node
wavded 4429 0.5 0.0 ... grep node
wavded 4376 0.0 0.0 ... (node)
```

Output shows we're executing a subshell with execSync

Error handing with synchronous child process methods

If a non-zero exit status is returned in `execSync` or `execFileSync`, an exception will be thrown. The error object will include everything we saw returned using `spawnExec`. We'll have access to important things like the status code and `stderr` stream:

```
var ex = require('child-process').execFileSync;
try {
  ex('cd', ['non-existent-dir'], {
     encoding: 'utf8'
  });
} catch (err) {
  console.error('exit status was', err.status);
  console.error('stderr', err.stderr);
}
```

Executing cd on nonexistent directory gives non-zero exit status

Although more verbose than toString(), setting encoding to UTF-8 here will set it for all our stdio streams when we handle the error

This program yields the following output:

```
exit status was 1
stderr /usr/bin/cd: line 4:cd:
  non-existent-dir: No such file or directory
```

We talked errors in `execFile` and `execFileSync`. What about `spawnSync`? Since `spawnSync` returns everything that happens when running the process, it doesn't throw an exception. Therefore, you're responsible to check the success or failure.

8.4 Summary

In this chapter you learned to integrate different uses of external applications in Node by using the `child_process` module. Here are some tips in summary:

- Use `execFile` in cases where you just need to execute an external application. It's fast, simple, and safer when dealing with user input.
- Use `spawn` when you want to do something more with the I/O of the child process, or when you expect the process to have a large amount of output. It provides a nice streamable interface, and is also safer when dealing with user input.
- Use `exec` when you want to access your shell's facilities (pipes, redirects, blobs). Many shells allow running multiple applications in one go. Be careful with user input though, as it's never a good idea to put untrusted input into an `exec` call.
- Use `fork` when you want to run a Node module as a separate process. This enables computation and file descriptor handling (like an incoming socket) to be handled off the main Node process.

- Detach `spawn`ed processes you want to survive after a Node process dies. This allows Node to be used to set up long-running processes and let them live on their own.
- Pool a cluster of Node processes and use the built-in IPC channel to save the overhead of starting and destroying processes on every `fork`. This is useful for building computational clusters of Node processes.

This concludes our dive into Node fundamentals. We focused on specific core module functionality, focusing on idiomatic Node principals. In the next section, our focus will expand beyond core concepts into real-world development recipes.

Part 2

Real-world recipes

In the first section of this book, we took a deep dive into Node's standard library. Now we'll take a broader look at real-world recipes many Node programs encounter. Node is most famously known for writing fast network-based programs (high-performance HTTP parsing, ease-of-use frameworks like Express), so we devoted a whole chapter to web development.

In addition, there are chapters to help you grasp what a Node program is doing preemptively with tests, and post-mortem with debugging. In closing, we set you up for success when deploying your applications to production environments.

9 The Web: Build leaner and meaner web applications

This chapter covers

- Using Node for client-side development
- Node in the browser
- Server-side techniques and WebSockets
- Migrating Express 3 applications to Express 4
- Testing web applications
- Full-stack frameworks and real-time services

The purpose of this chapter is to bring together the things you've learned about networking, buffers, streams, and testing to write better web applications with Node. There are practical techniques for browser-based JavaScript, server-side code, and testing.

Node can help you to write better web applications, no matter what your background is. If you're a client-side developer, then you'll find it can help you work

more efficiently. You can use it for preprocessing client-side assets and for managing client-side workflows. If you've ever wanted to quickly spin up an HTTP server that builds your CSS or CoffeeScript for a single-page web app, or even just a website, then Node is a great choice.

The previous book in this series, *Node.js in Action*, has a detailed introduction to web development with Connect and Express, and also templating languages like Jade and EJS. In this chapter we'll build on some of these ideas, so if you're completely new to Node, we recommend reading *Node.js in Action* as well. If you're already using Express, then we hope you'll find something new in this chapter; we've included techniques for structuring Express applications to make them easier to scale as your projects grow and mature.

The first section in this chapter has some techniques that focus on the browser. If you're a perplexed front-end developer who's been using Node because your client-side libraries need it, then you should start here. If you're a server-side developer who wants to bring Node to the browser, then skip ahead to technique 66 to see how to use Node modules in the browser.

9.1 Front-end techniques

This section is all about Node and its relationship to client-side technology. You'll see how to use the DOM in Node and Node in the DOM, and run your own local development servers. If you've come to Node from a web design background, then these techniques should help you get into the swing of things before we dive in to deeper server-side examples. But if you're from a server-side background, then you might like to see how Node can help automate front-end chores.

The first technique shows you how to create a quick, static server for simple websites or single-page web applications.

TECHNIQUE 64 Quick servers for static sites

Sometimes you just want to start a web server to work on a static site, or a single-page web application. Node's a good choice for this, because it's easy to get a web server running. It can also neatly encapsulate client-side workflows, making it easier to collaborate with others. Rather than manually running programs over your client-side JavaScript and CSS, you can write Node programs that you can share with other people.

This technique introduces three solutions for starting up a web server: a short Connect script, a command-line web server, and a mini–build system that uses Grunt.

Problem

You want to quickly start a web server so you can develop a static site, or a single-page application.

Solution

Use Connect, a command-line web server, or a client-side workflow tool like Grunt.

Discussion

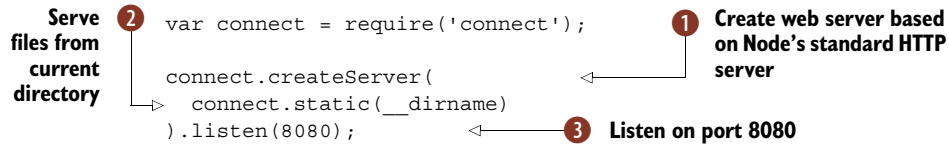
Plain old HTML, JavaScript, CSS, and images can be viewed in a browser without a server. But because most web development tasks end up with files on a server somewhere, you often need a server just to make a static site. It's a chore, but it doesn't need to be! The power of browsers also means you can create sophisticated web applications by contacting external web APIs: single-page web applications, or so-called *serverless apps*.

In the case of serverless web applications, you can work more efficiently by using build tools to preprocess and package client-side assets. This technique will show you how to start a web server for developing static sites, and also how to use tools like Grunt to get a small project going without too much trouble.

Although you could use Node's built in `http` module to serve static sites, it's a lot of work. You'll need to do things like detect the content type of each file to send the right HTTP headers. While the `http` core module is a solid foundation, you can save time by using a third-party module.

First, let's look at how to start a web server with Connect, the HTTP middleware module used to create the popular Express web framework. The first listing demonstrates just how simple this is.

Listing 9.1 A quick static web server



To use the example in listing 9.1, you'll need to install Connect. You can do that by running `npm install connect`, but it's a better idea to create a package.json file so it's easier for other people to see how your project works. Even if your project is a simple static site, creating a package.json file will help your project to grow in the future. All you need to do is memorize these commands: `npm init` and `npm install --save connect`. The first command creates a manifest file for the current directory, and the second will install Connect and save it to the list of dependencies in the new package.json file. Learn these and you'll be creating new Node projects in no time.

The `createServer` method (1) is derived from Node's `http.createServer`, but it's wrapped with a few things that Connect adds behind the scenes. The `static` server middleware component (2) is used to serve files from the current directory (`__dirname` with two underscores means "current directory"), but you can change the directory if you like. For example, if you have client-side assets in public/, then you can use `connect.static(__dirname + '/public')` instead.

Finally, the server is set to listen on port 8080 (3). That means if you run this script and visit http://localhost:8080/file.html in a browser, you should see file.html.

If you've been sent a bunch of HTML files from a designer, and you want to use a server to view them because they make use of paths to images and CSS files with a leading forward slash (/), then you can also use a command-line web server. There are many of these available on npm, and they all support different options. One example is `glance` by Jesse Keane. You can find it on GitHub at https://github.com/jarofghosts/glance, and on npm as `glance`.

To use `glance` on the command line, navigate to a directory where you have some HTML files that you want to look at. Then install `glance` systemwide with `npm install --global glance`, and type `glance`. Now go to http://localhost:61403/file, where *file* is a file you want to look at, and you should see it in your browser.

`glance` can be configured in various ways—you can change the port from 61403 to something else with `--port`, and specify the directory to be served with `--dir`. Type `--help` to get a list of options. It also has some nice defaults for things like 404s—figure 9.1 shows what a 404 looks like.

The third way of running a web server is to use a task runner like Grunt. This allows you to automate your client-side tasks in a way that others can replicate. Using Grunt is a bit like a combination of the previous two approaches: it requires a web server module like Connect, and a command-line tool.

To use Grunt for a client-side project you'll need to do three things:

1. Install the `grunt-cli` module.
2. Make a package.json to manage the dependencies for your project.
3. Use a Grunt plugin that runs a web server.

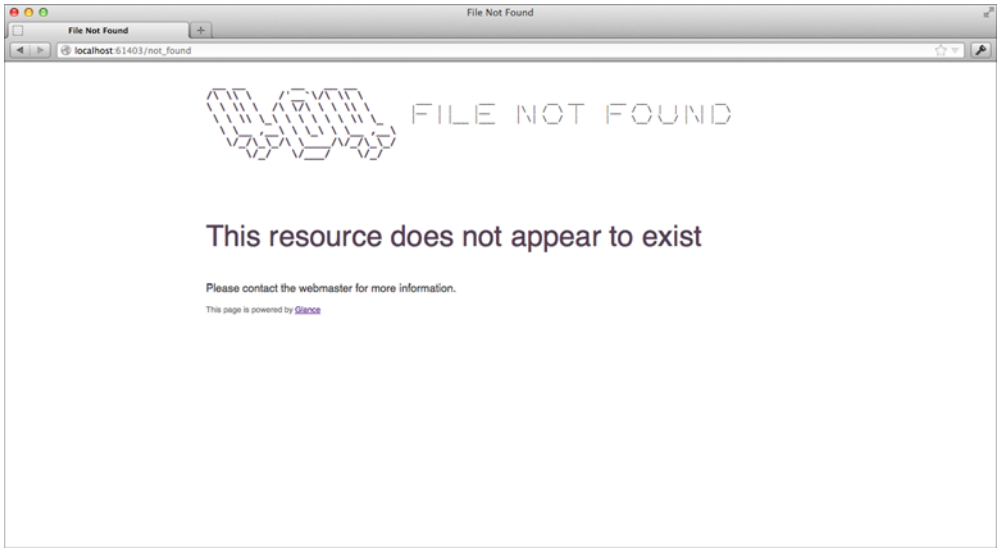


Figure 9.1 Glance has built-in pages for errors.

The first step is easy: install `grunt-cli` as a global module with `npm install -g grunt-cli`. Now you can run Grunt tasks from any project that includes them by typing `grunt`.

Next, make a new directory for your project. Change to this new directory and type `npm init`—you can press Return to accept each of the defaults. Now you need to install a web server module: `npm install --save-dev grunt grunt-contrib-connect` will do the job.

The previous command also installed `grunt` as a development dependency. The reason for this is it locks Grunt to the current version—if you look at package.json you'll see something like `"grunt": "~0.4.2"`, which means Grunt was installed first at `0.4.2`, but newer versions on the `0.4` branch will be used in the future. The popularity of modules like Grunt forced npm to support something known as *peer dependencies*. Peer dependencies allow Grunt plugins to express a dependency on a specific version of Grunt, so the Connect module we'll use actually has a `peerDependencies` property in its package.json file. The benefit of this is you can be sure plugins will work as Grunt changes—otherwise, as Grunt's API changes, plugins might just break with no obvious cause.

> **ALTERNATIVES TO GRUNT** At the time of writing, Grunt was the most popular build system for Node. But new alternatives have appeared and are rapidly gaining adoption. One example is Gulp (http://gulpjs.com/), which takes advantage of Node's streaming APIs and has a light syntax that is easy to learn.

In case all this is new to you, we've included a screenshot of what your project should look like in figure 9.2.

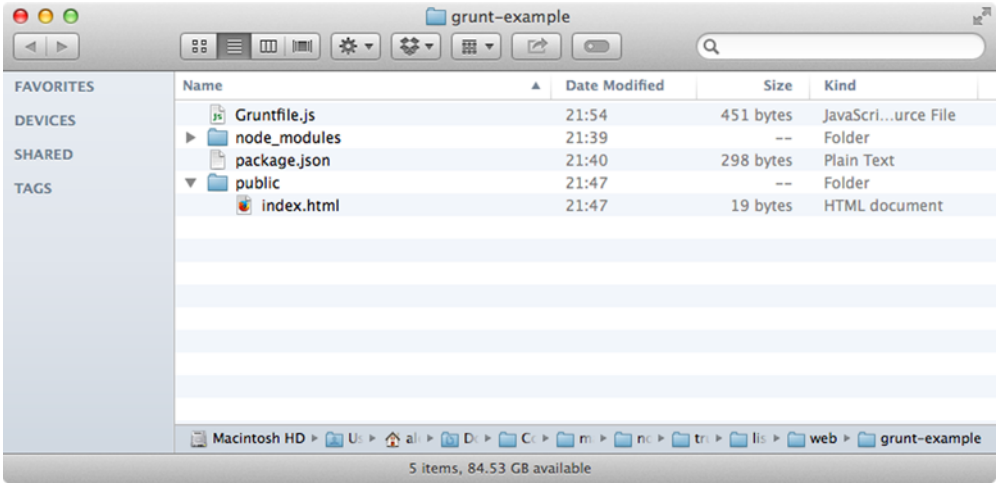


Figure 9.2 Projects that use Grunt typically have a package.json and a Gruntfile.js.

Now that we have a fresh project set up, the final thing to do is create a file called Gruntfile.js. This file contains a list of tasks that `grunt` will run for you. The next listing shows an example that uses the `grunt-contrib-connect` module.

Listing 9.2 A Gruntfile for serving static files

```
module.exports = function(grunt) {
  grunt.loadNpmTasks('grunt-contrib-connect');

  grunt.initConfig({
    connect: {
      server: {
        options: {
          port: 8080,
          base: 'public',
          keepalive: true
        }
      }
    }
  });

  grunt.registerTask('default', ['connect:server']);
};
```

All Gruntfiles export a function (1)

(2) This loads Connect plugin

(3) The base path for static files

(4) Default command is aliased here

You should also create a directory called `public` with an index.html file—the HTML file can contain anything you like. After that, type `grunt connect` from the same directory as your Gruntfile.js, and the server should start. You can also type `grunt` by itself, because we set the default task to `connect:server` (4).

Gruntfiles use Node's standard module system, and receive an object called `grunt` (1) that can be used to define tasks. Plugins are loaded with `grunt.loadNpmTasks`, allowing you to reference modules installed with npm (2). Most plugins have different options, and these are set by passing objects to `grunt.initConfig`—we've defined a server port and base path, which you can change by modifying the `base` property (3).

Using Grunt to start a web server is more work than writing a tiny Connect script or running `glance`, but if you take a look at Grunt's plugin list (http://gruntjs.com/plugins), you'll see over 2,000 entries that cover everything from building optimized CSS files to Amazon S3 integration. If you've ever needed to concatenate client-side JavaScript or generate image sprites, then chances are there's a plugin that will help you automate it.

In the next technique you'll learn how to reuse client-side code in Node. We'll also show you how to render web content inside Node processes.

TECHNIQUE 65 Using the DOM in Node

With a bit of work, it's possible to simulate a browser in Node. This is useful if you want to make web scrapers—programs that convert web pages into structured content. This is technically rather more complicated than it may seem. Browsers don't just provide JavaScript runtimes; they also have Document Object Model (DOM) APIs that don't exist in Node.

Such a rich collection of libraries has evolved around the DOM that it's sometimes hard to imagine solving problems without them. If only there were a way to run libraries like jQuery inside Node! In this technique you'll learn how to do this by using browser JavaScript in a Node program.

Problem

You want to reuse client-side code that depends on the DOM in Node, or render entire web pages.

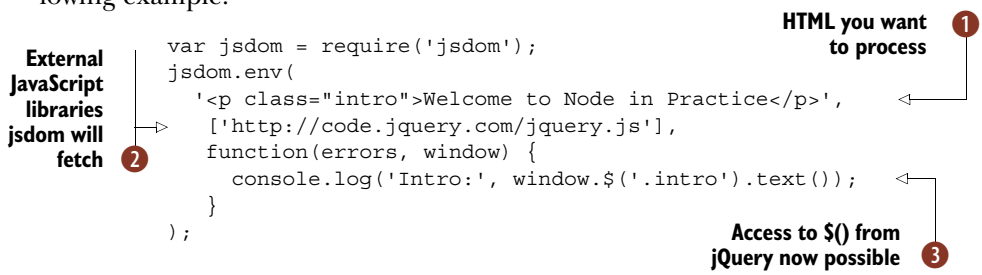
Solution

Use a third-party module that provides a DOM layer.

Discussion

The W3C DOM is a well-defined standard. When designers struggle with browser incompatibilities, they're often dealing with the fact that standards require a degree of interpretation, and browser manufacturers have naturally interpreted the standards slightly differently. If your goal is just to run JavaScript that depends on the JavaScript DOM APIs, then you're in luck: these standards can be re-created well enough that you can run popular client-side libraries in Node.

One early solution to this problem was `jsdom` (https://github.com/tmpvar/jsdom). This module accepts an *environment* specification and then provides a `window` object. If you install it with `npm install -g jsdom`, you should be able to run the following example:



This example takes in HTML 1, fetches some remote scripts 2, and then gives you a `window` object that looks a lot like a browser `window` object 3. It's good enough that you can use jQuery to manipulate the HTML snippet—jQuery works as if it's running in a browser. This is useful because now you can write scripts that process HTML documents in the way you're probably used to: rather than using a parser, you can query and manipulate HTML using the tools you're familiar with. This is amazing for writing succinct code for tasks like web scraping, which would otherwise be frustrating and tedious.

Others have iterated on jsdom's approach, simplifying the underlying dependencies. If you really just want to process HTML in a jQuery-like way, then you could use `cheerio` (https://npmjs.org/package/cheerio). This module is more suited to web scraping, so if you're writing something that downloads, processes, and indexes HTML, then `cheerio` is a good choice.

In the following example, you'll see how to use `cheerio` to process HTML from a real web page. The actual HTML is from manning.com/index.html, but as designs change frequently, we've kept a copy of the page we used in our code samples. You can find it in cheerio-manning/index.html. The following listing opens the HTML file and queries it using a CSS selector, courtesy of `cheerio`.

Listing 9.3 Scraping a web page with `cheerio`

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

fs.readFile('./index.html', 'utf8', function(err, html) {
  var $ = cheerio.load(html);
  var releases = $('.Releases a strong');

  releases.each(function(i) {
    console.log('New release:', this.text());
  });
});
```

Load HTML content (1)

Query it using CSS selector (2)

Extract the text (3)

The HTML is loaded with `fs.readFile`. If you were doing this for real then you'd probably want to download the page using HTTP—feel free to replace `fs.readFile` with `http.get` to fetch Manning's index page over the network. We have a detailed example of `http.get` in chapter 7, technique 51, "Following redirects."

Once the HTML has been fetched, it's passed to `cheerio.load` (1). Setting the result as a variable called `$` is just a convention that will make your code easier to read if you're used to jQuery, but you could name it something else.

Now that everything is set up, you can query the HTML; `$('.Releases a strong')` is used (2) to query the document for the latest books that have been released. They're in a `div` with a class of `Releases`, as anchor tags.

Each element is iterated over using `releases.each`, just like in jQuery. The callback's context is changed to be the current element, so `this.text()` is called to get the text contained by the node (3).

Because Node has such a wide collection of third-party modules, you could take this example and make all sorts of amazing things with it. Add Redis for caching and queueing websites to process, then scrape the results and throw it at Elasticsearch, and you've got your own search engine!

Now you've seen how to run JavaScript intended for browsers in Node, but what about the opposite? You might have some Node code that you want to reuse client-side, or you might want to just use Node's module system to organize your client-side code. Much like we can simulate the DOM in Node, we can do the same in the browser. In the next technique you'll learn how to do this by running your Node scripts in browsers.

TECHNIQUE 66 Using Node modules in the browser

One of Node's selling points for JavaScript is that you can reuse your existing browser programming skills on servers. But what about *reusing* Node code in browsers without any changes? Wouldn't that be cool? Here's an example: you've defined data models in Node that do things like data validation, and you want to reuse them in the browser to automatically display error messages when data is invalid.

This is almost possible, but not quite: unfortunately browsers have quirks that must be ironed out. Also, important features like `require` don't exist in client-side JavaScript. In this technique you'll see how you can take code intended for Node, and convert it to work with most web browsers.

Problem

You want to use `require()` to structure your client-side code, or reuse entire Node modules in the browser.

Solution

Use a program like Browserify that is capable of converting Node JavaScript into browser-friendly code.

Discussion

In this technique we'll use Browserify (http://browserify.org/) to convert Node modules into browser-friendly code. Other solutions also exist, but at this point Browserify is one of the more mature and popular solutions. It doesn't just patch in support for `require()`, though: it can convert code that relies on Node's stream and network APIs. You can even use it to recursively convert modules from npm.

To see how it works, we'll first look at a short self-contained example. To get started, install Browserify with npm: `npm install -g browserify`. Once you've got Browserify installed, you can convert your Node modules into Browser scripts with `browserify index.js -o bundle.js`. Any `require` statements will cause the files to be included in bundle.js, so you shouldn't change this file. Instead, overwrite it whenever your original files have changed.

Listing 9.4 shows a sample Node program that uses `EventEmitter` and `utils.inherit` to make the basis of a small messaging class.

Listing 9.4 Node modules in the browser

```
var EventEmitter = require('events').EventEmitter;
var util = require('util');

function MessageBus(options) {
  EventEmitter.call(this, options);
  this.on('message', this.messageReceived.bind(this));
}

util.inherits(MessageBus, EventEmitter);

MessageBus.prototype.messageReceived = function(message) {
  console.log('RX:', message);
```

1 Load modules with require() as you usually would.

2 EventEmitter can be inherited using util.inherits.

```
};

var messageBus = new MessageBus();
messageBus.emit('message', 'Hello world!');
```

Running Browserify on this script generates a bundle that's about 1,000 lines long! But we can use `require` as we would in any Node program (1), and the Node modules we know and love will work, as you can see in listing 9.4 by the use of `util.inherits` and `EventEmitter` (2).

With Browserify, you can also use `require` and `module.exports`, which is better than having to juggle `<script>` tags. The previous example can be extended to do just that. In listing 9.5, Browserify is used to make a client-side script that can load `MessageBus` and jQuery with `require`, and then modify the DOM when messages are emitted.

Listing 9.5 Node modules in the browser

```
var MessageBus = require('./messagebus');
var messageBus = new MessageBus();
var $ = require('jquery')(window);

messageBus.on('message', function(msg) {
  $('#messages').append('<p>' + msg + '</p>');
});

$(function() {
  messageBus.emit('message', 'Hello from example 2');
});
```

(1) jQuery can be loaded with Browserify!

(2) jQuery's DOM ready function can be used.

By creating a package.json file with `jquery` as a dependency, you can load jQuery using Browserify (1). Here we've used it to attach a `DOMContentLoaded` listener (2) and append paragraphs to a container element when messages are received.

Source maps

If the JavaScript files you generate with Browserify raise errors, then it can be hard to untangle the line numbers in stack traces, because they refer to line numbers in the monolithic bundle. If you include the `--debug` flag when building the bundle, then Browserify will generate mappings that point to the original files and line numbers.

These mappings require a compatible debugger—you'll also need to tell your browser's debugging tools to use them. In Chrome you'll need to select *Enable source maps*, under the options in Chrome's DevTools.

To make this work, all you need to do is add `module.exports = MessageBus` to the example from listing 9.4, and then generate the bundle with `browserify index.js -o bundle.js`, where index.js is listing 9.5. Browserify will dutifully follow the `require` statements from index.js to pull in jQuery from `./node_modules` and the `MessageBus` class from messagebus.js.

Because people might forget how to build the script, you can add a `scripts` entry to your package.json file, like this: `"build": "browserify index.js -o bundle.js"`. The downloadable code samples for this book include both a sample package.json file and a suitable HTML file for running the entire example in a browser.

There's another way to build bundles with Browserify: by using Browserify as a module in a Node program. To use it, you need to create a `Browserify` instance (1) and then tell it what files you want to build (2):

```
var browserify = require('browserify');
var b = browserify();
b.add('./index.js');
b.bundle().pipe(process.stdout);
```

(1) Create Browserify instance

(2) Specify files you want to bundle

You could use this as part of a more complex build process, or put in a Grunt task to automate your build process. Now that you've seen how to use Node modules in the browser and how to simulate the browser in Node, it's time to learn how to improve your server-side web applications.

9.2 Server-side techniques

This section includes general techniques for building web applications. If you're already using Express, then you'll be able to use these techniques to improve how your Express programs are organized. Express aims to be simple, which makes it flexible, but sometimes it's not easy to see how to use it in the best way. The patterns and solutions we've created have come from using Express to build commercial and open source web applications over the last few years. We hope they'll help you to write better web applications.

> **EXPRESS 3 AND 4** The techniques in this section refer to Express 3. Most will work with Express 4, or may require minor modifications. For more about migrating to Express 4, see technique 75.

TECHNIQUE 67 Express route separation

The documentation and popular tutorials for Express usually organize all the code in a single file. In real projects, this eventually becomes unmanageable. This technique uses Node's module system to separate related routes into files, and also includes ways to get around the Express `app` object being in a different file.

Problem

Your main Express application file has become extremely large, and you want a better way to organize all of those routes.

Solution

Use route separation to split related routes into modules.

Discussion

Express is a minimalist framework, so it doesn't hold your hand when it comes to organizing projects. Projects that start simple can become unwieldy if you don't pay

attention. The secret to successfully organizing larger projects is to embrace Node's module system.

The first avenue of attack is routes, but you can apply this technique to every facet of development with Express. You can even treat applications as self-contained Node modules, and mount them within other applications.

Here's a typical example of some Express routes:

```
app.get('/notes', function(req, res, next) {
  db.notes.findAll(function(err, notes) {
    if (err) return next(err);
    res.send(notes);
  });
});

app.post('/notes', function(req, res, next) {
  db.notes.create(req.body.note, function(err, note) {
    if (err) return next(err);
    res.send(note);
  });
});
```

This route displays a list of notes.

This route is used to create notes.

The full example project can be found in listings/web/route_separation/app_monolithic.js. It contains a set of CRUD routes for creating, finding, and updating notes. An application like this would have other CRUD routes as well: perhaps notes can be organized into notebooks, and there will definitely be some user account management, and extra features like setting reminders. Once you have about four or five of these sets of routes, the application file could be hundreds of lines of code.

If you wrote this project as a single, large file, then it would be prone to many problems. It would be easy to make mistakes where variables are accidentally global instead of local, so dangerous side effects can be encountered under certain conditions. Node has a built-in solution which can be applied to Express and other web frameworks: directories as modules.

To refactor your routes using modules, first create a directory called `routes`, or `controllers` if you prefer. Then create a file called index.js. In our case it'll be a simple three-line file that exports the notes routes:

```
module.exports = {
  notes: require('./notes')
};
```

1 Export each routing module like this

Here we have just one routing module, which can be loaded with `require` and a relative path 1. Next, copy and paste the entire set of routes into routes/notes.js. Then delete the route definition part—for example, `app.get('/notes',`, and replace it with an export: `module.exports.index = function(req, res) {`.

The refactored files should look like the next listing.

Listing 9.6 A routing module without the rest of the application

```
var db = require('./../db');

module.exports.index = function(req, res, next) {
  db.notes.findAll(function(err, notes) {
    if (err) return next(err);
    res.send(notes);
  });
};

module.exports.create = function(req, res, next) {
  db.notes.create(req.body.note, function(err, note) {
    if (err) return next(err);
    res.send(note);
  });
};

module.exports.update = function(req, res, next) {
  db.notes.update(req.param('id'), req.body.note, function(err, note) {
    if (err) return next(err);
    res.send(note);
  });
};

module.exports.show = function(req, res, next) {
  db.notes.find(req.param('id'), function(err, note) {
    if (err) return next(err);
    res.send(note);
  });
};
```

1 Export each routing function with a name that reflects its CRUD operation.

Each routing function is exported with a CRUD-inspired name (index, create, update, show) 1. The corresponding app.js file can now be cleared up. The next listing shows just how clean this can look.

Listing 9.7 A refactored app.js file

```
var express = require('express');
var app = express();
var routes = require('./routes');

app.use(express.bodyParser());

app.get('/notes', routes.notes.index);
app.post('/notes', routes.notes.create);
app.patch('/notes/:id', routes.notes.update);
app.get('/notes/:id', routes.notes.show);

module.exports = app;
```

1 Load all routes at once

2 Bind them to HTTP verb and partial URL

3 Export app object

All of the routes can be loaded at once with `require('./routes')` 1. This is convenient and clean, because there are fewer `require` statements that would otherwise

clutter app.js. All you need to do is remove the old route callbacks and add in references to each routing function ❷.

Don't put an `app.listen` call in this file; export `app` instead ❸. This makes it easier to test the application. Another advantage of exporting the `app` object is that you can easily load the `app.js` module from anywhere within the application. Express allows you to get and set configuration values, so making `app` accessible can be useful if you want to refer to these settings in places outside the routes. Also note that `res.app` is available from within routes, so you don't need to pass the `app` object around too often.

If you want to easily load app.js without creating a server, then name the application file app.js, and have a separate server.js file that calls `app.listen`. You can set up the `server` property in package.json to use `node server.js`, which allows people to start the application with `npm start`—you can also leave out the `server` property, because `node server.js` is the default, but it's better to define it so people know how you intend them to use it.

> **Directories as modules**
>
> This technique puts all of the routes in a directory, and then exports them with an index.js file so they can be loaded in one go with `require('./routes')`.
>
> This pattern can be reused in other places. It's great for organizing middleware, database modules, and configuration files.
>
> For an example of using directories as modules to organize configuration files, see technique 69.

The full example for this technique can be found in listings/web/route-separation, and it includes sample tests in case you want to unit test your own projects.

Properly organizing your Express projects is important, but there are also workflow issues that can slow down development. For example, when you're working on a web application, you'll typically make many small changes and then refresh the browser to see the results. Most Node frameworks require the process to be restarted before seeing the changes take effect, so the next technique explores how this works and how to efficiently solve this problem.

TECHNIQUE 68 Automatically restarting the server

Although Node comes with tools for monitoring changes to files, it can be a lot of work to use them productively. This technique looks at `fs.watch`, and introduces a popular third-party tool for automatically restarting web applications as files are edited.

Problem

You need to restart your Node web application every time you edit files.

Solution

Use a file watcher to restart the application automatically.

Discussion

If you're used to languages like PHP or ASP, Node's in-process server-based model might seem unusual. One of the big differences about Node's model is that you need to restart the process when files change. If you think about how `require` and V8 work, then this makes sense—files are generally loaded and interpreted once.

One way to get around this is to detect when files change, and then restart the application. Node makes good use of non-blocking I/O, and one of the properties of non-blocking file system APIs is that listeners can be used to wait for specific events. To solve this problem, you could set up file system event handlers for all of the files in your project. Then, when files change, your event handler can restart the project.

Node provides an API for this in the `fs` module called `fs.watch`. At the time of writing, this API is unstable—that means it may be changed in subsequent versions of Node. This method has been covered in chapter 6, section 6.1.4. Let's look at how it could be used with a web application. Figure 9.8 shows a program that can watch and reload a simple web server.

Listing 9.8 Reloading a Node process

```
var fs = require('fs');
var exec = require('child_process').exec;

function watch() {
  var child = exec('node server.js');
  var watcher = fs.watch(__dirname + '/server.js', function(event) {
    console.log('File changed, reloading.');
    child.kill();
    watcher.close();
    watch();
  });
}

watch();
```


Watching a file for changes with `fs.watch` is slightly convoluted, but you can use `fs.watchFile`, which is based on file polling instead of I/O events. The way listing 9.8 works is to start a child process—in this case `node server.js` 1—and then watch that file for changes 2. Starting and stopping processes is managed with the `child_process` core module, and the `kill` method is used to stop the child process 3.

On Mac OS we found it's best to also stop watching the file with `watcher.close` 4, although Node's documentation indicates that `fs.watch` should be "persistent." Once all of that is done, the `watch` function is called recursively to launch the web server again 5.

This example could be run with a server.js file like this:

```
var http = require('http');
var server = http.createServer(function(req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
```

```
  res.end('This is a super basic web application');
});

server.listen(8080);
```

This works, but it's not exactly elegant. And it's not complete, either. Most Node web applications consist of multiple files, so the file-watching logic will become more complicated. It's not enough to recurse over the parent directories, because there are lots of files that you don't want to watch—you don't want to watch the files in `.git`, and if you're writing an Express application you probably don't want to watch view templates, because they're loaded on demand without caching in development mode.

Suddenly automatically restarting Node programs seems less trivial, and that's where third-party modules can help. One of the most widely used modules that solves this problem is Remy Sharp's `nodemon` (http://nodemon.io/). It works well for watching Express applications out of the box, and you can even use it to automatically restart any kind of program, whether it's written in Node or Python, Ruby, and so on.

To try it out, type `npm install -g nodemon`, and then navigate to a directory that contains a Node web application. If you want to use a small sample script, you can use our example from listings/web/watch/server.js.

Start running and watching server.js by typing `nodemon server.js`, and you'll find you can edit the text in `res.end` and the change will be reflected the next time you load http://localhost:8080/.

You might notice a small delay before changes are visible—that's just Nodemon setting up `fs.watch`, or `fs.watchFile` if it's not available on your OS. You can force it to reload by typing `rs` and pressing Return.

Nodemon has some other features that will help you work on web applications. Typing `nodemon --help` will show a list of command-line options, but you can get greater, VCS-friendly control by creating a nodemon.json file. This allows you to specify an array of files to ignore, and you can also map file extensions to program names by using the `execMap` setting. Nodemon's documentation includes a sample file that illustrates each of the features.

The next listing is an example Nodemon configuration that you can adapt for your own projects.

Listing 9.9 Nodemon's configuration file

```
{
  "ignore": [
    ".git",
    "node_modulesnode_modules"
  ],
  "execMap": {
    "js": "node --harmony"
  },
  "watch": [
    "test/fixtures/",
```

1 A list of paths to ignore

2 Automatically map .js files to use node with harmony flag

3 Specify paths to watch

```
    "test/samples/"
  ],
  "env": {                          <-- List environmental variables (4)
    "NODE_ENV": "development"
  },
  "ext": "js json"
}
```

The basic options allow you to ignore specific paths (1), and list multiple paths to watch (3). This example uses `execMap` to automatically run `node` with the `--harmony` flag[1] for all JavaScript files (2). Nodemon can also set environmental variables—just add some values to the `env` property (4).

Once your workflow is streamlined thanks to Nodemon, the next thing to do is to improve how your project is configured. Most projects need some level of configuration—examples include the database connection details and authorization credentials for remote APIs. The next technique looks at ways to configure your web application so you can easily deploy it to multiple environments, run it in test mode, and even tweak how it behaves during local development.

TECHNIQUE 69 Configuring web applications

This technique looks at the common patterns for configuring Node web applications. We'll include examples for Express, but you can use these patterns with other web frameworks as well.

Problem

You have configuration options that change between development, testing, and production.

Solution

Use JSON configuration files, environmental variables, or a module for managing settings.

Discussion

Most web applications require some configuration values to operate correctly: database connection strings, cache settings, and email server credentials are typical. There are many ways to store application settings, but before you install a third-party module to do it, consider your requirements:

- Is it acceptable to leave database credentials in your version control repository?
- Do you really need configuration files, or can you embed settings into the application?
- How can configuration values be accessed in different parts of the application?
- Does your deployment environment offer a way to store configuration values?

The first point depends on your project or organization's policies. If you're building an open source web application, you don't want to leave database accounts in the public repository, so configuration files might not be the best solution. You want people to

[1] `--harmony` is used to enable all of the newer ECMAScript features available to Node.

install your application quickly and easily, but you don't want to accidentally leak your passwords. Similarly, if you work in a large organization with database administrators, they might not be comfortable about letting everyone have direct access to databases.

In such cases, you can set configuration values as part of the deployment environment. Environmental variables are a standard way to configure the behavior of Unix and Windows programs, and you can access them with `process.env`. The basic example of this is switching between deployment environments, using the `NODE_ENV` setting. The following listing shows the pattern Express uses for storing configuration values.

Listing 9.10 Configuring an Express application

```
var express = require('express');
var app = express();

app.set('port', process.env.PORT || 3000);

app.configure('development', function() {
  app.set('db', 'localhost/development');
});

app.configure('production', function() {
  app.set('db', 'db.example.com/production');
});

app.listen(app.get('port'), function() {
  console.log('Using database:', app.get('db'));
  console.log('Listening on port:', app.get('port'));
});
```

(1) Use 3000 if the environmental variable PORT is not set.

(2) This callback will only run if NODE_ENV is set to development.

(3) This callback will only run if NODE_ENV is set to production.

(4) Use app.get() to access settings.

Express has a small API for setting application configuration values: `app.set`, `app.get` (4), and `app.configure`. You can also use `app.enable` and `app.disable` to toggle Boolean values, and `app.enabled` and `app.disabled` to query them. The `app.configure` blocks are equivalent to `if (process.env.NODE_ENV === 'development')` (2) and `if (process.env.NODE_ENV === 'production')` (3), so you don't really need to use `app.configure` if you don't want to. It will be removed in Express 4. If you're not using Express, you can just query `process.env`.

The `NODE_ENV` environmental variable is controlled by the shell. If you want to run listing 9.10 in production mode, you can type `NODE_ENV=production node config.js`, and you should see it print the production database string. You could also type `export NODE_ENV=production`, which will cause the application to always run in production mode while the current shell is running.

The reason we've used `PORT` (1) to set the port is because that's the default name Heroku uses. This allows Heroku's internal HTTP routers to override the port your application listens on.

You could use `process.env` throughout your code instead of `app.get`, but using the `app` object feels cleaner. You don't need to pass `app` around—if you've used the route separation pattern from technique 67, then you'll be able to access it through `res.app`.

If you'd rather use configuration files, the easiest and quickest way is to use the folder as a module technique with JSON files. Create a folder called config/, and then create an index.js file, and a JSON file for each environment. The next listing shows what the index.js file should look like.

Listing 9.11 A JSON configuration file loader

```
var config = {
  development: require('./development.json'),
  production: require('./production.json'),
  test: require('./test.json')
};

module.exports = config[process.env.NODE_ENV || 'development'];
```

1 Load a JSON file with require()

2 Check NODE_ENV to see which file to use

Node's module system allows you to load a JSON file with `require` 1, so you can load each environment's configuration file and then export the relevant one using `NODE_ENV` 2. Then whenever you need to access settings, just use `var config = require('./config')`—you'll get a plain old JavaScript object that contains the settings for the current environment. The next listing shows an example Express application that uses this technique.

Listing 9.12 Loading the configuration directory

```
var express = require('express');
var app = express();
var config = require('./config');

app.listen(config.port, function() {
  console.log('Using database:', config.db);
  console.log('Listening on port:', config.port);
});
```

Load settings using require()

This is so easy it almost feels like cheating! All you have to do is call `require('./config')` and you've got your settings. Node's module system should cache the file as well, so once you've called `require` it shouldn't need to evaluate the JSON files again. You can repeatedly call `require('./config')` throughout your application.

This technique takes advantage of JavaScript's lightweight syntax for setting and accessing values on objects, as well as Node's module system. It works well for lots of types of projects.

There's one more approach to configuration: using a third-party module. After the last technique, you might think this is overkill, but third-party modules can offer a lot of functionality, including command-line option parsing. It might be that you often need to switch between different options, so overriding application settings with command-line options is attractive.

The web framework Flatiron (http://flatironjs.org/) has an application configuration module called `nconf` (https://npmjs.org/package/nconf) that handles configuration files, environmental variables, and command-line options. Each can be given precedence, so you can make command-line options override configuration files. It's a unifying framework for processing options.

The following listing shows how `nconf` can be used to configure an Express application.

Listing 9.13 Using `nconf` to configure an Express application

```
var express = require('express');
var app = express();
var nconf = require('nconf');
var routes = require('./routes');

nconf
  .argv()
  .env()
  .file({ file: 'config.json' });

nconf.set('db', 'localhost/development');
nconf.set('port', 3000);

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

app.listen(nconf.get('port'), function() {
  console.log('Using database:', nconf.get('db'));
  console.log('Listening on port:', nconf.get('port'));
});
```

1 **Tell nconf to optionally use configuration file, and override it with command-line arguments**

2 **Set a default for db setting**

3 **Get the port**

Here we've told `nconf` to prioritize options from the command line, but to also read a configuration file if one is available 1. You don't need to create a configuration file, and `nconf` can create one for you if you use `nconf.save`. That means you could allow users of your application to change settings and persist them. This works best when `nconf` is set up to use a database to save settings—it comes with built-in Redis support.

Default values can be set with `nconf.set` 2. If you run this example without any options, it should use port 3000, but if you start it with `node app.js --port 3001`, it'll use whatever you pass with `--port`. Getting settings is as simple as `nconf.get` 3.

And you don't need to pass the `nconf` object around! Settings are stored in memory. Other files in your project can access settings by loading `nconf` with `require`, and then calling `nconf.get`. The next listing loads `nconf` again, and then tries to access the `db` setting.

Listing 9.14 Loading `nconf` elsewhere in the application

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

module.exports.index = function(req, res) {
  res.send('Using database:', nconf.get('db'));
};
```

1 **If you load nconf again, it'll know what to do.**

Even though it seems like `var nconf = require('nconf')` might return a pristine copy of `nconf`, it doesn't ❶.

A well-organized and carefully configured web application can still go wrong. When your application crashes, you'll want logs to help debug the problem. The next technique will help you improve how your application handles errors.

TECHNIQUE 70 Elegant error handling

This technique looks at using the `Error` constructor to catch and handle errors in your application.

Problem

You want to centralize error handling to simplify your web applications.

Solution

Inherit from `Error` with error classes that include HTTP status codes, and use a middleware component to handle errors based on content type.

Discussion

JavaScript has an `Error` constructor that you can inherit from to represent specific types of errors. In web development, some errors frequently crop up: incorrect URLs, incorrect parameters for query parameters or form values, and authentication failures. That means you can define errors that include HTTP codes alongside the typical things `Error` provides.

Rather than branching on error conditions in HTTP routers, you should call `next(err)`. The next listing shows how that works.

Listing 9.15 Passing errors to middleware

```
var db = require('./../db');
var errors = require('./../errors');

module.exports.show = function(req, res, next) {
  db.notes.find(req.param('id'), function(err, note) {
    if (err) return next(err);
    if (!note) {
      return next(new errors.NotFound('That note was not found.'));
    }
    res.send(note);
  });
};
```

Make sure the route handler signature includes the third parameter, next. ❷

Keep error objects organized in separate file. ❶

If an error was passed by the database API, return early. ❸

If a note couldn't be found, create an instance of a suitable error class. ❹

In this example, error classes have been defined in a separate file ❶, which you can find in listing 9.16. The route handler includes a third argument, `next` ❷, after the standard `req, res` arguments that we've used in previous techniques.

Many of your route handlers will load data from a database, whether it's MySQL, PostgreSQL, MongoDB, or Redis, so this example is based around a generic asynchronous database API. If an error was encountered by the database API, then return early and call `next`, including the error object as the first argument. This will pass the error along to the next middleware component ❸. This route handler has an additional

piece of logic—if a note wasn't found in the database, then an error object is instantiated and passed along using `next` ❹.

The following listing shows how to inherit from `Error`.

Listing 9.16 Inheriting errors and including status codes

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

function HTTPError() {
  Error.call(this, arguments);
}

util.inherits(HTTPError, Error);

function NotFound(message) {
  HTTPError.call(this);
  Error.captureStackTrace(this, arguments.callee);
  this.statusCode = 404;
  this.message = message;
  this.name = 'NotFound';
}
util.inherits(NotFound, HTTPError);

module.exports = {
  HTTPError: HTTPError,
  NotFound: NotFound
};
```

❶ Create generic HTTPError class

❷ Inherit from Error, using util.inherits

❸ Optionally capture stack trace

❹ Set status code that can be passed to browser

❺ Additional HTTP errors can inherit from HTTPError

Here we've opted to create two classes. Instead of just defining `NotFound`, we've created `HTTPError` ❶ and inherited from it ❺. This is so it's easier to track if an error is related to HTTP, or if it's something else. The base `HTTPError` class inherits from `Error` ❷.

In the `NotFound` error, we've captured the stack trace to aid with debugging ❸, and set a `statusCode` property ❹ that can be reported to the browser.

The next listing shows how to create an error-handling middleware component in a typical Express application.

Listing 9.17 Using an error-handling middleware component

```
var errors = require('./errors');
var express = require('express');
var app = express();
var routes = require('./routes');

app.use(express.bodyParser());

app.get('/notes/:id', routes.notes.show);

app.use(function(err, req, res, next) {
  if (process.env.NODE_ENV !== 'test') {
    console.error(err.stack);
  }
```

❶ If four arguments are used with app.use, then the first argument is the error object.

❷ Print stack traces if you're not running in the test mode.

```
  res.status(err.statusCode || 500);

  res.format({
    text: function() {
      res.send(err.message);
    },

    json: function() {
      res.send(err);
    },

    html: function() {
      res.render('errors', { err: err });
    }
  });
});

module.exports = app;
```

3 Respond with errors in the expected format.

This middleware component is fairly simple, but it has some tweaks that we've found work well in production. To get the error objects passed by `next`, make sure to use the four-parameter form of `app.use`'s callback 1. Also note that this middleware component comes at the end of the chain, so you need to put it after all your other middleware and route definitions.

You can conditionally print stack traces so they're not visible when specifically testing expected errors 2—errors may be triggered as part of testing, and you wouldn't want stack traces cluttering the test output.

Because this centralizes error handling into the main application file, it's a good idea to conditionally return different formats. This is useful if your application provides a JSON API as well as HTML pages. You can use `app.format` to do this 3, and it works by checking the MIME type in the request's `Accept` header. The JSON response might not be needed, but it's possible that your API would return well-formed errors that can be consumed by clients—it can be difficult to deal with APIs that suddenly respond with HTML when you're asking for JSON.

Somewhere in your tests you should check that these errors do what you want. The following snippet shows a Mocha test that makes sure 404s are returned when expected, and in the expected format:

```
describe('Error handling', function() {
  it('should return a 404 for IDs that do not exist', function(done) {
    request(app)
      .get('/notes/999')
      .expect(404, done);
   });

  it('should send JSON errors when requested', function(done) {
    request(app)
      .get('/notes/999')
      .set('Accept', 'application/json')
```

1 Check that expected status code was returned

2 Set Accept header to get JSON

```
        .expect(404, function(err, res) {
          assert.equal(res.body.name, 'NotFound');
          done();
        });
    });
});
```

❸ Check that body was in expected format

This snippet includes two requests. The first checks that we get an error with a 404 ❶, and the second sets the `Accept` header to make sure we get back JSON ❷. This is implemented with SuperTest, which will give us JSON in responses, so the assertion can check to make sure we get an object in the format we expect ❸. The full source for this example can be found in listings/web/error-handling.

Error email cheat sheet

If you're going to make your application send email notifications when unexpected errors occur, here's a list of things you should include in the email to aid with debugging:

- A string version of the error object
- The contents of `err.stack`—this is a nonstandard property of error objects that Node includes
- The request method and URL
- The Express `req.route` property, if available
- The remote IP, which is `req.ip` in Express
- The request body, which you can convert to a string with `inspect(req.body)`

This error-handling pattern is widely used in Express apps, and it's even built into the restify framework (https://npmjs.org/package/restify). If you remember to pass error objects to `next`, you'll find testing and debugging Express applications easier.

Errors can also be sent as emails with useful transcripts. To make the most out of error emails, include the request and error objects in the email so you can see exactly where things broke. Also, you probably don't want to send details about errors with certain status codes, but that's up to you.

In this technique we mentioned adapting code to work with REST APIs. The next technique delves deeper into the world of REST, and has examples for both Express and restify.

TECHNIQUE 71 RESTful web applications

At some stage you might want to add an API to your application. This technique is all about building RESTful APIs. There are examples for both Express and restify, and tips on how to create APIs that use the right HTTP verbs and idiomatic URLs.

Problem

You want to create a RESTful web service in Express, restify, or another web framework.

Solution

Use the right HTTP methods, URLs, and headers to build an intuitive, RESTful API.

Discussion

REST stands for *representational state transfer*,[2] which isn't terribly useful to memorize unless you want to impress someone in a job interview. The way web developers talk about it is usually in contrast to SOAP (Simple Object Access Protocol), which is seen as a more corporate and strict way to create web APIs. In fact, there's such a thing as a strict REST API, but the key distinction is that REST embraces HTTP at a fundamental level—the HTTP methods themselves have semantic meaning.

You should be familiar with using `GET` and `POST` requests if you've ever made a basic HTML form. In REST, these HTTP verbs have specific meanings. For example, `POST` will *create a resource*, and `GET` means *fetch a resource*.

Node developers typically create APIs that use JSON. JSON is the easiest structured data format to generate and read in Node, but it also works well in client-side JavaScript. But REST doesn't imply JSON—you're free to use any data format. Certain clients and services expect XML, and we've even seen those that work with CSV and spreadsheet formats like Excel.

The desired data format is specified by the request's `Accept` header. For JSON that should be `application/json`, and `application/xml` for XML. There are other useful request headers as well—`Accept-Version` can be used to request a different version of the API. This allows clients to lock themselves against a supported version, while you're free to improve the server without breaking backward compatibility—you can always update your server faster than people can update their clients.

Express provides a lightweight layer over Node's `http` core module, but it doesn't include any data persistence functionality outside of in-memory sessions and cookies. You'll have to decide which database and database module to use. The same is true with restify: it doesn't automatically map data from HTTP to be stored offline; you'll need to find a way to do that.

Restify is superficially similar to Express. The difference is that Express has features that help you build web applications, which includes rendering templates. Conversely, restify is focused on building REST APIs, and that brings a different set of requirements. Restify makes it easy to serve multiple versions of an API with semantic versioning using HTTP headers, and has an event-based API for emitting and listening for HTTP-related events and errors. It also supports throttling, so you can control how quickly responses are made.

Figure 9.3 shows a typical RESTful API that allows *page* objects to be created, read, updated, and deleted.

To get started building REST APIs, you should consider what your objects are. Imagine you're building a content management system: it probably has pages, users, and images. If you want to add a button that allows pages to be toggled between "published" and "draft," and if you've already got a REST API and it supports requests to `PATCH /pages/:id`, you could just tie the button to some client-side JavaScript or a form that posts to `/pages/:id` with `{ state: 'published' }` or `{ state: 'draft' }`. If

[2] For more about REST, see Fielding's dissertation on the subject at http://mng.bz/7Fhj.

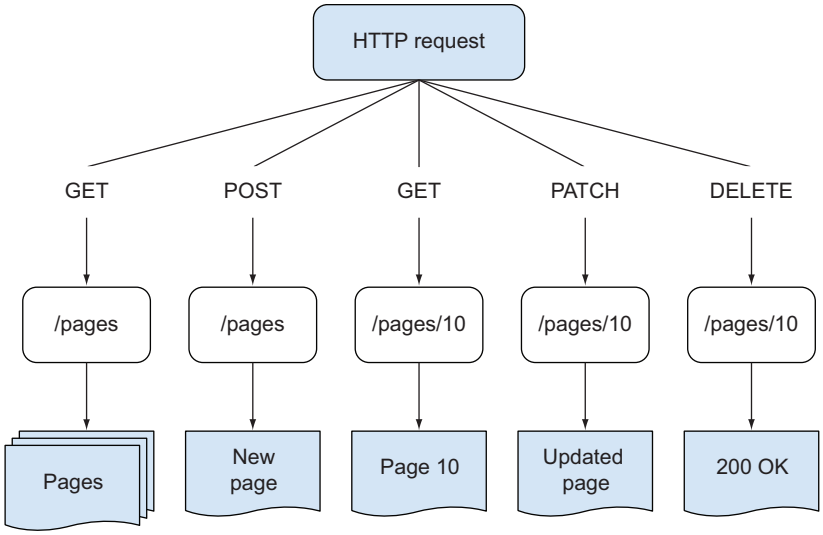


Figure 9.3 Making requests to a REST API

you've been given an Express application that only has `PUT /pages/:id`, then you could probably derive the code for `PATCH` from the existing implementation.

Plural or singular?

When you design your API's URI endpoints, you should generally use *plural nouns*. That means `/pages` and also `/pages/1` for a specific page, not `/page/1`. It'll be easier to use your API if the endpoints are consistent.

You may find there are certain resources that should be singular nouns, because there's only ever one such item. If it makes semantic sense, use a singular noun, but use it consistently. For example, if your API requires that users sign in, and you don't want to expose a unique user ID, then `/account` might be a sensible endpoint for user account management, if there's only ever one account for a given user.

Table 9.1 shows HTTP verbs alongside the typical response. Note that `PUT` and `PATCH` have different but similar meanings—`PATCH` means modify some of the fields in a resource, while `PUT` means *replace* the entire resource. It can take some practice to get the hang of building applications this way, but it's pragmatic and easy to test, so it's worth learning properly. If these HTTP terms are new to you, then use table 9.1 when you're designing the API for your application.

In an Express application, these URLs and methods are mapped using routes. Routes specify the HTTP verb and a partial URL. You can map these to any function that you like, but if you use the route separation pattern from technique 67, which is advisable, then you should use the method names that are close to their associated

Table 9.1 Choosing the correct HTTP verbs

Verb	Description	Response
`GET /animals`	Get a list of animals.	An array of animal objects
`GET /animals/:id`	Get a single animal.	A single animal object, or an error
`POST /animals`	Create an animal by sending the properties of a single animal.	The new animal
`PUT /animals/:id`	Update a single animal record. All properties will be replaced.	The updated animal
`PATCH /animals/:id/`	Update a single animal record, but only change the fields specified.	The updated animal

HTTP verbs. Listing 9.18 shows the routes for a RESTful resource in Express, and some of the required configuration to make it work.

Listing 9.18 A RESTful resource in Express

```
var app;
var express = require('express');
var routes = require('./routes');

module.exports = app = express();

app.use(express.json());
app.use(express.methodOverride());

app.get('/pages', routes.pages.index);
app.get('/pages/:id', routes.pages.show);
app.post('/pages', routes.pages.create);
app.patch('/pages/:id', routes.pages.patch);
app.put('/pages/:id', routes.pages.update);
app.del('/pages/:id', routes.pages.remove);
```

1 Use JSON body parser

methodOverride middleware component allows a query parameter to specify extra HTTP methods 2

3 The routes for the resource

This example uses some middleware for automatically parsing JSON requests 1, and overrides the HTTP method `POST` with the query parameter, `_method` 2. That means that the `PUT`, `PATCH`, and `DELETE` HTTP verbs are actually determined by the `_method` query parameter. This is because most browsers can only send a `GET` or `POST`, so `_method` is a hack used by many web frameworks.

The routes in listing 9.18 define each of the usual RESTful resource methods 3. Table 9.1 shows how these routes map to actions.

Table 9.2 Mapping routes to responses

Verb, URL	Description
`GET /pages`	An array of pages.
`GET /pages/:id`	An object containing the page specified by `id`.

Table 9.2 Mapping routes to responses *(continued)*

Verb, URL	Description
`POST /pages`	Create a page.
`PATCH /pages/:id`	Load the page for `id`, and change some of the fields.
`PUT /pages/:id`	Replace the page for `id`.
`DELETE /pages/:id`	Remove the page for `id`.

Listing 9.19 is an example implementation for the route handlers. It has a generic Node database API—a real Redis, MongoDB, MySQL, or PostgreSQL database module wouldn't be too far off, so you should be able to adapt it.

Listing 9.19 RESTful route handlers

```
var db = require('./../db');

module.exports.index = function(req, res, next) {
  db.pages.findAll(function(err, pages) {
    if (err) return next(err);
    res.send(pages);
  });
};

module.exports.create = function(req, res, next) {
  var page = req.body.page;
  db.pages.create(page, function(err, page) {
    if (err) return next(err);
    res.send(page);
  });
};

module.exports.update = function(req, res, next) {
  var id = req.param('id');
  var page = req.body.page;
  db.pages.update(id, page, function(err, page) {
    if (err) return next(err);
    res.send(page);
  });
};

module.exports.show = function(req, res, next) {
  db.pages.find(req.param('id'), function(err, page) {
    if (err) return next(err);
    res.send(page);
  });
};

module.exports.patch = function(req, res, next) {
  var id = req.param('id');
  var page = req.body.page;
```

1 **Fall through to next middleware component when an error is raised by the database**

2 **Calling send will automatically return JSON to browser**

```
  db.pages.patch(id, page, function(err, page) {
    if (err) return next(err);
    res.send(page);
  });
};

module.exports.remove = function(req, res, next) {
  var id = req.param('id');
  db.pages.remove(id, function(err) {
    if (err) return next(err);
    res.send(200);
  });
};
```

❸ Most database modules won't have method named patch, but something similar

Although this example is simple, it illustrates something important: you should keep your route handlers lightweight. They deal with HTTP and then let other parts of your code handle the underlying business logic. Another pattern used in this example is the error handling—errors are passed by calling `next(err)` ❶. Try to keep error-handling code centralizing and generic—technique 70 has more details on this.

To return the JSON to the browser, `res.send()` is called with a JavaScript object ❷. Express knows how to convert the object to JSON, so that's all you need to do.

All of these route handlers use the same pattern: map the query or body to something the database can use, and then call the corresponding database method. If you're using an ORM or ODM—a more abstracted database layer—then you'll probably have something analogous to `PATCH` ❸. This could be an API method that allows you to update only the specified fields. Relational databases and MongoDB work that way.

If you download this book's source code, you'll get the other files required to try out the full example. To run it, type `npm start`. Once the server is running, you can use some of the following Curl commands to communicate with the server.

The first command creates a page:

❶ Use JSON as body encoding

Method is POST ❷

```
curl -H "Content-Type: application/json" \
  -X POST -d '{ "page": { "title": "Home" } }' \
  http://localhost:3000/pages
```

❸ URL for creating pages is /pages

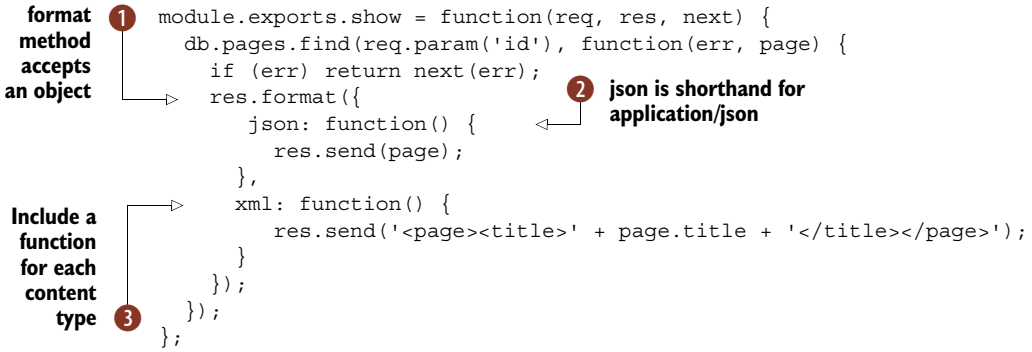
First we specify the `Content-Type` using the `-H` option ❶. Next, the request is set to use `POST`, and the request body is included as a JSON string ❷. The URL is `/pages` because we're creating a resource ❸.

Curl is a useful tool for exploring APIs, once you understand the basic options. The ones to remember are `-H` for setting headers, `-X` for setting the HTTP method, and `-d` for the request body.

To see the list of pages, just use `curl http://localhost:3000/pages`. To change the contents, try `PATCH`:

```
curl -H "Content-Type: application/json" \
  -X PATCH -d '{ "page": { "title": "The Moon" } }' \
  http://localhost:3000/pages/1
```

Express has a few other tricks up its sleeves for creating RESTful web services. Remember that some REST APIs use other data formats, like XML? What if you want both? You can solve this by using `res.format`:




```
module.exports.show = function(req, res, next) {
  db.pages.find(req.param('id'), function(err, page) {
    if (err) return next(err);
    res.format({
       json: function() {
         res.send(page);
       },
       xml: function() {
         res.send('<page><title>' + page.title + '</title></page>');
       }
    });
  });
};
```

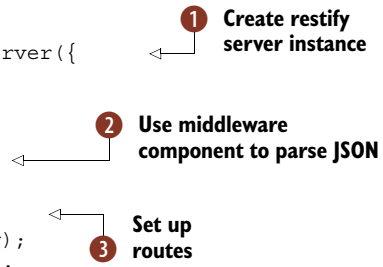
To use XML instead of JSON, you have to include the `Accept` header in the request. With Curl, you can do this:

```
curl -H 'Accept: application/xml' \
  http://localhost:3000/pages/1
```

Just remember that `Accept` is used to ask the server for a specific format, and `Content-Type` is used to tell the server what format you're sending it. It sometimes makes sense to include both in a single request!

Now that you've seen how REST APIs in Express work, we can compare them with restify. The patterns used to structure Express applications can be reused for restify projects. The two important patterns are route separation, as described in technique 67, and defining the application in a separate file to the server (for easier testing and internal reuse). Listing 9.20 is the restify equivalent of listing 9.18.

Listing 9.20 A restify application




```
var app;
var restify = require('restify');
var routes = require('./routes');

module.exports = app = restify.createServer({
  name: 'NIP CMS',
});

app.use(restify.bodyParser());

app.get('/pages', routes.pages.index);
app.get('/pages/:id', routes.pages.show);
app.post('/pages', routes.pages.create);
app.patch('/pages/:id', routes.pages.patch);
app.put('/pages/:id', routes.pages.update);
app.del('/pages/:id', routes.pages.remove);
```

Using restify, instances of servers are created with some initial configuration options (1). You don't have to pass in any options, but here we've specified a name. The options are actually the same as Node's built-in `http.Server.listen`, so you can pass in options for SSL/TLS certificates, if you want to use encryption. Restify-specific options that aren't available in Express include `formatters`, which allows you to set up functions that `res.send` will use for custom content types.

This example uses `bodyParser` to parse JSON in the request bodies (2). This is like the Express middleware component in the previous example.

The route definitions are identical to Express (3). The actual route callbacks are slightly different. Listing 9.21 shows a translation of listing 9.19. See if you can spot the differences.

Listing 9.21 Restify routes

```
var db = require('./../db');

module.exports.index = function(req, res, next) {
  db.pages.findAll(function(err, pages) {
    if (err) return next(err);
    res.send(pages);
  });
};

module.exports.create = function(req, res, next) {
  var page = req.body.page;
  db.pages.create(page, function(err, page) {
    if (err) return next(err);
    res.send(page);
  });
};

module.exports.update = function(req, res, next) {
  var id = req.params.id;
  var page = req.body.page;
  db.pages.update(id, page, function(err, page) {
    if (err) return next(err);
    res.send(page);
  });
};

module.exports.show = function(req, res, next) {
  db.pages.find(req.params.id, function(err, page) {
    if (err) return next(err);
    res.send(page);
  });
};

module.exports.patch = function(req, res, next) {
  var id = req.params.id;
  var page = req.body.page;
  db.pages.patch(id, page, function(err, page) {
    if (err) return next(err);
```

(1) The callback arguments are similar to Express.

(2) Getting URL parameters is slightly different.

```
    res.send(page);
  });
};

module.exports.remove = function(req, res, next) {
  var id = req.params.id;
  db.pages.remove(id, function(err) {
    if (err) return next(err);
    res.send(200);
  });
};
```

❸ **Passing an integer to send() returns the status code.**

The first thing to note is the callback arguments for route handlers are the same as Express ❶. In fact, you can almost lift the equivalent code directly from Express applications. There are a few differences though: `req.param()` doesn't exist—you need to use `req.params` instead, and note this is an object rather than a method ❷. Like Express, calling `res.send()` with an integer will return a status code to the client ❸.

Using other HTTP headers

In this technique you've seen how the `Content-Type` and `Accept` headers can be used to deal with different data formats. There are other useful headers that you should take into account when building APIs.

One such header, supported by restify, is `Accept-Version`. When you define a route, you can include an optional first parameter that includes options, instead of the usual string. The `version` property allows your API to respond differently based on the `Accept-Version` header.

For example, using `app.get({ path: '/pages', version: '1.1.8' }, routes.v1.pages);` allows you to bind specific route handlers to version 1.1.8. If you have to change your API in 2.0.0, then you can do this without breaking older clients.

There's nothing to stop you from using this header in an Express application, but it's easier in restify. If you decide to take this approach, you should learn how *major.minor.patch* works in semantic versioning (http://semver.org/).

If you download the full example and run it (listings/web/restify), you can try out some of the Curl commands we described earlier. Create, update, and show should work the same way.

Knowing that Express and restify applications are similar is useful, because you can start to compose applications made from both frameworks. Both are based on Node's `http` module, which means you could technically mount a restify application inside Express using `app.use(restifyApp)`. This works well if the restify application is in its own module—you could install it using npm, or put it in its own directory.

Both Express and restify use middleware, and you'll find well-structured applications have loosely coupled middleware that can be reused across different projects. In

the next technique you'll see how to write your own middleware, so you can start decorating applications with useful features like custom logging.

TECHNIQUE 72 Using custom middleware

You've seen middleware being used for error handling, and you've also used some of Express's built-in middleware. You can also use middleware to add custom behavior to routes; this might add new functionality, improve logging, or control access based on authentication or permissions.

The benefit of middleware is that it can improve code reuse in your application. This technique will teach you how to write your own middleware, so you can share code between projects, and structure projects in a more readable way.

Problem

You want to add behavior—in a reusable, testable manner—that's triggered when certain routes are accessed.

Solution

Write your own middleware.

Discussion

When you first start using Express, middleware sounds like a complicated concept that other people use for writing plugins that extend Express. But in fact, writing middleware is a fundamental part of using Express, and you should start writing middleware as soon as possible. And if you can write routes, then you can write middleware: it's basically the same API!

In technique 70, you saw how to handle errors with a middleware component. Error handling is a special case—you have to include a fourth parameter to capture the error object: `app.use(function(err, req, res, next) {`. With other middleware, you can just use three arguments, like standard route handlers. This is the simplest middleware component:

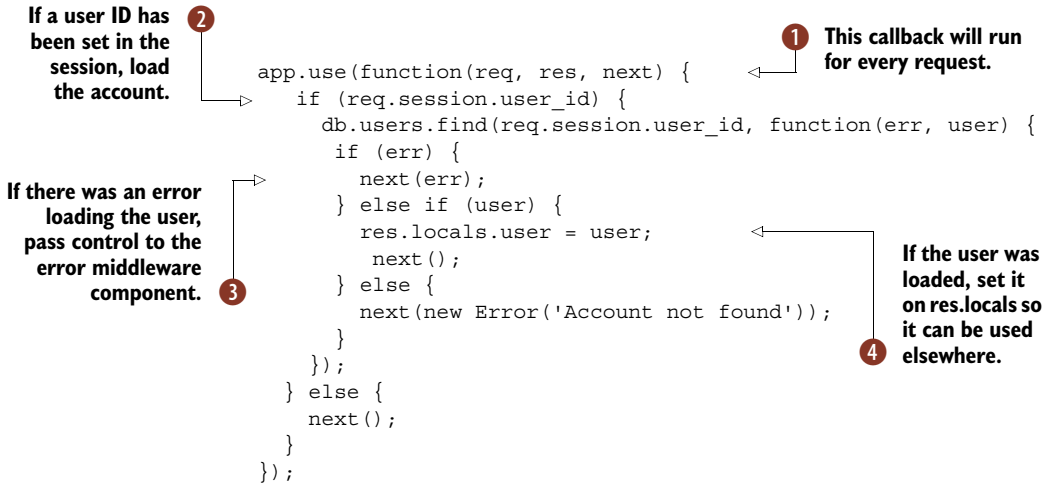
```
app.use(function(req, res, next) {
  console.log('%s %s', req.method, req.url);
  next();
});
```

1 Apply middleware component by calling app.use()

2 Call next() to continue execution to next middleware component

By passing an anonymous callback to `app.use` 1, the middleware component will always run, unless a previous middleware component fails to call `next`. When your code is finished, you can call `next` 2 to trigger the next middleware component in the stack. That means two things: asynchronous APIs are supported, and the order in which you add middleware is important.

The following example shows how you can use asynchronous APIs inside middleware. This example is based on the idea of loading a user based on a user ID that has been set in the session:




```
app.use(function(req, res, next) {
  if (req.session.user_id) {
    db.users.find(req.session.user_id, function(err, user) {
      if (err) {
        next(err);
      } else if (user) {
        res.locals.user = user;
        next();
      } else {
        next(new Error('Account not found'));
      }
    });
  } else {
    next();
  }
});
```

This middleware will be triggered for every request 1. It loads user accounts from a database, but only when the user's ID has been set in the session 2. The code that loads the user is asynchronous, so `next` could be called after a short delay. There are several points where `next` is called: for example, if an error was encountered when loading the user, `next` will be called with an error 3.

In this example the loaded user is set as a property of `res.locals` 4. By using `res.locals`, you'll be able to access the user in other middleware, route handlers, and templates.

This isn't necessarily the best way to use middleware. Including an anonymous function this way means it can be hard to test—you can only test middleware by starting up the entire Express application. You might want to write simpler unit tests that don't use HTTP requests, so it would be better to refactor this code into a function. The function would have the same signature, and would be used like this:

```
var middleware = require('./middleware');
app.use(middleware.loadUser);
```

1 Loading a module that contains middleware

By grouping all the middleware together as modules 1, you can load the middleware from other locations, whether they're entirely different projects, test code, or inside separated routes. This function has decoupled the middleware to improve how it can be reused.

If you're using the route separation pattern from technique 67, then this makes sense, because middleware can be applied to specific routes that might be defined in different files. Let's say you're using the RESTful API style from technique 71, and your *page* resource can only be updated by signed-in users, but other parts of the application should be accessible to anyone. You can restrict access to the page resource routes like this:

```
var middleware = require('./middleware');

app.get('/pages', routes.pages.index);
app.get('/pages/:id', routes.pages.show);
app.post('/pages', middleware.loadUser, routes.pages.create);
app.patch('/pages/:id', middleware.loadUser, routes.pages.patch);
```

1 Anyone can view pages.

2 Only signed-in users can create or update pages.

In this fragment, routes are defined for a resource called `pages`. Some routes are accessible to anyone 1, but creating or updating pages is limited to people with accounts on the system 2. This is done by supplying the `loadUser` middleware component as the second argument when defining a route. In fact, multiple arguments could be used—you could have a generic user loading route, and then a more specific permission checking route that ensures users are administrators, or have the necessary rights to change pages.

Figure 9.4 shows how requests can pass through several callbacks until the final response is sent back to the client. Sometimes this might cause a response to finish before other middleware has had a chance to run—if an error is encountered and passed to `next(err)`.

You can even apply middleware to batches of routes. It's common to see something like `app.all('/admin/*', middleware.loadUser);` in Express applications.

If you use modules to manage your middleware, and simplify route handlers by moving shared functionality into separate files, then you'll find that organizing middleware into modules becomes a fundamental architectural tool for organizing applications.

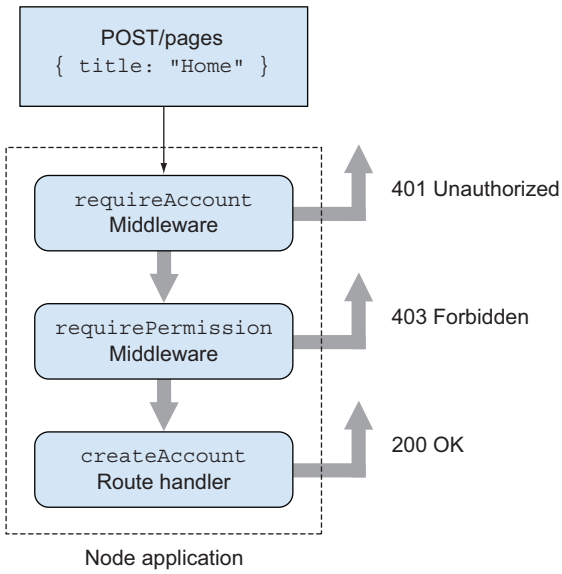


Figure 9.4 Requests can pass through several callbacks until the final response is sent.

If you're designing a new Express application, you should think in terms of middleware. Ask yourself what kinds of HTTP requests you're going to deal with, and what kinds of filtering they might need.

Now it's time to combine all of these ideas into a worked example. Listing 9.22 demonstrates one way of parsing requests that contain XML. Middleware has been used to parse the XML, turning it into plain old JavaScript objects. That means two things: only a small part of your code has to worry about XML, and you could potentially add support for other data formats as well.

Listing 9.22 Three types of middleware

```
var express = require('express');
var app = express();
var Schema = require('validate');
var xml2json = require('xml2json');
var util = require('util');
var Page = new Schema();

Page.path('title').type('string').required();

function ValidatorError(errors) {
  this.statusCode = 400;
  this.message = errors.join(', ');
}
util.inherits(ValidatorError, Error);

function xmlMiddleware(req, res, next) {
  if (!req.is('xml')) return next();

  var body = '';
  req.on('data', function(str) {
    body += str;
  });

  req.on('end', function() {
    req.body = xml2json.toJson(body.toString(), {
      object: true,
      sanitize: false
    });
    next();
  });
}

function checkValidXml(req, res, next) {
  var page = Page.validate(req.body.page);
  if (page.errors.length) {
    next(new ValidatorError(page.errors));
  } else {
    next();
  }
}

function errorHandler(err, req, res, next) {
```

1. Define some data validation to ensure pages have titles
2. Inherit from standard error object so validation errors can be handled in error middleware component
3. This function will be used as XML middleware component
4. Request object will emit data events when body is read from the client
5. Data-validation middleware component
6. Passing errors to next() will stop route handler from running
7. This is error-handling middleware component

```
  console.error('errorHandler', err);
  res.send(err.statusCode || 500, err.message);
}

app.use(xmlMiddleware);

app.post('/pages', checkValidXml, function(req, res) {
  console.log('Valid page:', req.body.page);
  res.send(req.body);
});

app.use(errorHandler);

app.listen(3000);
```

8 Use XML middleware component for all requests

9 Validate XML for specific requests

10 Last middleware component to be added should be error handler

In summary, this example defines three middleware components to parse XML, validate it, and then either respond with a JSON object or display an error. We've used an arbitrary data-validation library here 1—your database module may come with something similar.

The routes deal with *page* resources, and the expected format for pages is XML. It's passed in as request bodies and validated. An error object, `ValidatorError` 2, is used to return a 400 error when invalid data is sent to the server. The XML parser 3 reads in the request body using the standard event-based API 4. This middleware component is called for every request 8 because it's passed directly to `app.use`, but it only runs if the `Content-Type` is set to XML.

The data-validation middleware component 5 ensures a page title has been set—this is just an arbitrary example we've chosen to illustrate how this kind of validation works. If the data is invalid, an instance of `ValidatorError` is passed when `next` is called 6. This will trigger the error-handling middleware component 7.

Data is only validated for certain requests. This is done by passing `checkValidXml` when the /pages route is defined 9.

The global error handler is the last middleware component to be added 10. This should always be the case, because middleware is executed in the order it's defined. Once `res.send` has been called, then no more processing will occur, so errors won't be triggered.

To try this example out, run `node server.js` and then try posting XML to the server using `curl`:

```
curl -H "Content-Type: application/xml" \
  -X POST -d '<page><title>Node in Practice</title></page>' \
  http://localhost:3000/pages
```

You should try leaving out a title to ensure a 400 error is raised!

This approach can be used for XML, JSON, CSV, or any other data formats you like. It works well for minimizing the code that has to deal with XML, but there are other ways you can write decoupled code in Node web applications. In the next technique you'll see how something fundamental to Node—events—can be used as another useful architectural pattern.

TECHNIQUE 73 Using events to decouple functionality

In the average Express application, most code is organized into methods and modules. This can make sharing functionality inconvenient in some cases, particularly if you want to neatly separate concerns within your application. This technique uses sending emails as an example of something that doesn't fit neatly into routers, models, or views. Events are used to decouple emails from routers, which keeps email-related code outside of HTTP code.

Problem

You want to do things that aren't related to HTTP, like send emails, but aren't sure how to structure the code so it's neatly decoupled and easy to test.

Solution

Use easily accessible `EventEmitter` objects, like the Express `app` object.

Discussion

Express and restify applications generally follow the Model-View-Controller (MVC) pattern. Models are used to save data, controllers are route handlers, and views are the templates in the views/ directory.

Some code doesn't fit neatly into these categories. For example, where would you keep email-handling code? Email generation clearly doesn't belong in routes, because email isn't related to HTTP. But like route handlers, it does require templates. It also isn't really a model, because it doesn't interact with the database.

What if you did put the email-handling code into models? In that case, given an instance of a `User` model, you want to send an email when a new account is created. You could put the email code in the `User.prototype.registerUser` method. The problem with that is you might not always want to send emails when users are created. It might not be convenient during testing, or some kind of periodic maintenance tasks.

The reason why sending email isn't quite suitable for models or HTTP routes can be understood by thinking about the SOLID principles (http://en.wikipedia.org/wiki/SOLID). There are two principles that are relevant to us: the *single responsibility principle* and the *dependency inversion principle*.

Single responsibility dictates that the class that deals with HTTP routes really shouldn't send emails, because these are different responsibilities that shouldn't be mixed together. Inversion of control is a specific type of dependency inversion, and can be done by removing direct invocation—rather than calling `emails.sendAccountCreation`, your email-handling class should respond to events.

For Node programmers, events are one of the most important tools available. And fortunately for us, the SOLID principles indicate that we can write better HTTP routers by removing our email code, and replacing it with abstract and generalized events. These events can then be responded to by the relevant classes.

Figure 9.5 shows what our idealized application structure might look like. But how do we achieve this? Take Express applications as an example; they don't typically have a suitable global event object. You could technically create a global variable somewhere

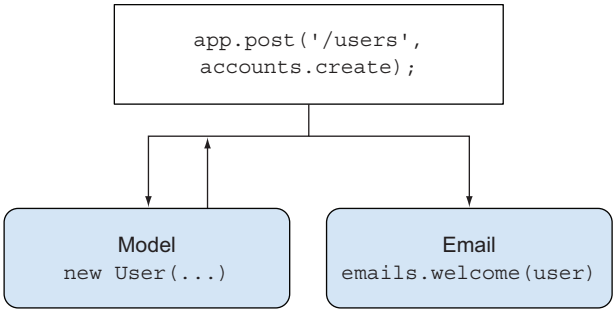


Figure 9.5 Applications can be easier to understand if organized according to the SOLID principles.

central, like the file that calls `express()`, but that would introduce a global shared state, and that would break the principles we described earlier.

Fortunately, Express includes a reference to the `app` object in the request. Route handlers, which accept the `req, res` parameters, always have access to `app` in `res.app`. The `app` object inherits from `EventEmitter`, so we can use it to broadcast when things happen. If your route handler creates and saves new users, then it can also call `res.app.emit('user:created', user)`, or something similar—you can use any naming scheme for events as long as it's consistent. Then you can listen for `user:created` events and respond accordingly. This could include sending email notifications, or perhaps even logging useful statistics about users.

The following listing shows how to listen for events on the application object.

Listing 9.23 Using events to structure an application

```
var express = require('express');
var app = express();
var emails = require('./emails');
var routes = require('./routes');

app.use(express.json());

app.post('/users', routes.users.create);

app.on('user:created', emails.welcome);

module.exports = app;
```

1 **Set up a route for creating users.**

2 **Listen for user creation events, and bind them to the email code.**

In this example a route for registering users is defined 1, and then an event listener is defined and bound to a method that sends emails 2.

The route is shown in the next listing.

Listing 9.24 Emitting events

```
var User = require('./../models/user');

module.exports.create = function(req, res, next) {
  var user = new User(req.body);
```

```
  user.save(function(err) {
    if (err) return next(err);
    res.app.emit('user:created', user);
    res.send('User created');
  });
};
```

Emit user creation events when users are successfully registered.

This listing contains an example model for `User` objects. If a user is successfully created, then `user:created` is emitted on the `app` object. The downloadable code for this book includes a more complete example with the code that sends emails, but the basic principle for removing direct invocation and adhering to the single responsibility principle is represented here.

Communication with events inside applications is useful when you need to make the code easier for other developers to understand. There are also times when you need to communicate with client-side code. The next technique shows you how to take advantage of WebSockets in your Node applications, while still being able to access resources like sessions.

TECHNIQUE 74 Using sessions with WebSockets

Node has strong support for the real-time web. Adopting event-oriented, asynchronous APIs means supporting WebSockets is a natural fit. Also, it's trivial to run two servers in the same process: a WebSocket server and a standard Node HTTP server can coexist happily.

This technique shows you how to reuse the Connect and Express middleware that we've been using so far with a WebSocket server. If your application allows users to sign in, and you want to add WebSocket support, then read on to learn how to master sessions in WebSockets.

Problem

You want to add WebSocket support to an existing Express application, but you're not sure how to access session variables, like whether the user is currently signed in.

Solution

Reuse Connect's cookie and session middleware with your WebSocket server.

Discussion

This technique assumes you have a passing familiarity with WebSockets. To recap: HTTP requests are stateless and relatively short-lived. They're great for downloading documents, and requesting a state change for a resource. But what about streaming data to and from a server?

Certain types of events originate from servers. Think about a web mail service. When you create and send a message, you push it to the server, and the server sends it to the recipients. If the recipient is sitting watching their inbox, there's no easy way for their browser to get updated. It could periodically check for new messages using an Ajax request, but this isn't very elegant. The server *knows* it has a new message for the recipient, so it would be much better if it could push that message directly to the user.

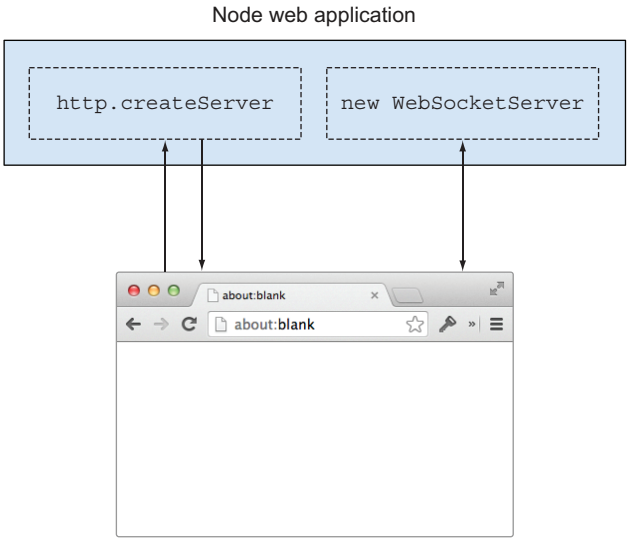


Figure 9.6 A Node web application should support both standard HTTP requests and WebSockets.

That's where WebSockets come in. They're conceptually like the TCP sockets we saw in chapter 7: a bidirectional bridge is set up between the client and server. To do this you need a WebSocket server in addition to your standard Express server, or plain old Node `http` server. Figure 9.6 illustrates how this works in a typical Node web application.

HTTP requests are short-lived, have specific endpoints, and use methods like `POST` and `PUT`. WebSockets are long-lived, don't have specific endpoints, and don't have methods. They're conceptually different, but since they're used to communicate with the same application, they typically need access to the same data.

This presents a problem for sessions. The Express examples we've looked at used middleware to automatically load the session. Connect middleware is based on the HTTP request and response, so how do we map this to WebSockets, which are long-lived and bidirectional? To understand this, we need to look at how WebSockets and sessions work.

Sessions are loaded based on unique identifiers that are included in cookies. Cookies are sent with every HTTP request. WebSockets are initiated with a standard HTTP request that asks to be upgraded to a WebSocket. This means there's a point where you can grab the cookie from the request, and then load the session. For each WebSocket, you can store a reference to the user's session. Now you can do all the usual things you need to do with a session: verify the user is signed in, set preferences, and so on.

Figure 9.7 extends figure 9.6 to show how sessions can be used with WebSockets, by incorporating the Connect middleware for parsing cookies and loading the session.

Now that you know how the parts fit together, how do you go about building it? The cookie-parsing middleware component can be found in `express.cookieParser`. This is actually a simple method that gets the cookie from the request headers, and then parses the cookie string into separate values. It accepts an argument, `secret`,

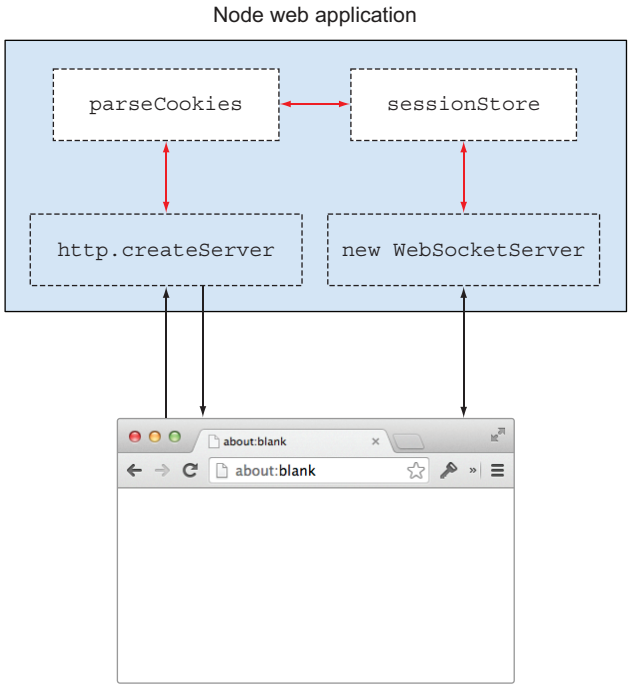


Figure 9.7 Accessing sessions from WebSockets

which is the value used to sign the cookie. Once the cookie is decrypted, you can get the session ID from it and load the session.

Sessions in Express are modeled on an asynchronous API for storing and retrieving values. They can be backed by a database, or you can use the built-in memory-based class. Passing the session ID and a callback to `sessionStore.get` will load the session, if the session ID is correct.

In this technique we'll use the `ws` WebSocket module (https://www.npmjs.org/package/ws). This is a fast-but-minimal implementation that has a very different API than Socket.IO. If you want to learn about Socket.IO, then *Node in Action* has some excellent tutorials. Here we're using a simpler module so you can really see how WebSockets work.

To make `ws` load the session, you need to parse the cookies from the HTTP upgrade request, and then call `sessionStore.get`. A full example that shows how it all works follows.

Listing 9.25 An Express application that uses WebSockets

```
var express = require('express');
var WebSocketServer = require('ws').Server;
var parseCookie = express.cookieParser('some secret');
var MemoryStore = express.session.MemoryStore;
var store = new MemoryStore();
```

Load cookie-parser middleware component and set the secret (1)

(2) **Load desired session store**

```
var app = express();
var server = app.listen(process.env.PORT || 3000);
var webSocketServer;

app.use(parseCookie);
app.use(express.session({ store: store, secret: 'some secret' }));
app.use(express.static(__dirname + '/public'));

app.get('/random', function(req, res) {
  req.session.random = Math.random().toString();
  res.send(200);
});

webSocketServer = new WebSocketServer({ server: server });

webSocketServer.on('connection', function(ws) {
  var session;

  ws.on('message', function(data, flags) {
    var message = JSON.parse(data);

    if (message.type === 'getSession') {
      parseCookie(ws.upgradeReq, null, function(err) {
        var sid = ws.upgradeReq.signedCookies['connect.sid'];

        store.get(sid, function(err, loadedSession) {
          if (err) console.error(err);
          session = loadedSession;
          ws.send('session.random: ' + session.random, {
            mask: false
          });
        });
      });
    } else {
      ws.send('Unknown command');
    }
  });
});
```

Tell Express to use session store, and set the secret 3

Create Express route that will set a session value for testing 4

Start up WebSocket server, and pass it the Express server 5

On connection events, create WebSocket for the client 6

Data sent by the client is assumed to be JSON, and is parsed here 7

Get session ID for WebSocket from the HTTP upgrade request 8

Get user's session from the store 9

Send value from session back through the WebSocket 10

This example starts by loading and configuring the cookie parser 1 and the session store 2. We're using signed cookies, so note that `ws.upgradeReq.signedCookies` is used when loading the session later.

Express is set up to use the session middleware component 3, and we've created a route that you can use for testing 4. Just load http://localhost:3000/random in your browser to set a random value in the session, and then visit http://localhost:3000/ to see it printed back.

The `ws` module works by using a plain old constructor, `WebSocketServer`, to handle WebSockets. To use it, you instantiate it with a Node HTTP server object—we've just passed in the Express server here 5. Once the server is started, it'll emit events when connections are created 6.

The client code for this example sends JSON to the server, so there's some code to parse the JSON string and check whether it's valid 7. This wasn't entirely necessary

for this example, but we included it to show that ws requires this kind of extra work to be used in most practical situations.

Once the WebSocket server has a connection, the session ID can be accessed through the cookies on the upgrade request (8). This is similar to what Express does behind the scenes—we just need to manually pass a reference to the upgrade request to the cookie-parser middleware component. Then the session is loaded using the session store's get method (9). Once the session has been loaded, a message is sent back to the client that contains a value from the session (10).

The associated client-side implementation that's required to run this example is shown in the following listing.

Listing 9.26 The client-side WebSocket implementation

```
<!DOCTYPE html>
<html>
<head>
<script>
var host = window.document.location.host.replace(/:.*/, '');
var ws = new WebSocket('ws://' + host + ':3000');

setInterval(function() {
  ws.send('{ "type": "getSession" }');
}, 1000);

ws.onmessage = function(event) {
  document.getElementById('message').innerHTML = event.data;
};
</script>
</head>
<body>
  <h1>WebSocket sessions</h1>
  <div id='message'></div><br>
</body>
</html>
```

Periodically send message to the server

All it does is periodically send a message to the server. It'll display undefined until you visit http://localhost:3000/random. If you open two windows, one to http://localhost:3000/random and the other to http://localhost:3000/, you'll be able to keep refreshing the random page so the WebSocket view shows new values.

Running this example requires Express 3 and ws 0.4—we've included a package.json with everything you need in the book's full listings.

The next technique has tips for migrating from Express 3 to Express 4.

TECHNIQUE 75 Migrating Express 3 applications to Express 4

This book was written before Express 4 was released, so our Express examples are written with version 3 of the framework in mind. We've included this technique to help you migrate, and also so you can see how version 4 differs from the previous versions.

Problem

You have an Express 3 application and want to upgrade it to use Express 4.

Solution

Update your application configuration, install missing middleware, and take advantage of the new routing API.

Discussion

Most of the updates from Express 3 to 4 were a long time coming. Certain changes have been hinted at in Express 3's documentation, so the API changes weren't unexpected or even too dramatic for the most part. You'll probably spend most of your time replacing the middleware that used to ship with Express, because Express 4 no longer has any built-in middleware components, apart from `express.static`.

The `express.static` middleware component enables Express to mount your `public` folder that contains JavaScript, CSS, and image assets. This has been left in because it's convenient, but the rest of the middleware components have gone. That means you'll need to use `npm install --save body-parser` if you previously used `bodyParser`, for example. Refer to table 9.1 that has the old middleware names and the newer equivalents. Just remember that you need to `npm install --save` each one that you need, and then `require` it in your app.js file.

Table 9.3 Migrating Express middleware components

Express 3	Express 4 npm package	Description
`bodyParser`	`body-parser`	Parses URL-encoded and JSON POST bodies
`compress`	`compression`	Compresses the server's responses
`timeout`	`connect-timeout`	Allows requests to timeout if they take too long
`cookieParser`	`cookie-parser`	Parses cookies from HTTP headers, leaving the result in `req.cookies`
`cookieSession`	`cookie-session`	Simple session support using cookies
`csrf`	`csurf`	Adds a token to the session that you can use to protect forms from CSRF attacks
`error-handler`	`errorhandler`	The default error handler used by Connect
`session`	`express-session`	Simple session handler that can be extended with *stores* that write sessions to databases or files
`method-override`	`method-override`	Maps new HTTP verbs to the `_method` request variable
`logger`	`morgan`	Log formatting
`response-time`	`response-time`	Track response time
`favicon`	`serve-favicon`	Send favicons, including a built-in default if you don't have one yet

Table 9.3 Migrating Express middleware components *(continued)*

Express 3	Express 4 npm package	Description
`directory`	`serve-index`	Directory listings, similar to Apache's directory indexing
`vhost`	`vhost`	Allow routes to match on subdomains

You might not use most of these modules. In my applications I (Alex) usually have only `body-parser`, `cookie-parser`, `csurf`, `express-session`, and `method-override`, so migration isn't too difficult. The following listing shows a small application that uses these middleware components.

Listing 9.27 Express 4 middleware

```
var bodyParser = require('body-parser');
var cookieParser = require('cookie-parser');
var csurf = require('csurf');
var session = require('express-session');
var methodOverride = require('method-override');
var express = require('express');
var app = express();

app.use(cookieParser('secret'));
app.use(session({ secret: 'secret' }));
app.use(bodyParser());
app.use(methodOverride());
app.use(csurf());

app.get('/', function(req, res) {
  res.send('Hello');
});

app.listen(3000);
```

1 Load middleware modules

2 Configure each piece of middleware

3 Define a route

To install Express 4 and the necessary middleware, you should run the following command in a new directory:

```
npm install --save body-parser cookie-parser \
            csurf express-session method-override \
            serve-favicon express
```

This will install all of the required middleware modules along with Express 4, and save them to a package.json file. Once you've loaded the middleware components with `require` 1, you can add them to your application's stack with `app.use` as you did in Express 3 2. Route handlers can be added exactly as they were in Express 3 3.

OFFICIAL MIGRATION GUIDE The Express authors have written a migration guide that's available in the Express wiki on GitHub.[3] This includes a quick rundown of every change.

[3] https://github.com/visionmedia/express/wiki/Migrating-from-3.x-to-4.x

You can't use `app.configure` anymore, but it should be easy to stop using it. If you're using `app.configure` to do only certain things for specific environments, then just use a conditional statement with `process.env.NODE_ENV`. The following example assumes a fictitious middleware component called `logger` that can be set to be noisy, which might not be desirable when the tests are running:

```
if (process.env.NODE_ENV !== 'test') {
  app.use(logger({ verbose: true }));
}
```

The new routing API reinforces the concept of mini-applications that can be mounted on different endpoints. That means your RESTful resources can leave off the resource name from URLs. Instead of writing `app.get('/songs', songs.index)`, you can now write `songs.get('/', index)` and mount `songs` on `/songs` with `app.use`. This fits in well with the route separation pattern in technique 67.

The next listing shows how to use the new router API.

Listing 9.28 Express 4 middleware

```
var express = require('express');
var app = express();

app.get('/', function(req, res) {
  res.send('Hello');
});

var songs = express.Router();

songs.get('/', function(req, res) {
  res.send('A list of songs');
});

songs.get('/:id', function(req, res) {
  res.send('A specific song');
});

app.use('/songs', songs);

app.listen(3000);
```

1 Create new router

2 Add route handler to this set of routes

3 Mount router with a URL prefix

After creating a new router 1, you can add routes the same way you always did, using HTTP verbs like `get` 2. The cool thing about this is you can also add middleware that will be confined to these routes only: just call `songs.use`. That was previously trickier in older versions of Express.

Once you've set up a router, you can mount it using a URL prefix 3. That means you could do things like mount the same route handler on different URLs to easily alias them.

If you put the routers in their own files and mount them in your main app.js file, then you could even distribute routers as modules on npm. That means you could compose applications from reusable routers.

The final thing we'll mention about Express 4 is the new `router.param` method. This allows you to run asynchronous code when certain route parameters are present. Let's say you have `'/songs/:song_id'`, and `:song_id` should only ever be a valid song that's in the database. With `route.param` you can validate that the value is a number *and* exists in the database, before any route handlers run!

```
router.param('song_id', function(req, res, next, id) {
  Song.find(id, function(err, song) {
    if (err) {
      return next(err);
    } else if (!song) {
      return next(new Error('Song not found'));
    }
    req.song = song;
    next();
  });
});

router.get('/songs/:song_id', function(req, res, next) {
  res.send(req.song);
});
```

In this example, `Song` is assumed to be a class that fetches songs from a database. The actual route handler is now extremely simple, because it only runs if a valid song has been found. Otherwise, `next` will shortcut execution and pass an error to the error-handling middleware.

That wraps up our section on web application development techniques. There's one more important thing before we move on to the next chapter. Like everything else, web applications should be well tested. The next section has some techniques that we've found useful when testing web applications.

9.3 Testing web applications

Testing can feel like a chore, but it can also be an indispensable tool for verifying ideas, particularly if you're creating web APIs without user interfaces.

Chapter 10 has an introduction to testing in Node, and technique 84 has an example for testing web applications. In the next technique we extend this example to show you how to test authenticated routes.

TECHNIQUE 76 Testing authenticated routes

Test frameworks like Mocha make tests easy to read and write, and SuperTest helps keep HTTP-related tests clean. But authentication support isn't usually built into such modules. In this technique you'll learn one way to handle authentication in tests, and the approach is general enough that it can be reused with other test modules as well.

Problem

You want to test parts of your application that are behind a session-based username and password.

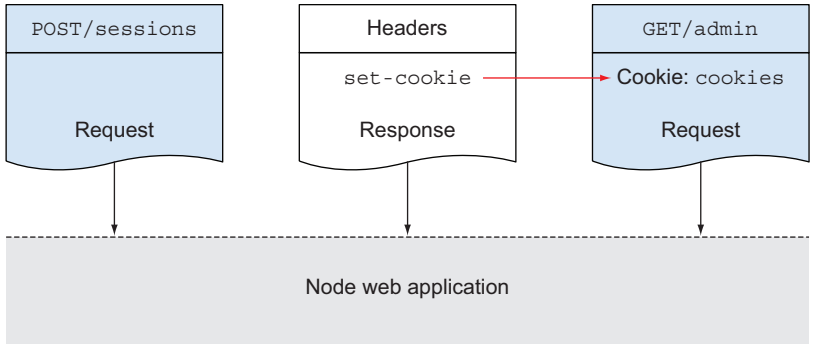


Figure 9.8 You can test authenticated routes by catching cookies.

Solution

Make a request that signs in during the setup phase of the tests, and then reuse the cookies for subsequent tests.

Discussion

Some web frameworks and testing libraries handle sessions for you, so you can test routes without worrying too much about logging in. This isn't true for Mocha and SuperTest, which we've used before in this book, so you'll need to know a bit about how sessions work.

The session handling that Express uses from Connect is based around a cookie. Once the cookie has been set, it can be used to load the user's session. That means that to write a test that accesses a secure part of your application, you'll need to make a request that signs in the user, grabs the cookies, and then use the cookies for subsequent requests. This process is shown in figure 9.8.

To write tests that access authenticated routes, you'll need a test user account, which usually involves creating database fixtures. You'll read about fixtures in chapter 10, technique 87.

Once the data is ready, you can use a library like SuperTest to make a `POST` to your session-handling endpoint with a username and password. Cookies are transmitted using HTTP headers, so you can read them from `res.headers['set-cookie']`. You should also make an assertion to ensure the account was signed in.

Now any new requests just need to set the `Cookie` header with the value from `res.headers`, and your test user will be signed in. The next listing shows how this works.

Listing 9.29 Testing authenticated requests

```
var app = require('./../app');
var assert = require('assert');
var request = require('supertest');
var administrator = {
  username: 'admin',
  password: 'secret'
};
```

(1) This is a test user that would usually be loaded from a fixture.

```
describe('authentication', function() {
  var cookies;

  before(function(done) {
    request(app)
      .post('/session')
      .field('username', administrator.username)
      .field('password', administrator.password)
      .end(function(err, res) {
        assert.equal(200, res.statusCode);
        cookies = res.headers['set-cookie'];
        done();
      });
  });

  it('should allow admins to access the admin area', function(done) {
    request(app)
      .get('/admin')
      .set('Cookie', cookies)
      .expect(200, done);
  });
});
```

The session cookie is in the set-cookie header. (3)

(2) **Post the username and password.**

This route is behind a login. (4)

(5) **Set the Cookie header with the saved session cookie.**

The first part of this test loads the required modules and sets up an example user (1). This would usually be stored in a database, or set by a fixture. Next, a `POST` is made with the username and password (2). The session cookie will be available in the `set-cookie` header (3).

To access a route that's behind a login (4), set the `Cookie` header with the previously saved cookies (5). You should find that the request is handled as if the user had signed in normally.

The trick to understanding testing with sessions can be learned by looking at how Connect's session middleware component works. Other middleware isn't as easy to manage during testing, so the next technique introduces the concept of test *seams*, which will allow you to bring middleware under control during testing.

TECHNIQUE 77 Creating seams for middleware injection

Middleware is flexible and composable. This modular approach makes Connect-based applications a joy to work on. But there's a downside to middleware: testability. Some middleware makes routes inherently difficult to test. This technique looks at ways to get around this by creating *seams*.

Problem

You're using middleware that has made your application difficult to test.

Solution

Find seams where middleware can be replaced for the duration of the tests.

Discussion

The term *seam* is a formal way of describing places in code that can be changed without editing the original code. The concept is extended to apply to languages like

JavaScript by Stephen Vance in his book *Quality Code: Software Testing Principles, Practices, and Patterns.*[4]

> *A seam in our code gives us the opportunity to take control of that code and exercise it in a testing context. Any place that we can execute, override, inject, or control the code could be a seam.*

One example of this is the `csrf` middleware component from Connect. It creates a session variable that can be included in forms to avoid cross-site request forgery attacks. Let's say you have a web application that allows registered users to create calendar entries. If your site didn't use CSRF protection, someone could create a web page that tricks a user of your site into deleting items from their calendar. The attack might look like this:

```
<img src="http://calendar.example.com/entry/1?_method=delete">
```

The user's browser will dutifully load the image source that's hosted on an external site. But it references your site in a potentially dangerous way. To prevent this, a random token is generated on each request and inserted into forms. The attacker doesn't have access to the token, so the attack is mitigated.

Unfortunately, simply adding `express.csrf` to routes that render forms isn't entirely testable. Tests can no longer post to route handlers without first loading the form and scraping out the session variable that contains the secret CSRF token.

To get around this, you need to take `express.csrf` under your control. Refactor it to create a seam: place it in a module that contains your other custom middleware, and then change it during tests. You don't need to test `express.csrf` because the authors of Express and Connect have done that for you—instead, change its behavior during tests.

Two other options are available: checking if `process.env.NODE_ENV` is set to `test` and then branching to a test-only version of the CSRF middleware component, or patching `express.csrf`'s internals so you can extract the secret token. There are problems with both of these approaches: the first means you can't get 100% code coverage—your production code has to include test code. The second approach is potentially brittle: it's too sensitive to Connect changing the way CSRF works in the future.

The seam-based concept that we'll use requires that you create a middleware file if you don't already have one. This is just a file that groups all of your middleware together into a module that can be easily loaded. Then you need to create a function that wraps around `express.csrf`, or just returns it. A basic example follows.

Listing 9.30 Taking control of middleware

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

module.exports.csrf = express.csrf;
```

1 Create a place where other middleware can be injected.

[4] https://www.informit.com/store/quality-code-software-testing-principles-practices-9780321832986

All this does is export the original `csrf` middleware component ❶, but now it's much easier to inject different behavior during tests. The next listing shows what such a test might look like.

Listing 9.31 Injecting new behavior during tests

```
var middleware = require('./../middleware');

middleware.csrf = function() {
  return function(req, res, next) {
    req.session._csrf = '';
    next();
  };
};

var app = require('./../app');
var request = require('supertest');

describe('calendar', function() {
  it('should allow us to turn off csrf', function(done) {
    request(app)
      .post('/calendars')
      .expect(200, done);
  });
});
```

❶ Load middleware component first, and replace csrf method.

❷ This stops views from breaking if they expect _csrf to be set.

❸ A 200 should be returned, not a 403!

This test loads our custom middleware module before anything else, and then replaces the `csrf` method ❶. When it loads `app` and fires off a request using SuperTest, Express will use our injected middleware component because `middleware.js` will be cached. The `_csrf` value is set just in case any views expected it ❷, and the request should return a 200 instead of a 403 (forbidden) ❸.

It might not seem like we've done much, but by refactoring how `express.csrf` is loaded, we've been able to run our application in a more testable way. You may prefer to make two requests to ensure the `csrf` middleware component is used normally, but this technique can be used for other things as well. You can bring any middleware under control for testing. If there's something you don't want to run during tests, look for seams that allow you to inject the desired behavior, or try to create a seam using simple JavaScript or Node patterns—you don't need a complex dependency injection framework; you can take advantage of Node's module system.

The next technique builds on some of these ideas to allow tests to interact with simulated versions of remote services. This will make it easier if you're writing tests for an application that accesses remote services, like a payment gateway.

TECHNIQUE 78 Testing applications that depend on remote services

Third-party modules can help you integrate your applications with remote services like GitHub, Twitter, and Facebook. But how do you test applications that depend on such remote services? This technique looks at ways to insert stubs for remote dependencies, to make your tests faster and more maintainable.

■ **Problem**

You're using a social network for authentication, or a service to accept payments, and you don't want your tests to access these remote dependencies.

■ **Solution**

Find the seams between your application, the remote service, and the things you want to test, and then insert your own HTTP servers to simulate parts of the remote dependency.

■ **Discussion**

One of the things that most web applications need, yet is easy to get dangerously wrong, is user accounts. Using a Node module that supports the authorization services provided by companies like GitHub, Google, Facebook, and Twitter is both quick and potentially safer than creating a bespoke solution.

It's comparatively easy to adopt one of these services, but how do you test it? In technique 76, you saw how to write tests for authenticated routes. This involved signing in and saving the session cookies so subsequent requests appeared authenticated. You can't use the same approach with remote services, because your tests would have to make requests to real-life production services. You could use a test account, but what if you wanted to run your tests offline?

To get around this, you need to create a seam between your application and the remote service. Whenever your application attempts to communicate with the remote service, you need to slot in a fake version that emits similar responses. In unit tests, mock objects simulate other objects. What you want is to mock a service.

There are two requirements that your application needs to satisfy to make this possible:

- Configurable remote services
- A web server that can stand in for the remote service

The first condition means your application should allow the URLs of remote services to be changed. If it needs to connect to http://auth.example.com/signin, then you'll need to specify http://localhost:3001/signin during testing. The port is entirely up to you—some solutions we've seen use a sequence of port numbers so multiple services can be run at once for the same tests.

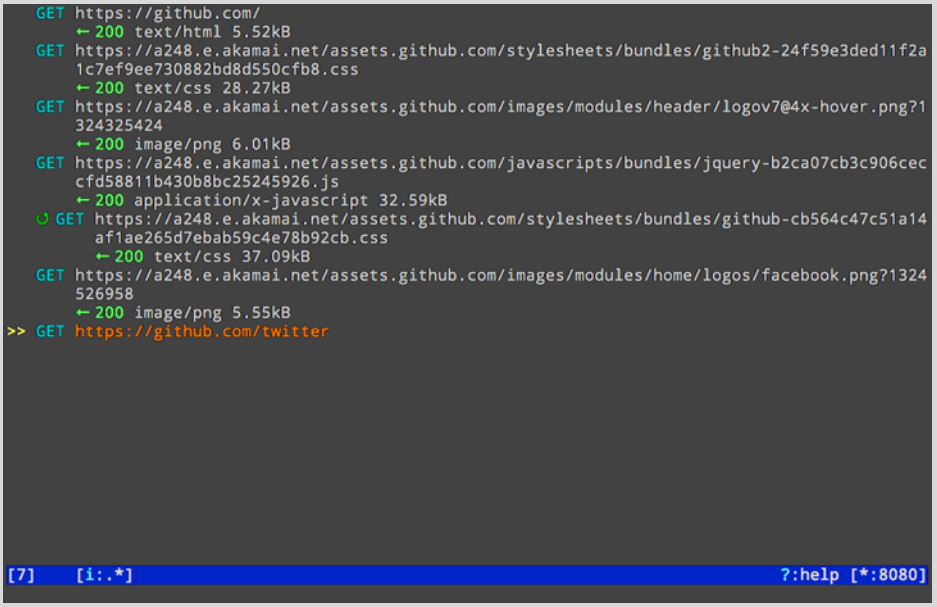
The second condition can be handled however you want. If you're using Express, you could start an Express server with a limited set of routes defined—just enough routes and code to simulate the remote service. This server can be kept in its own module, and loaded in the tests that need it.

In practice this doesn't require much code, so once you understand the principle it shouldn't be too difficult to reuse it to handle practically any API. If the API you're attempting to simulate isn't well documented, then you may need to capture real requests to figure out how it works.

Investigating remote APIs

There are times when remote APIs aren't well documented. Once you get beyond the basic API calls, there are bound to be parts that aren't easy to understand. In cases like this, we find it's best to make requests with a command-line tool like `curl`, and watch the requests and responses in an HTTP logging tool.

If you're using Windows, then Fiddler (http://www.telerik.com/fiddler) is absolutely essential. It's described as a HTTP debugging proxy, and it supports HTTPS as well.



Glance has built-in pages for errors.

For Linux and Mac OS, mitmproxy (http://mitmproxy.org/) is a powerful choice. It allows HTTP traffic to be observed in real time, dumped, saved, and replayed. We've found it perfect for debugging our own Node-powered APIs that support desktop apps, as well as figuring out the quirks of certain popular payment gateways.

In the following three listings, you'll see how to create a *mock server* that a test can use to simulate some of PayPal's behavior. The first listing shows the application itself.

Listing 9.32 A small web store that uses PayPal

```
var express = require('express');
var app = express();
var PayPal = require('./paypal');
var payPal = new PayPal({
  user: 'NIP',
  payPalUrl: 'http://localhost:3001/validate',
```

1 **These settings control PayPal's behavior.**

```
  rootUrl: 'http://localhost:3000'
});

app.use(express.bodyParser());

app.post('/buy', function(req, res, next) {
  var url = payPal.generateUrl(req.body);

  // Send the user to the PayPal payment page
  res.redirect(url);
});

app.post('/paypal/success', function(req, res, next) {
  payPal.verify(req.body, function(err) {
    if (err) next(err);
    app.emit('purchase:accepted', req.body);
    res.send(200);
  });
});

module.exports = app;
```

2 Get the purchase URL used by PayPal.

3 Handle payment notifications.

4 When a payment is successful, emit an event.

The settings passed to the `PayPal` class near the top of the file 1 are used to control PayPal's behavior. One of them, `payPalUrl`, could be https://www.sandbox.paypal.com/cgi-bin/webscr for testing against PayPal's staging server. Here we use a local URL, because we're going to run our own mock server.

If this were a real project, you should use a configuration file to store these options. One for each environment would make sense. Then the test configuration could point to a local server, staging could use PayPal sandbox, and live would use PayPal.com. For more on configuration files, see technique 69.

To make a payment, the user is forwarded to PayPal's hosted forms. Our demonstration `PayPal` class has the ability to generate this URL, and it'll use `payPalUrl` 2. This example also features payment notification handling 3—known as IPN in PayPal's nomenclature.

An extra feature we've added here is the call to `emit` 4. This makes it easier to test, because our tests can now listen for `purchase:accepted` events. It's also useful for setting up email handling—see technique 73 for more on that.

Now for the mock PayPal server. All it needs to do is handle IPN requests. It basically needs to say, "Yes, that purchase has been validated." It could also optionally report errors so we can test error handling on our side as well. The next listing shows what the tiny mocked server looks like.

Listing 9.33 Mocking PayPal's IPN requests

```
var express = require('express');
var paypalApp = express();

paypalApp.returnInvalid = false;

paypalApp.post('/validate', function(req, res) {
```

1 Allow errors to be toggled

2 Handle IPN validation

```
  if (paypalApp.returnInvalid) {
    res.send('INVALID');
  } else {
    res.send('VERIFIED');
  }
});

module.exports = paypalApp;
```

Real-life PayPal stores receive a `POST` from PayPal with an order's details, near the end of the sales process. You need to take that order and send it back to PayPal for verification. This prevents attackers from crafting a `POST` request that tricks your application into thinking a fake purchase was made.

This example includes a toggle so errors can be turned on (1). We're not going to use it here, but it's useful in real projects because you'll want to test how errors are handled. There will be customers that encounter errors, so ensuring they're handled gracefully is critical.

Once all that's in place, all we need to do is send back the text `VERIFIED` (2). That's all PayPal does—it can be frustratingly abstruse at times!

Finally, let's look at a test that puts all of this together. The next listing uses both the mocked PayPal server and our application to make purchases.

Listing 9.34 Testing PayPal

```
var app = require('./../app');
var assert = require('assert');
var request = require('supertest');
var payPalMock = require('./paypalmock');

function makeCustomer() {
  return {
    address1: '123',
    city: 'Nottingham',
    country: 'GB',
    email: 'user@example.com',
    first_name: 'Paul',
    last_name: 'Smith',
    state: 'Nottinghamshire',
    zip: 'NG10932',
    tax_number: ''
  };
}

function makeOrder() {
  return {
    id: 1,
    customer: makeCustomer()
  };
}

function makePayPalIpn(order) {
  // More fields should be used for the real PayPal system
```

(1) Customer fixture

(2) Order fixture

(3) What PayPal would send

```
  return {
    'payment_status': 'Completed',
    'receiver_email': order.customer.email,
    'invoice': order.id
  };
}

describe('buying the book', function() {
  var payPalServer;

  before(function(done) {
    payPalServer = payPalMock.listen(3001, done);
  });

  after(function(done) {
    payPalServer.close(done);
  });

  it('should redirect the user to paypal', function(done) {
    var order = makeOrder();

    request(app)
      .post('/buy')
      .send(order)
      .expect(302, done);
  });

  it('should handle IPN requests from PayPal', function(done) {
    var order = makeOrder();

    app.once('purchase:accepted', function(details) {
      assert.equal(details.receiver_email, order.customer.email);
    });

    request(app)
      .post('/paypal/success')
      .send(makePayPalIpn(order))
      .expect(200, done);
  });
});
```

4 Before each test, start up mock PayPal server

5 After each test, close mock PayPal server

6 User should be redirected for valid orders

7 200 OK should be returned for valid orders

This test sets up a sample order 2, which requires a customer 1. We also create an object that has the same fields as a PayPal IPN request—this is what we're going to send to our mock PayPal server for validation. Before 4 and after 5 each test, we have to start and stop the mock PayPal server. That's because we don't want servers running when they're not needed—it might cause other tests to behave strangely.

When the user fills out the order form on our site, it will be posted to a route that generates a PayPal URL. The PayPal URL will forward the user's browser to PayPal for payment. Listing 9.34 includes a test for this 6, and the URL it generates will start with our local test PayPal URL from listing 9.32.

There's also a test for the notification sent by PayPal 7. This is the one we're focusing on that requires the PayPal mocked server. First we have to POST to our server

at /paypal/success with the notification object ❸—this is what PayPal would normally do—and then our application will make an HTTP request to PayPal, which will hit the mocked server, and then return `VERIFIED`. The test simply ensures a 200 is returned, but it's also able to listen for the `purchase:accepted` event, which indicates a given purchase is complete.

It might seem like a lot of work, but you'll be able to work more efficiently once your remote services are simulated with mock servers. Your tests run faster, and you can work offline. You can also make your mocked services generate all kinds of unusual responses, which will help you get better test coverage if that's one of your goals.

This is the last web-related technique that we cover in this chapter. The next sections discuss emerging trends in Node web development.

9.4 Full stack frameworks

In this chapter you've seen how to build web applications with Node's built-in modules, Connect, and Express. There's an emerging class of new frameworks known as *full stack frameworks*. They provide features that are needed to make rich, browser-based applications with modern tools like data binding, but also handle server-side concerns like modeling business logic and data persistence.

If you're set on using Express, then you can still start working with full stack frameworks today. The *MEAN* solution stack uses MongoDB, Express, AngularJS, and Node. There could be many MEAN implementations out there, but the MEAN Stack from Linnovate (https://github.com/linnovate/mean) is currently the most popular. It comes with Mongoose for data models, Passport for authorization, and Twitter Bootstrap for the user interface. If you're working in a team that's already familiar with Bootstrap, AngularJS, and Mongoose, then this is a great way to get new projects off the ground quickly.

The book *Getting MEAN*[5] introduces full stack development and covers Mongoose models, RESTful API design, and account management with Facebook and Twitter.

Another framework that builds on Express and MongoDB is Derby (http://derbyjs.com/). Instead of Mongoose, Derby uses Racer to implement data models. This allows data from different clients to be synchronized, using operational transformation (OT). OT is specifically designed to support collaborative systems, so Derby is a good choice for developing software inspired by Etherpad (http://etherpad.org/). It also has client-side features like templates and data binding.

If you like Express but want more features, then one option that we haven't covered is Kraken (http://krakenjs.com/) by PayPal. This framework adds more structure to Express projects by adding subdirectories for configuration, controllers, Grunt tasks, and tests. It also supports internationalization out of the box.

Some frameworks are almost entirely focused on the browser, relying on Node only for sensitive operations and data persistence. One popular example is Meteor (https://www.meteor.com/). Like Derby and MEAN Stack, it uses MongoDB, but the creators

[5] *Getting MEAN* by Simon Holmes: http://www.manning.com/sholmes/.

are planning support for other databases. It's based around a pub/sub architecture, where JSON documents are pushed between the client and server. Clients retain an in-memory copy of the documents—servers publish sets of documents, while clients subscribe to them. This means most model-related code in the browser can be written synchronously.

Meteor embraces reactive programming, a paradigm that's currently popular in desktop development circles. This allows *reactive computations* to be bound to methods. If you subscribe a function to such a value, the function will be rerun when the value changes. The overall effect in a real application is streamlined code—there's essentially less pub/sub management and event-handling code.

Hoodie (http://hood.ie/) is a competitor to Meteor. It uses CouchDB, and is suitable for mobile applications because it synchronizes data when possible. Almost everything can happen locally. It comes with built-in account management, which is as simple as `hoodie.account.signUp('alex@example.com', 'pass')`. There's even a global public store, so data can be saved for specific users or made available to everyone using a given application.

There's lots of activity in the Node web framework scene, but there's another aspect to Node web development that we haven't mentioned yet: real-time development.

9.5 Real-time services

Node is the natural choice for web-based real-time services. Broadly speaking, this involves three types of applications: statistics servers, collaboration services, and latency-sensitive applications like game servers.

It's not that difficult to start a server with Express and collect data about your other applications, servers, weather sensor data, or dog-feeding robot. Unfortunately, doing this well isn't trivial. If you're logging something every time someone plays your free-to-play iOS game, what happens when there are thousands of events a minute? How do you scale this, or view critical information in real time?

Some companies have this problem on a huge scale, and fortunately some of them have created open source tools that we can reuse. One example is Cube (http://square.github.io/cube/) by Square. Cube allows you to collect timestamped events and then derive metrics on them. It uses MongoDB, so you could feed data out to something that generates graphs. Square has a solution for visualizing the data called Cubism.js (http://square.github.io/cubism/), which renders new values in real-time (see figure 9.9).

The Etherpad project (http://etherpad.org/) is a Node-powered collaborative document editor. It allows users to chat as they make changes to documents, and color-codes the changes so it's easy to see what each person is doing. It's based on some of the modules you've seen in this book: Mikeal Rogers' `request`, Express, and Socket.IO.

WebSockets make these projects possible. Without WebSockets, pushing data to the client would be more cumbersome. Node has a rich set of WebSockets

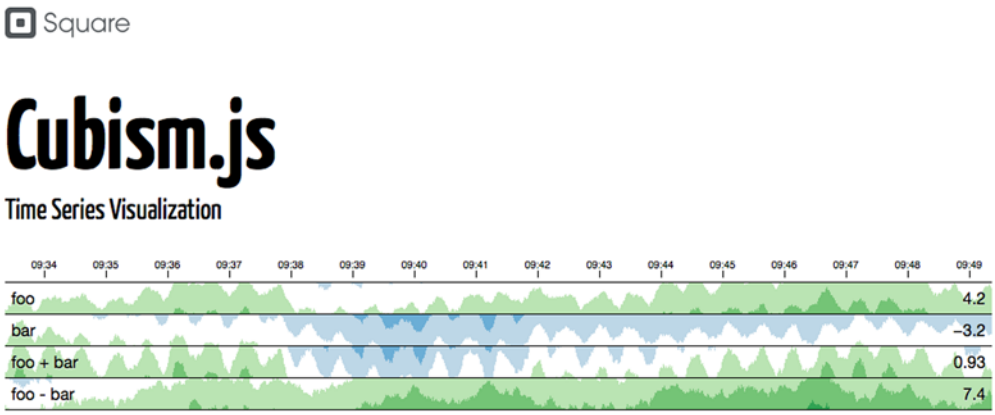


Figure 9.9 Cubism.js shows time series values in real time.

implementations—Socket.IO (http://socket.io/) is the most popular, but there's also `ws` (https://www.npmjs.org/package/ws), which claims to be the fastest WebSocket implementation.

There's a parallel between sockets and streams; SocketStream (http://socketstream.org/) aims to bridge the gap by building web applications entirely around streams. It uses the HTML5 `history.pushState` API with single-page applications, Connect middleware, and code sharing with the browser.

9.6 Summary

In this chapter you've seen how Node fits in with modern web development. It can be used to improve client-side tooling—it's now normal for client-side developers to install Node and a Node build tool.

Node is also used for server-side development. Express is the major web framework, but many projects can get off the ground with a subset from Connect. Other frameworks are similar to Express, but have a different focus. Restify is one example, and can be used to make strict RESTful APIs (technique 71).

Writing well-structured Express applications means you should adopt certain patterns and idioms that the Node community has adopted. This includes error handling (technique 70), folders as modules and route separation (technique 67), and decoupling through events (technique 73).

It's also increasingly common to use Node modules in the browser (technique 66), and client-side code in Node (technique 65).

If you want to write better code, you should adopt test-driven development as soon as possible. We've included some techniques that enable you to test things like authentication (technique 76) and mocking remote APIs (technique 78), but the simple act of writing a test to think about new code is one of the best ways to improve your Node web applications. One way you can do this is every time you want to add a new route to a web application, write the test first. Practice using Super-

Test, or a comparable HTTP request library, and use it to plan out new API methods, web pages, and forms.

The next chapter shows you how to write better tests, whether they're simple scripts or database-driven web applications.

10 Tests: The key to confident code

This chapter covers

- Assertions, custom assertions, and automated testing
- Ensuring things fail as expected
- Mocha and TAP
- Testing web applications
- Continuous integration
- Database fixtures

Imagine that you wanted to add a new currency to an online shop. First you'd add a test to define the expected calculations: subtotal, tax, and the total. Then you'd write code to make this test pass. This chapter will help you learn how to write tests by looking at Node's built-in features for testing: the `assert` module and test scripts that you can set in your package.json file. We also introduce two major test frameworks: Mocha and `node-tap`.

Introduction to testing

This chapter assumes you have some experience at writing unit tests. Table 10.1 includes definitions of the terminology used; if you want to know what we mean by assertions, test cases, or test harnesses, you can refer to this table.

For a more detailed introduction to testing, *The Art of Unit Testing, Second Edition* (Roy Osherove, Manning, 2013; http://manning.com/osherove2/) has step-by-step examples for writing maintainable and readable tests. *Test Driven Development: By Example* (Kent Beck, Addison-Wesley, 2002; http://mng.bz/UT12) is another well-known foundational book on the topic.

One of the advantages of working with Node is that the community adopted testing early on, so there's no shortage of modules to help you write fast and readable tests. You might be wondering what's so great about tests and why we write them early on during development. Well, tests are important for exploring ideas before committing to them—you can think of them like small, flexible experiments. They also communicate your *intent*, which means they help document and expand on the ideas in the key parts of the project. Tests can also help reduce maintenance in mature projects by allowing you to check that changes haven't broken existing working features.

The first thing to learn about is Node's `assert` module. This module allows you to define an expectation that will throw an error when it isn't met. Expressing and confirming expectations is the main purpose of tests, so you'll see a lot of assertions in this chapter. Although you don't have to use `assert` to write tests, it's a built-in core module and similar to assertion libraries you might've used before in other languages. The first set of techniques in this chapter is all about assertions.

To get everyone up to speed, the next section includes a list of common terms used when working with tests.

10.1 Introduction to testing with Node

To make it easier for newcomers to automated testing, we've included table 10.1 that defines common terminology. This table also outlines what we mean by specific terms, because some programming communities use the same terms slightly differently.

Table 10.1 Node testing concepts

Term	Description
Assertion	A logical statement that allows you to test expressions. Supported by the `assert` core module; for example: `assert.equal(user.email, 'name@example.com');.`
Test case	One or more assertions that test a particular concept. In Mocha, a test case looks like this: `it('should calculate the square of a number', function() { assert.equal(square(4), 16); });`

Table 10.1 Node testing concepts *(continued)*

Term	Description

Test harness	A program that runs tests and collates output. The resulting reports help diagnose problems when tests fail.
	This builds on the previous example, so with Mocha a test harness looks like this:
	`var assert = require('assert');`
	`var square = require('./square');`
	`describe('Squaring numbers', function() {`
	`  it('should calculate the square of a number', function() {`
	`  assert.equal(square(4), 16);`
	`  });`
	`  it('should return 0 for 0', function() {`
	`  assert.equal(square(0), 0);`
	`  });`
	`});`

Fixture	Test data that is usually prepared before tests are run. Let's say you want to test a user accounts system. You could predefine users and their passwords, and then include the passwords in the tests to ensure users can sign in correctly. In Node, JSON is a popular file format for fixtures, but you could use a database, SQL dump, or CSV file. It depends on your application's requirements.
Mock	An object that simulates another object. Mocks are often used to replace I/O operations that are either slow or difficult to run in unit tests; for example, downloading data from a remote web API, or accessing a database.
Stub	A method stub is used to replace functionality for the duration of tests. For example, methods used to communicate with an I/O source like a disk or remote API can be stubbed to return predefined data.
Continuous integration server	A CI server runs automated tests whenever a project is updated through a version control server.

The only feature from table 10.1 that Node directly supports is assertions. The other features are provided through third-party libraries—you'll learn about CI servers in technique 86, and mocks and fixtures in technique 87. You don't have to use all of these things to write tests, you can actually write tests with just the assertion module. The next section introduces the `assert` module so you can start writing basic tests.

10.2 Writing simple tests with assertions

So far we've briefly mentioned that assertions are used to test expressions. But what does this involve? Typically assertions are functions that cause an exception to be raised if a condition isn't met. A failing assertion is like your credit card being declined in a store—your program will refuse to run no matter how many times you try. The idea of assertions has been around for a long time; even C has assertions.

In C, the standard library includes the `assert()` macro, which is used for verifying expressions. In Node, we have the `assert` core module. There are other assertion modules out there, but `assert` is built-in and easy to use and extend.

> **COMMONJS UNIT TESTING** The `assert` module is based on the CommonJS Unit Testing 1.1 specification (http://wiki.commonjs.org/wiki/Unit_Testing/1.1). So even though it's a built-in core module, you can use other assertion modules as well. The underlying principles are always the same.

This section introduces Node's built-in assertions. By following the first technique, you'll be able to write tests using the `assert` core module by using `assert.equal` to check for equality, and to automate the running of tests by using npm scripts.[1]

TECHNIQUE 79 Writing tests with built-in modules

Have you ever tried to write a quick test for an important feature, but you found yourself lost in test library documentation? It can be hard to get started actually writing tests; it seems like there's a lot to learn. If you just start using the `assert` module, though, you can write tests right now without any special libraries.

This is great when you're writing a small module and don't want to install any dependencies. This technique demonstrates how to write clean, expressive, single-file tests.

Problem

You have a clear idea of the acceptable input and output values for your module, class, or functions, and you want it to be clear when the output values don't match the input.

Solution

Use the `assert` module and npm scripts.

Discussion

Node comes with an assertion module. You can think of this as a toolkit for checking expectations against outcomes. Internally this is done by comparing *actual* values against *expected* values. The `assert.equal` method demonstrates this perfectly: the arguments are `actual, expected`. There's also a third optional argument: `message`. Passing a message makes it easier to understand what happened when tests fail.

Let's say you're writing an online shop that calculates order prices, and you've sold three items at $3.99 each. You could ensure the correct price gets calculated with this:

```
assert.equal(
  order.subtotal, 11.97,
  'The price of three items at $3.99 each'
);
```

In methods with only a single required argument, like `assert(value)`, the expected value is `true`, so it uses the same pattern.

---

[1] This is defined by the `scripts` property in a package.json file. See `npm help scripts` for details on this feature.

To see what happens when a test fails, try running the next listing.

Listing 10.1 The `assert` module

```
var assert = require('assert');
var actual = square(2);
var expected = 4;

assert(actual, 'square() should have returned a value');
assert.equal(
  actual,
  expected,
 'square() did not calculate the correct value'
);

function square(number) {
  return number * number + 1;
}
```

(1) Load assertion module

assert module is a function for testing truth (2)

assert.equal allows expectations to be set up for shallow equality checks (3)

(4) square() function is what we're testing

The first line you'll see in most test files is one that loads the `assert` module (1). The `assert` variable is also a function aliased from `assert.ok`—which means you can use either `assert()` or `assert.ok()` (2).

It's easy to forget the order of the arguments for `assert.equal`, so you might find yourself checking Node's documentation a lot. It doesn't really matter how you order the arguments—some people might find it easier to list the expected value first so they can scan the code for values—but you should be consistent. That's why this example is explicit about the naming of `actual` and `expected` (3).

This test has a function that has an intentional bug (4). You can run the test with `node assertions.js`, which should display an error with a stack trace:

```
assert.js:92
  throw new assert.AssertionError({
        ^
AssertionError: square() did not calculate the correct value
    at Object.anonymous (listings/testing/assertions.js:7:8)
     at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:901:3
```

(1) File and line number where the assertion failed

These stack traces can be hard to read. But because we've included a message with the assertion that failed, we can see a description of what went wrong. We can also see that the assertion failed in the file assertions.js on line 7 (1).

The `assert` module has lots of other useful methods for testing values. The most significant is `assert.deepEqual`, which can check for equality between two objects. This is important because `assert.equal` can only compare shallow equality. Shallow equality is used for comparing primitive values like strings or numbers, whereas `deepEqual` can compare objects with nested objects and values.

You might find `deepEqual` useful when you're writing tests that return complex objects. Think about the online shop example from earlier. Your shopping cart might look like this: `{ items: [ { name: "Coffee beans", price: 4.95 } ], subtotal: 4.95 }`. It's an object that contains an array of shopping cart items, and a subtotal that is calculated by another object. Now, to check this entire object against one that you've defined in your unit test, you'd use `assert.deepEqual`, because it's able to compare objects rather than just primitive values.

The `deepEqual` method can be seen in the next listing.

Listing 10.2 Testing object equality

```
var assert = require('assert');
var actual = login('Alex');
var expected = new User('Alex');

assert.deepEqual(actual, expected, 'The user state was not correct');

function User(name) {
  this.name = name;
  this.permissions = {
    admin: false
  };
}

function login(name) {
  var user = new User(name);
  user.permissions.admin = true;
  return user;
}
```

1 Load the assert module.

Use deepEqual to compare objects. 2

3 The login system has a bug!

This example uses the `assert` module 1 to test objects created by a constructor function, and an imaginary login system. The login system is accidentally loading normal users as if they were administrators 3.

The `assert.deepEqual` method 2 will go over each property in the objects to see if any are different. When it runs into `user.permissions.admin` and finds the values differ, an `AssertionError` exception will be raised.

If you take a look at the `assert` module's documentation, you'll see many other useful methods. You can invert logic with `notDeepEqual` and `notEqual`, and even perform strict equality checks just like `===` with `strictEqual` and `notStrictEqual`.

There's another aspect to testing, and that's ensuring that things fail the way we expect. The next technique looks at testing for failures.

TECHNIQUE 80 Testing for errors

Programs will eventually fail, but when they do, we want them to produce useful errors. This technique is about ensuring that expected errors are raised, and about how to cause exceptions to be raised during testing.

Problem

You want to test your error-handling code.

Solution

Use `assert.throws` and `assert.ifError`.

Discussion

One of the conventions we use as Node developers is that asynchronous methods should return an error as the first argument. When we design our own modules, we know there are places where errors are likely to occur. Ideally we should test these cases to make sure the correct errors are passed to callbacks.

The following listing shows how to ensure an error hasn't been passed to an asynchronous function.

Listing 10.3 Handling errors from asynchronous APIs

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

function readConfigFile(cb) {
  fs.readFile('config.cfg', function(err, data) {
    if (err && err.code === 'ENOENT') {
      cb(null, { database: 'psql://localhost/test' });
    } else if (err) {
      cb(err);
    } else {
      // Do important configuration stuff
      cb(null, data);
    }
  });
}

// Test to make sure non-existent configuration
// files are handled correctly.
readConfigFile(function(err, data) {
  assert.ifError(err);
});
```

1 The function we want to test takes a callback.

2 If the error is "file not found," return default values.

3 Otherwise, pass the error to the callback.

4 Now ifError will fail if any errors are passed.

Although `assert.ifError` works synchronously, it makes semantic sense to use it for testing asynchronous functions that pass errors to callbacks. Listing 10.3 uses an asynchronous function called `readConfigFile` 1 to read a configuration file. In reality this might be the database configuration for a web application, or something similar. If the file isn't found, then it returns default values 2. Any other error—and this is the important part—will be passed to the callback 3.

That means the `assert.ifError` test 4 can easily detect whether an unexpected error has occurred. If something changes in the structure of the project that causes an unusual error to be raised, then this test will catch that and warn the developers before they release potentially dangerous code.

Now let's look at raising exceptions during testing. Rather than using `try` and `catch` in our tests, we can use `assert.throws`.

To use `assert.throws`, you must supply the function to be run and an expected error constructor. Because a function is passed, this works well with asynchronous APIs, so you can use it to test things that depend on I/O operations.

The next listing shows how to use `assert.throws` with a fictitious user account system.

Listing 10.4 Ensuring that exceptions are raised

```
var assert = require('assert');
var util = require('util');

assert.throws(
  function() {
    loginAdmin('Alex');
  },
  PermissionError,
  'A PermissionError was expected'
);

function PermissionError() {
  Error.call(this, arguments);
}
util.inherits(PermissionError, Error);

function User(name) {
  this.name = name;
  this.permissions = {
    admin: false
  };
}

function loginAdmin(name) {
  var user = new User(name);
  if (!user.permissions.admin) {
    throw new PermissionError('You are not an administrator');
  }
  return user;
}
```

1 The first argument of assert.throws is the function being tested.

2 The second argument is the expected error.

3 PermissionError inherits from the standard Error constructor.

4 This is a fake login system that only allows administrators to sign in.

The assertion 1 checks to ensure the expected exception is thrown. The first argument is a function to test, in this case `loginAdmin`, and the second is the expected error 2.

This highlights two things about `assert.throws`: it can be used with asynchronous APIs because you pass it a function, and it expects error objects of some kind. When developing projects with Node, it's a good idea to use `util.inherits` to inherit from the built-in `Error` constructor. This allows people to easily catch your errors, and you can decorate them with extra properties that include useful additional information if required.

In this case we've created `PermissionError` 3, which is a clear name and therefore self-documenting—if someone sees a `PermissionError` in a stack trace, they'll know what went wrong. A `PermissionError` is subsequently thrown in the `loginAdmin` function 4.

This technique delved into error handling with the `assert` module. Combined with the previous techniques, you should have a good understanding of how to test a range of situations with assertions. With `assert.equal` you can quickly compare numbers and

strings, and this covers a lot of problems like checking prices in invoices or email addresses in web application account-handling code. A lot of the time, `assert.ok`—which is aliased as `assert()`—is enough to get by, because it's a quick and handy way for checking for *truthy* expressions. But there's one last thing to master if you want to really take advantage of the `assert` module; read on to learn how to create custom assertions.

TECHNIQUE 81 Creating custom assertions

Node's built-in assertions can be extended to support application-specific expressions. Sometimes you find yourself repeatedly using the same code to test things, and it seems like there might be a better way. For example, suppose you're checking for valid email addresses with a regular expression in `assert.ok`. Writing custom assertions can solve this problem, and is easier than you might think. Learning how to write custom assertions will also help you understand the assertion module from the inside out.

Problem

You're repeating a lot of code in your tests that could be replaced if only you had the right assertion.

Solution

Extend the built-in `assert` module.

Discussion

The `assert` module is built around a single function: `fail`. `assert.ok` actually calls `fail` with the logic inverted, so it looks like this: `if (!value) fail(value)`. If you look at how `fail` works, you'll see that it just throws an `assert.AssertionError`:

```
function fail(actual, expected, message, operator, stackStartFunction) {
  throw new assert.AssertionError({
    message: message,
    actual: actual,
    expected: expected,
    operator: operator,
    stackStartFunction: stackStartFunction
  });
}
```

The error object is decorated with properties that make it easier for test reporters to break down the location and cause of failures. The people who wrote this module knew that others would like to write their own assertions, so the `fail` function is exported, which means it can be reused.

Writing a custom assertion involves the following steps:

1. Define a method with a signature similar to the existing assertion library.
2. Call `fail` when an expectation isn't matched.
3. Test to ensure failure results in an `AssertionError`.

Listing 10.5 puts these steps together to define a custom assertion that ensures a regular expression is matched.

Listing 10.5 A custom assertion

```
var assert = require('assert');                          ← 1 Load assert module
assert.match = match;

function match(actual, regex, message) {
  if (!actual.match(regex)) {                            ← Execute regular expression match against a string using String.prototype.match (2)
    assert.fail(actual, regex, message, 'match', assert.match);
  }
}

assert.match('{ name: "Alex" }', /Alex/, 'The name should be "Alex"');   ← Make sure tests pass (3)

assert.throws(                                           ← Make sure tests fail (4)
  function() {
    assert.match('{ name: "Alex" }', /xlex/, 'This should fail');
  },
  assert.AssertionError,
  'A non-matching regex should throw an AssertionError'
);
```

This example loads the assertion module (1) and then defines a function called `match` that runs `assert.fail` to generate the right exception when the regular expression doesn't match the *actual* value (2). The key detail to remember is to define the argument list to be consistent with other methods in the assertion module—the example here is based on `assert.equal`.

Listing 10.5 also includes some tests. In reality these would be in a separate file, but here they illustrate how the custom assertion works. First we check to see if it passes a simple test by matching a string against a regular expression (3), and then `assert.throws` is used to ensure the test really does fail when it's meant to (4).

Your own domain-specific language

Using custom assertions is but one technique for creating your own testing DSL (domain-specific language). If you find you're duplicating code between test cases, then by all means wrap that code in a function or class.

For example, `setUpUserAccount({ email: 'user@example.com' })` is more readable than three or four lines of setup code, particularly if it's repeated between test cases.

This example might seem simple, but understanding how to write custom assertions improves your knowledge of the underlying module. Custom assertions can help clean up tests where expectations have been made less expressive by squeezing concepts into built-in assertions. If you want to be able to say something like `assert.httpStatusOK`, now you can!

With assertions out of the way, it's time to look at how to organize tests across multiple files. The next technique introduces test harnesses that can be used to organize groups of test files and run them more easily.

10.3 Test harnesses

A test harness, or automated test framework, generally refers to a program that sets up the runtime environment and runs tests, and then collects and compares the results. Since it's automated, tests can be run by other systems including continuous integration (CI) servers, covered in technique 86.

Test harnesses are used to execute groups of test files. That means you can easily run lots of tests with a single command. This not only makes it easier for you to run tests, but makes it easier for your collaborators as well. You may even decide to start all projects with a test harness before doing anything else. The next technique shows you how to make your own test harness, and how to save time by adding scripts to your package.json files.

TECHNIQUE 82 Organizing tests with a test harness

Suppose you're working on a project and it keeps on growing, and using a single test file is starting to feel messy. It's hard to read and causes confusion that leads to mistakes. So you'd like to use separate files that are related in some way. Perhaps you'd even like to run tests one file at a time to help track down issues when things go wrong.

Test harnesses solve this problem.

Problem

You want to write tests organized into test cases and test suites.

Solution

Use a test harness.

Discussion

First, let's consider what a test harness is. In Node, a test harness is a command-line script that you can run by typing the name of the script. At its most basic, it must run a group of test files and display any errors that occur. We don't need anything particularly special to do that—a failed assertion will cause an exception to be thrown; otherwise the program will exit silently with a return code of `0`.

That means a basic test harness is just `node test/*.js`, where test/ is a directory that contains a set of test files. We can go one better than that. All Node projects should have a package.json file. One of the properties in this file is `scripts`, and one of the default scripts is `test`. Any string you set here will be executed like a shell command.

The following listing shows an example package.json file with a test script.

Listing 10.6 A package.json with a test script

```
{
  "name": "testrunner",
  "version": "0.0.0",
  "description": "A test runner",
  "main": "test-runner.js",
  "dependencies": {},
  "devDependencies": {},
  "scripts": {
    "test": "node test-runner.js test.js test2.js"
  }
}
```

1 Test script invocation goes here

```
  },
  "author": "",
  "license": "MIT"
}
```

With `node test-runner.js test.js test2.js` set as the `test` script ❶, other developers can now run your tests simply by typing `npm test`. This is much easier than having to remember a project-specific command.

Let's expand this example by looking at how test harnesses work. A test harness is a Node program that runs groups of test files. Therefore, we should be able to give such a program a list of files to test. Whenever a test fails, it should display a stack trace so we can easily track down the source of the failure.

In addition, it should exit with a non-zero return code whenever a test fails. That allows tests to be run in an automated way—other software can easily see if a test failed without having to parse the textual output from the tests. This is how continuous integration (CI) servers work: they automatically run tests whenever code is committed to a version control system like Git.

The next listing shows what a test file for this system should look like.

Listing 10.7 An example test file

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

it('should run a test', function() {
  assert('a' === 'a');
});

it('should allow a test to fail', function() {
  assert(true);
  assert.equal('a', 'b', 'Bad test');
});

it('should run a test after the failed test', function() {
  assert(true);
});
```

❶ The it() function represents a test case.

❷ A failing test is included so we can see what the results look like.

❸ This last test should still run.

The `it` function ❶ looks strange, but it's a global function that will be provided by our test framework. It gives each test case a name so it's easier to understand the results when the tests are run. A failing test is included ❷ so we can see what happens when tests fail. The last test case ❸ should run even though the second one failed.

Now, the final piece of the puzzle: the next listing includes a program capable of executing this test.

Listing 10.8 Running tests in a prescribed manner

```
var assert = require('assert');
var exitCode = 0;
var filenames = process.argv.slice(2);

it = function(name, test) {
  var err;

  try {
    test();
  } catch (e) {
    err = e;
  }

  console.log(' - it', name, err ? '[FAIL]' : '[OK]');

  if (err) {
    console.error(err);
    console.error(err.stack);
    exitCode = 1;
  }
};

filenames.forEach(function(filename) {
  console.log(filename);
  require('./' + filename);
});

process.on('exit', function() {
  process.exit(exitCode);
});
```

1. The it() function is defined as a global.
2. Tests are passed as callbacks and run inside a try/catch statement.
3. Results are printed based on the presence of an exception.
4. A stack trace is printed to help track down errors.
5. Each file passed on the command-line is run.
6. When the program exits, return a non-zero error code if a test failed.

This example can be run by passing test files as arguments: `node test-runner.js test.js test2.js test-n.js`. The `it` function is defined as a global 1, and is called `it` so the tests and their output read logically. This makes sense when the results are printed 3.

Because `it` takes a test case name and a callback, the callback can be run under whatever conditions we desire. In this case we're running it inside a `try/catch` statement 2, which means we can catch failed assertions and report errors 4 to the user.

Tests are loaded by calling `require` on each of the files passed in as command-line arguments 5. In a more polished version of this program, the file handling would need to be more sophisticated. Wildcard expressions would need to be supported, for example.

A failed test case causes the `exitCode` variable to be set to a non-zero value. This is returned to the controlling process with `process.exit` in the exit handler 6.

Even though this is a minimal example, it can be run with `npm test`, gives test cases a little syntax sugar with `it`, improves the error reporting over a simple file full of assertions, and returns a non-zero exit code when something goes wrong. This is the basis for most popular Node test frameworks like Mocha, which we'll look at in the next section.

10.4 Test frameworks

If you're starting a new project, then you should install a test framework early on. Suppose that you're building an online blogging system, or perhaps a simple content management system. You'd like to allow people to sign in, but only allow specific users to access the administration interface. By using a test framework like Mocha or `node-tap`, you can write tests that address these specific concerns: users signing up for accounts, and administrators signing in to the admin interface. You could create separate test files for these concerns, or bundle them up as groups of test cases under "user accounts tests."

Test frameworks include scripts to run tests and other features that make it easier to write and maintain tests. This section features the Mocha test framework in technique 84 and the Test Anything Protocol (TAP; http://testanything.org/) in technique 85—two popular test frameworks favored by the Node community. Mocha is lightweight: it runs tests, provides three styles for structuring test cases,[2] and expects you to use either Node's `assert` module or another third-party module. Conversely, `node-tap`, which implements TAP, uses an API that includes assertions.

TECHNIQUE 83 Writing tests with Mocha

There are many test frameworks for Node, so it's difficult to choose the right one. Mocha is a popular choice because it's well maintained and has the right balance of features and conventions.

In general, you use a test framework to organize tests for a project. You'd like to use a test framework that other people are familiar with so they can easily navigate and collaborate without learning a new module. Perhaps you're just looking for a way to run tests the same way every time, or trigger them from an automated system.

Problem

You need to organize your tests in a way other developers will be familiar with, and run the tests with a single command.

Solution

Use one of the many open source test frameworks for Node, like Mocha.

Discussion

Mocha must be installed from npm before you can do anything else. The best way to install it is with `npm install --save-dev mocha`. The `--save-dev` option causes npm to install Mocha into `node_modules/` and update your project's package.json file with the latest version from npm. It will be saved as a development dependency.

Listing 10.9 shows an example of a simple test written with Mocha. It uses the `assert` core module to make assertions, and should be invoked using the `mocha` command-line binary. You should add `"./node_modules/mocha/bin/mocha test/*.js"` to

[2] Mocha supports API styles based on Behavior Driven Development (BDD), Test Driven Development (TDD), and Node's module system (exports).

Mocha versions

The version of Mocha we use for this chapter is 1.13.x. We prefer to run the tests by installing it locally to the project rather than as a systemwide Node module. That means that tests can be run using `./node_modules/mocha/bin/mocha test/*.js` rather than just typing `mocha`. That allows different projects to have different versions of Mocha, just in case the API changes dramatically between major releases.

An alternative is to install Mocha globally with `npm install --global mocha`, and then run tests for a project by typing `mocha`. It will display an error if it can't find any tests.

the `"test"` property in package.json—see technique 82 for more details on how to do that.

Listing 10.9 A simple Mocha test

```
var index = require('./../index');
var assert = require('assert');

describe('Amazing mathematical operations', function() {
  it('should square numbers', function() {
    assert.equal(index.square(4), 16);
  });

  it('should run a callback after a delay', function(done) {
    index.randomTimeout(function() {
      assert(true);
      done();
    });
  });
});
```

1 Group related tests with describe()

2 Include done argument for asynchronous tests

3 Call done() when asynchronous test has finished

The `describe` and `it` functions are provided by Mocha. The `describe` function can be used to group related test cases together, and `it` contains a collection of assertions that form a test case 1.

Special handling for asynchronous tests is required. This involves including a `done` argument in the callback for the test case 2, and then calling `done()` when the test has finished 3. In this example, a timeout will be triggered after a random interval, which means we need to call `done` in the `index.randomTimeout` method. The corresponding file under test is shown in the next listing.

Listing 10.10 A sample module to test

```
module.exports.square = function(a) {
  return a * a;
};

module.exports.randomTimeout = function(cb) {
  setTimeout(cb, Math.random() * 500);
};
```

1 Simple synchronous function that squares numbers

2 Asynchronous function that will run after a random amount of time

CONTROLLING SYNCHRONOUS AND ASYNCHRONOUS BEHAVIOR If `done` isn't included as an argument to `it`, then Mocha will run the test synchronously. Internally, Mocha looks at the `length` property of the callback you pass to `it` to see if an argument has been included. This is how it switches between asynchronous and synchronous behavior. If you include an argument, then Mocha will wait around for `done` to be called until a timeout is reached.

This module defines two methods: one for squaring numbers (1) and another that runs a callback after a random amount of time (2). It's just enough to demonstrate Mocha's main features in listing 10.9.

To set up a project for Mocha, the index.js file we've used in this example should be in its own directory, and at the same level should be a package.json file with a `test` subproperty of the `scripts` property set to `"./node_modules/mocha/bin/mocha test/*.js"`. There should also be a test/ directory that contains `example_test.js`.[3] With all that in place, you can run the tests with `npm test`.

When the tests are run, you should notice some dots appearing. These mark a completed test case. When the tests take more than a preset amount of time, they'll change color to denote they ran slower than is acceptable. Since `index.randomTimeout` prevents the second test from completing for a random amount of time, there will be times when Mocha thinks the tests are running too slowly. You can increase this threshold by passing `--slow` to Mocha, like this: `./node_modules/mocha/bin/mocha --slow 2000 test/*.js`. Now you don't need to feel guilty about seemingly slow tests!

Assertions per test

In listing 10.9, each test case has a single assertion. Some consider this best practice—and it can result in readable tests.

But we prefer the idea of a single concept per test. This style structures test cases around well-defined concepts, using the absolute necessary amount of assertions. This will typically be a small number, but occasionally more than one.

To see all of the command-line options, type `node_modules/mocha/bin/mocha --help` or visit http://mochajs.org/.

We've included the final package.json file in listing 10.11 in case you have trouble writing your own. You can install Mocha and its dependencies with `npm install`.

Listing 10.11 The Mocha sample project's JSON file

```
{
  "name": "mocha-example-1",
  "version": "0.0.0",
  "description": "A basic Mocha example",
  "main": "index.js",
```

[3] The file can be called anything as long as it's in the test/ directory.

```
  "dependencies": {},
  "devDependencies": {
    "mocha": "~1.13.0"
  },
  "scripts": {
    "test": "./node_modules/mocha/bin/mocha --slow 2000 test/*.js"
  },
  "author": "Alex R. Young",
  "license": "MIT"
}
```

In this technique the `assert` core module has been used, but you could swap it for another assertion library if you prefer. Others are available, like chai (https://npmjs.org/package/chai) and should.js (https://github.com/visionmedia/should.js).

Mocha is often used for testing web applications. In the next technique, you'll see how to use Mocha for testing web applications written with Node.

TECHNIQUE 84 Testing web applications with Mocha

Let's suppose you're building a web application with Node. You'd like to test it by running it in a way that allows you to send requests and receive responses—you want to make HTTP requests to test the web application works as expected.

Problem

You're building a web application and would like to test it with Mocha.

Solution

Write tests with Mocha and the standard `http` module. Consider using an HTTP module designed for testing to simplify your code.

Discussion

The trick to understanding web application testing in Node is to learn to think in terms of HTTP. This technique starts off with a Mocha test and the `http` core module. Once you understand the principles at work and can write tests this way, we'll introduce a third-party HTTP testing module to demonstrate how to simplify such tests. The built-in `http` module is demonstrated first because it's useful to see what goes on behind the scenes and to get a handle on exactly how to construct such tests.

The following listing shows what the test looks like.

Listing 10.12 A Mocha test for a web application

```
var assert = require('assert');
var http = require('http');
var index = require('./../index');

function request(method, url, cb) {
  http.request({
    hostname: 'localhost',
    port: 8000,
    path: url,
    method: method
  }, function(res) {
    res.body = '';
```

1 This function is used to make HTTP requests in the tests.

```
    res.on('data', function(chunk) {
      res.body += chunk;
    });

    res.on('end', function() {
      cb(res);
    });
  }).end();
}

describe('Example web app', function() {
  it('should square numbers', function(done) {
    request('GET', '/square/4', function(res) {
      assert.equal(res.statusCode, 200);
      assert.equal(res.body, '16');
      done();
    });
  });

  it('should return a 500 for invalid square requests', function(done) {
    request('GET', '/square', function(res) {
      assert.equal(res.statusCode, 500);
      done();
    });
  });
});
```

2 After the request has been sent, collect any data that is sent back to the client.

3 When the request and response have both finished, run the callback.

4 Ensure the response is what we expect for the /square method.

5 Ensure the server correctly raises an error for invalid requests.

This example is a test for a web service that can square numbers. It's a simple web service that expects `GET` requests and responds with plain text. The goal of this test suite is to ensure that it returns the expected results and correctly raises errors when invalid data is sent. The tests aim to simulate browsers—or other HTTP clients, for that matter—and to do so, both the server and client are run in the same process.

To run a web service, all you need to do is create a web server with `http.createServer()`. Exactly how this is done is shown in listing 10.13. Before discussing that, let's finish looking at this test.

The test starts by creating a function for making HTTP requests 1. This is to reduce the amount of duplication that would otherwise be present in the test cases. This function could be its own module, which could be used in other test files. After a request has been sent, it listens for `data` events on the response object to store any data returned by the server 2. Then it runs the provided callback 3, which is passed in from the test cases.

Figure 10.1 shows how Node can run both servers and clients in the same process to make web application testing possible.

An example of this is the test for the `/square` method that ensures `4 * 4 === 16` 4. Once that's done, we also make sure invalid HTTP query parameters cause the server to respond with a 500 error 5.

The standard assertion module is used throughout, and `res.statusCode` is used to test the expected status codes are returned.

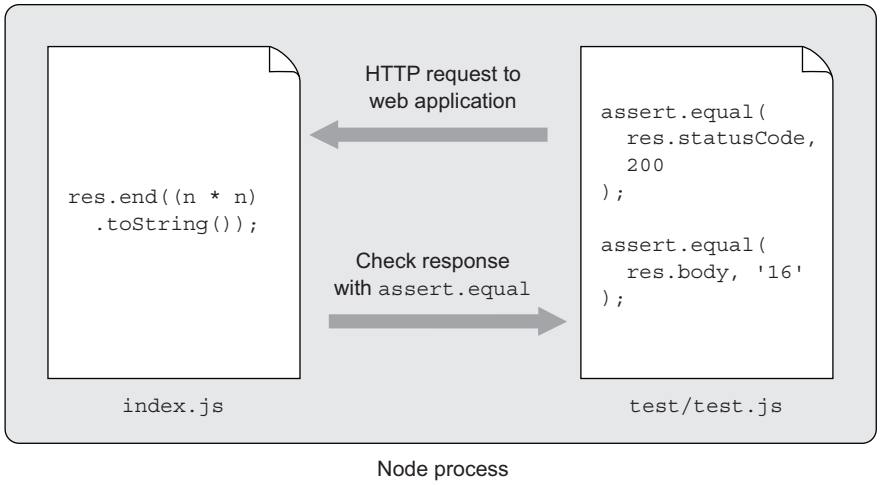


Figure 10.1 Node can run a web server and requests against it to support web application testing.

The implementation of the corresponding web service is shown in the next listing.

Listing 10.13 A web application that can square numbers

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

var server = http.createServer(function(req, res) {
  if (req.url.match(/^/square/)) {
    var params = req.url.split('/');
    var number;
    if (params.length > 1 && params[2]) {
      number = parseInt(params[2], 10);
      res.writeHead(200);
      res.end((number * number).toString());
    } else {
      res.writeHead(500);
      res.end('Invalid input');
    }
  } else {
    res.writeHead(404);
    res.end('Not found');
  }
})

server.listen(8000);

module.exports = server;
```

1 Parse out number parameter from the URL

2 Perform square calculation

3 Return a 500 when parameter is invalid

Before doing anything else, `http.createServer` is used to create a server. Near the end of the file, `.listen(8000)` is used to make the server start up and listen for connections. Whenever a request with a URL matching /square comes in, the URL is parsed for a numerical parameter 1 and then the number is squared and sent to the client 2. When the expected parameter isn't present, a 500 is returned instead 3.

One part of listing 10.12 that can be improved on is the `request` method. Rather than defining a wrapper around `http.request`, we can use a library designed specifically for testing with web requests.

The module we've chosen is `SuperTest` (https://github.com/visionmedia/supertest) by TJ Holowaychuk, who also wrote Mocha. There are other similar libraries out there. The general idea is to simplify HTTP requests and allow assertions to be made about the request.

You can add SuperTest to the development dependencies for this example by running `npm install --save-dev supertest`.

The following listing shows how the test can be refactored using the `SuperTest` module.

Listing 10.14 The refactored Mocha test that uses SuperTest

```
var assert = require('assert');
var index = require('./../index');
var request = require('supertest');

describe('Example web app', function() {
  it('should square numbers', function(done) {
    request(index)
      .get('/square/4')
      .expect(200)
      .expect(/16/, done);
  });

  it('should return a 500 for invalid square requests', function(done) {
    request(index)
      .get('/square')
      .expect(500, done);
  });
});
```

1 Pass HTTP server to SuperTest

2 Set up an assertion to make sure HTTP status is 200

3 Ensure request body contains the right answer

4 When passing invalid parameters, check a 500 is returned

Although functionally identical to listing 10.12, this example improves it by removing the boilerplate for making HTTP requests. The `SuperTest` module is easier to understand, and allows assertions to be expressed with less code while still being asynchronous. `SuperTest` expects an instance of an HTTP server 1, which in this case is the application that we want to test. Once the application has been passed to `SuperTest`'s main function, `request`, we can then make a `GET` request using `request().get`. Other HTTP verbs are also supported, and form parameters can be sent when using `post()` with the `send` method.

`SuperTest`'s methods are chainable, so once a request has been made, we can make an assertion by using `expect`. This method is polymorphic—it checks the type of the argument and acts accordingly. If you pass it a number 2, it'll ensure that the HTTP status was that number. A regular expression will make it check the response body for a match 3. These expectations are perfect for the requirements of this test.

Any HTTP status can be checked, so when we actually expect a 500, we can test for it ❹.

Though it's useful to understand how to make simple web applications and test them using the built-in `http` module, we hope you can see how third-party modules like `SuperTest` can simplify your code and make your tests clearer.

Mocha captures the zeitgeist of the current state of testing in Node, but there are other approaches that are just as valid. The next technique introduces TAP and the Test Anything Protocol, due to its endorsement by Node's maintainer and core contributors.

TECHNIQUE 85 The Test Anything Protocol

Test harness output varies based on programming language and test framework. There are initiatives to unify these reports. One such effort that has been adopted by the Node community is the Test Anything Protocol (http://testanything.org). Tests that use TAP will produce lightweight streams of results that can be consumed by compatible tools.

Suppose you need a test harness that's compatible with the Test Anything Protocol, either because you have other tools that use TAP, or because you're already familiar with it from other languages. It could be that you don't like Mocha's API and want an alternative, or are interested in learning about other solutions to testing in Node.

Problem

You want to use a test framework that's designed to interoperate with other systems.

Solution

Use Isaac Z. Schlueter's `tap` module.

Discussion

TAP is unique because it aims to bridge test frameworks and tools by specifying a protocol that implementors can use. The protocol is stream-based, lightweight, and human-readable. In comparison to other, heavier XML-based standards, TAP is easy to implement and use.

It's significant that the `tap` module (https://npmjs.org/package/tap) is written by Node's former maintainer, Isaac Z. Schlueter. This is an important seal of approval by someone highly influential in the Node community.

The example in this technique uses the number squaring and random timeout module used in technique 83 so you can compare how tests look in TAP and Mocha.

The following listing shows what the test looks like. For the corresponding module, see listing 10.10.

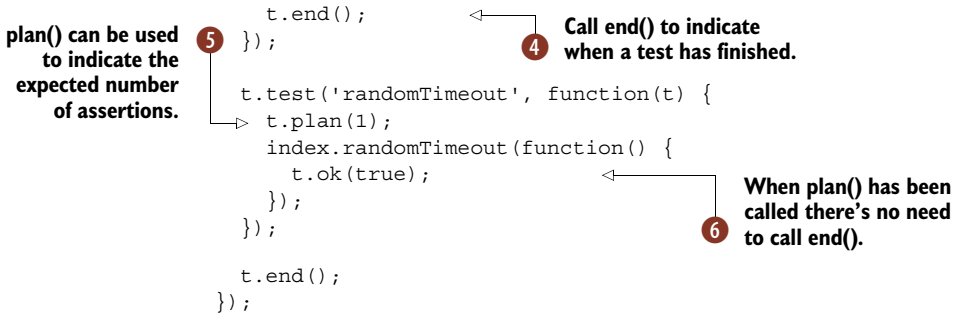
Listing 10.15 Testing with TAP

```
var index = require('./../index');
var test = require('tap').test;

test("Alex's handy mathematics module", function(t) {
  t.test('square', function(t) {
    t.equal(index.square(4), 16);
```

❶ Load the tap module, and assign a variable to the test() method.

❷ Define tests with test().

❸ Using tap's built-in assertions.



This is different from the Mocha example because it doesn't assume there are any global test-related methods like `it` and `describe`: a reference to `tap.test` has to be set up (1) before doing anything else. Tests are then defined with the `t.test()` method (2), and can be nested if needed. Nesting allows related concerns to be grouped, so in this case we've created a test case for each method being tested.

The `tap` module has built-in assertions, and we've used these throughout the test file (3). Once a test case has finished, `t.end()` must be called (4). That's because the `tap` module assumes tests are asynchronous, so `t.end()` could be called inside an asynchronous callback.

Another approach is to use `t.plan` (5). This method indicates that `n` assertions are expected. Once the last assertion has been called, the test case will finish running. Unlike the previous test case, the second one can leave off the call to `t.end()` (6).

This test can be run with `./node_modules/tap/bin/tap.js test/*_test.js`. You can add this line to the `test` property of `scripts` in the package.json file to make it run with `npm test`.

If you run the test with the `tap` script, you'll see some clean output that consolidates the results of each assertion. This is generated by one of `tap`'s submodules called `tap-results`. The purpose of the `tap-results` module is to collect lines from a TAP stream and count up skips, passes, and fails to generate a simplified report;

```
ok test/index_test.js ................................. 3/3
total ................................................. 3/3

ok
```

Due to the design of the `tap` module, you're free to run the tests with `node test/index_test.js`. This will print out the TAP stream instead:

```
# Alex's handy mathematics module
# square
ok 1 should be equal
# randomTimeout
ok 2 (unnamed assert)

1..2
# tests 2
# pass  2

# ok
```

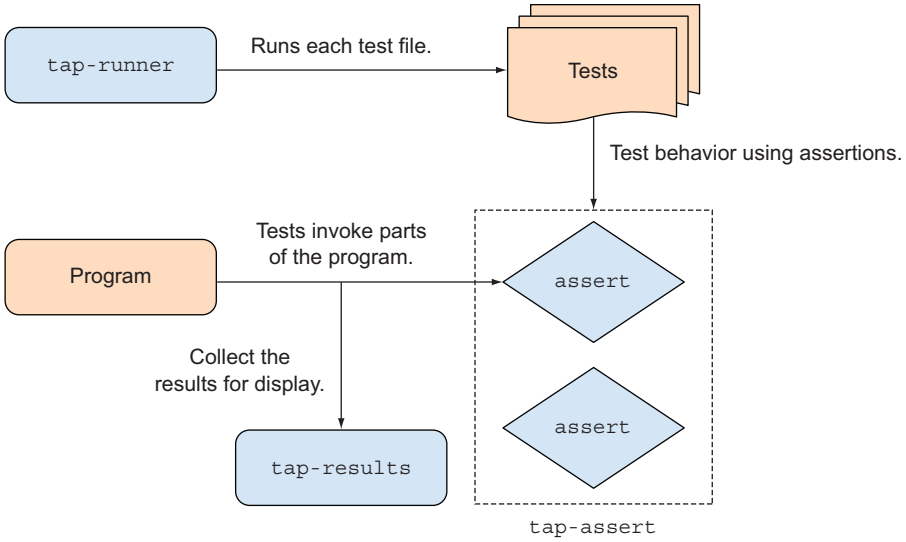


Figure 10.2 **`node-tap` uses several reusable submodules to orchestrate tests.**

Tests written with the `tap` module will still return a non-zero exit code to the shell when tests fail—you can use `echo $?` to see the exit code. Try making the test in listing 10.15 fail on purpose and take a look at `$?`.

The fact that TAP is designed around producing and consuming streams fits in well with Node's design. It's also a fact of life that tests must interact with other automated systems in most projects, whether it's a deployment system or a CI server. Working with this protocol is easier than heavyweight XML standards, so hopefully it will rise in popularity.

Figure 10.2 illustrates how some of `node-tap`'s submodules are used to test a program. Control is transferred from different modules, to your tests, back to your program, and then out through the reporter, which collects and analyzes results. The key thing to realize about this is that `node-tap`'s submodules can be reused and replaced—if you don't like the way results are displayed with `tap-results`, it could be replaced with something else.

Beyond test frameworks, real-world testing depends on several more important tools and techniques. The next section shows you how to use continuous integration servers and database fixtures, and how to mock I/O.

10.5 Tools for tests

When you're working in a team, you want to quickly see when someone has committed changes that break the tests. This section will help you to set up a continuous integration server so you can do this. It also has techniques for other project-related issues like using databases with tests and mocking web services.

TECHNIQUE 86 Continuous integration

Your tests are running, but what happens when someone makes a change that breaks the project? Continuous integration (CI) servers are used to automatically run tests. Because most test harnesses return a non-zero exit code on failure, they're conceptually simple enough. Their real value comes becomes apparent when they can easily be hooked up to services like GitHub and send out emails or instant messages to team members when tests fail.

Problem

You want to see when members of a team commit broken code so you don't accidentally release it.

Solution

Use a continuous integration server.

Discussion

You're working in a team and want to see when tests start to fail. You're already using a version control system like Git, and want to run tests whenever code is committed to a tracked repository. Or you've written an open source project and want to indicate on the GitHub or Bitbucket page that it's well tested.

There are many popular open source and proprietary continuous integration services. In this technique we'll look at Travis CI (https://travis-ci.org/), because it's free for open source projects and popular in the Node community. If you want an open source CI server that you can install locally, take a look at Jenkins (http://jenkins-ci.org/).

Travis CI provides a link to an image that shows your project's build status. To add a project, sign in with your GitHub account at travis-ci.org, and then go to the profile page at travis-ci.org/profile. You'll see a list of your GitHub projects, and toggling a switch to On will cause the repository to be updated with a service hook that will notify Travis CI whenever you push an update to GitHub.

Once you've done that, you need to add a .travis.yml file to the repository to tell Travis CI about the environment your code depends on. All you need to do is set the Node version.

Let's work through a full example and set up a project on Travis so you can see how it works. You'll need three files: a package.json, a file to test, and the .travis.yml file. The following listing shows the file we'll be testing.

Listing 10.16 A simple test to try with Travis CI

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

function square(a) {
  return a * a;
}

assert.equal(square(4), 16);
```

1 This simple test should pass when run with Travis.

This is just a simple test ❶ that we can play with to see what Travis CI does. After running, it should result in an exit code of zero—type `node test.js` and then `echo $?` to see the exit code. Put this file in a new directory so you can set up a Git repository for it later. Before that we'll need to create a package.json file. The next listing is a simple package.json that allows the tests to be run with `npm test`.

Listing 10.17 A basic package.json file

```
{
  "name": "travis-example",
  "version": "0.0.0",
  "description": "A sample project for setting up Travis CI and Node.",
  "main": "test.js",
  "scripts": {
    "test": "node test.js"
  },
  "author": "Alex R. Young",
  "license": "MIT"
}
```

Finally, you'll need a .travis.yml file. It doesn't need to do much other than tell Travis CI that you're using Node.

Listing 10.18 Travis CI configuration

```
language: node_js
 node_js:
   - "0.10"
```

Now go to GitHub.com and sign in; then click New Repository to create a public repository. We've called ours `travis-example` so people know it's purely educational. Follow the instructions on how to commit and push the project to GitHub—you'll need to run `git init` in the directory where you placed the preceding three code files, and then `git add .` and `git commit -m 'Initial commit'`. Then use `git remote add <url>` with the repository URL GitHub gives you, and push it with `git push -u origin master`.

Go to your profile at travis-ci.org/profile and toggle your new project to On. You might need to tell Travis CI to sync your project list—there's a button near the top of the page.

There's one last step before you can see any tests running on Travis CI. Make a single change in test.js—add another assertion if you like, and then commit and `git push` the change. This will cause GitHub to send an API request to Travis CI that will cause your tests to be run.

Travis CI knows how to run Node tests—it defaults to `npm test`. If you're adapting this technique to an existing project and you use another command (perhaps `make test`), then you can change what Travis CI runs by setting the `script` value in the YML file. Documentation can be found under "Configuring your build" in the documentation (http://about.travis-ci.org/docs/user/build-configuration/#script).

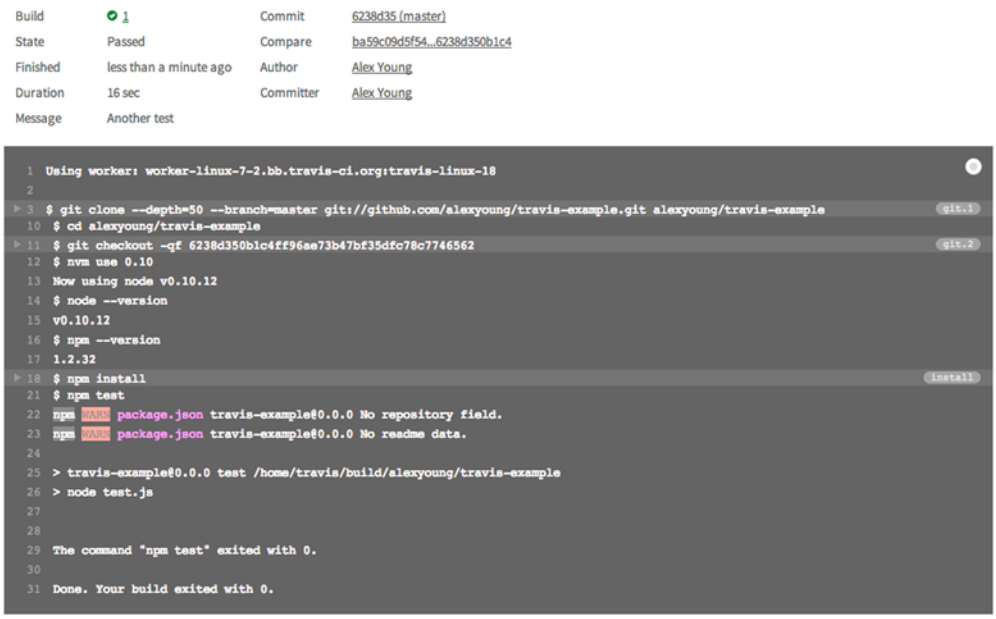


Figure 10.3 Travis CI running tests

If you go to the homepage at Travis CI, you should now see a console log with details on how the tests were run. Figure 10.3 shows what successful tests look like.

Now that you have tests running successfully, you should edit test.js to make the tests fail to see what happens.

Travis can be configured to use most of the things you expect when running tests in real-world projects—databases and other services can be added (http://about.travis-ci.org/docs/user/database-setup/), and even virtual machines.

Getting a database configured with suitable fixtures for your projects is one of the most important parts of testing. The next technique shows how to set up databases for your tests.

TECHNIQUE 87 Database fixtures

Most applications need to persist data in some way, and it's important to test that data is stored correctly. This technique explores three solutions for handling database fixtures in Node: loading database dumps, creating data during tests, and using mocks.

Problem

You need to test code that stores data in a database, or performs some other kind of I/O like sending data over a network. You don't want to access this I/O resource during testing, or you have test data that you want to preload before tests. Either way, your application is highly dependent on an I/O service, and you want to carefully test how your code interacts with it.

Solution

Preload data before the tests, or mock the I/O layer.

Discussion

The mark of well-written code is how testable it is. Code that performs I/O instinctively feels hard to test, but it shouldn't be if the APIs are cleanly decoupled.

For example, if your code performs HTTP requests, then as you've seen in previous techniques, you can run a customized HTTP server within your tests to simulate a remote service. This is known as *mocking*. But sometimes you don't want to mock I/O. You may wish to write tests that result in changes being made against a real database, albeit an instance of the database that tests can safely destroy and re-create. These types of tests are known as *integration tests*—they "integrate" disparate layers of software to deeply test behavior.

This technique presents two ways to handle database fixtures for integration tests; then we'll broaden the scope by demonstrating how to use mocks. First up: preloading data using database dumps.

Database dumps

Using database dumps is the sledgehammer of database fixture techniques. All you need is to be able to run some code before all of your other tests so you can clear out a database and drop in a pristine copy. If this test data is dumped from a database, then you can use your existing database tools for preparing and exporting the data.

Listing 10.19 uses Mocha and MySQL, but you could adapt the same principles to work with other databases and test frameworks. See technique 83 for more on Mocha.

Listing 10.19 The `assert` module

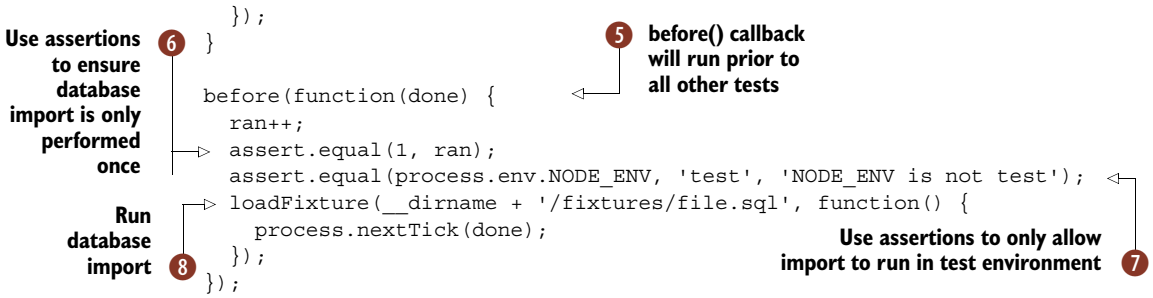
```
var assert = require('assert');
var exec = require('child_process').exec;
var path = require('path');
var ran = 0;
var db = {
  config: {
    username: 'nodeinpractice',
    password: 'password'
  }
};

function loadFixture(sqlFile, cb) {
  sqlFile = path.resolve(sqlFile);
  var command = 'mysql -u ' + db.config.username + ' ';
  command += db.config.database + ' < ' + sqlFile;

  exec(command, function(err, stdout, stderr) {
    if (err) {
      console.error(stderr);
      throw err;
    } else {
      cb();
    }
```

1 ran variable will be used to ensure fixtures aren't loaded more than once

2 loadFixture method is used to asynchronously prepare database

3 MySQL command line is prepared for importing database dump

4 Node's child_process.exec method is used to invoke mysql command-line tool to import data and overwrite existing data




```
  });
}

before(function(done) {
  ran++;
  assert.equal(1, ran);
  assert.equal(process.env.NODE_ENV, 'test', 'NODE_ENV is not test');
  loadFixture(__dirname + '/fixtures/file.sql', function() {
    process.nextTick(done);
  });
});
```

The basic principle of this example is to run a database import before the other tests. If you use this approach with your own tests, make sure the import wipes the database first. Relational databases can do this with `DROP TABLE IF EXISTS`, for example.

To actually run this test, you need to pass the filename to `mocha` before the other tests, and make sure the `test` environment is used. For example, if listing 10.19 is called `test/init.js`, then you could run these commands in the shell: `NODE_ENV=test ./node_modules/.bin/mocha test/init.js test/**/*_test.js`. Or simply place the commands in your project's package.json file under `scripts`, `test`.

The `ran` variable 1 is used to ensure the importer is only run once 6. Mocha's `before` function is used 5 to run the importer once, but if `test/init.js` is accidentally loaded elsewhere (perhaps by running `mocha test/**/*.js`), then the import would happen twice.

To import the data, the `loadFixture` function is defined 2 and run in the `before` callback 8. It accepts a filename and a callback, so it's easy to use asynchronously. An additional check is performed to make sure the import is only run in the `test` environment 7. The reasoning here is that the database settings would be set by the rest of the application based on `NODE_ENV`, and you wouldn't want to lose data by overwriting your development or production databases with the test fixtures.

Finally, the shell command to import the data is built up 3 and run with `child_process` 4. This is database-dependent—we've used MySQL as an example, but a similar approach would work with MongoDB, PostgreSQL, or pretty much any database that has command-line tools.

Using dump files for fixtures has some benefits: you can author test data with your favorite database tool (we like Sequel Pro), and it's easy to understand how it all works. If you change your database's schema or the "model" classes that work with the data, then you'll need to update your fixtures.

Creating test data with your ORM

An alternative approach is to create data programmatically. This style requires setup code—run in `before` callbacks or the equivalent in your test framework—which creates database records using your model classes.

The next listing shows how this works.

Listing 10.20 Preparing test data with an ORM

```
var assert = require('assert');
var crypto = require('crypto');

function User(fields) {
  this.fields = fields;
}

User.prototype.save = function(cb) {
  process.nextTick(cb);
};

User.prototype.signIn = function(password) {
  var shasum = crypto.createHash('sha1');
  shasum.update(password);
  return shasum.digest('hex') === this.fields.hashed_password;
};

describe('user model', function() {
  describe('sign in', function() {
    var user = new User({
      email: 'alex@example.com',
      hashed_password: 'a94a8fe5ccb19ba61c4c0873d391e987982fbbd3'
    });

    before(function(done) {
      user.save(done);
    });

    it('should accept the correct password', function() {
      assert(user.signIn('test'));
    });

    it('should not accept the wrong password', function() {
      assert.equal(user.signIn('wrong'), false);
    });
  });
});
```

1 A stand-in for a model class

2 Simulate non-blocking database save

3 Create user record to use for this test

4 Save user before the test starts

This example can be run with Mocha, and although it doesn't use a real database layer, the `User` class 1 fakes the kind of behavior you're likely to see with a library for a relational database or even a NoSQL database. A `save` function is defined that has an asynchronous API 2 so the tests look close to a real-world test.

In the `describe` block that groups together each test case, a variable called `user` is defined 3. This will be used by some of the following test cases. It's defined above their scope so they can all access it, but also because we want to persist it asynchronously in the `before` block. This runs prior to the test cases 4.

Mocking the database

The final approach that will be discussed in this technique is mocking the database API. Although you should always write *some* integration tests, you can also write tests that never touch the database at all. Instead, the database API is abstracted away.

Should I use the ORM for test data?

Like the database dump example in listing 10.19, using an ORM to create test data is useful for integration tests where you really want to talk to a database server. It's more programming effort than using database dumps, but it can be useful if you want to call methods defined above the database in the ORM layer. The downside of this technique is that a database schema change will potentially require changes in multiple test files.

JavaScript allows objects to be modified after they have been defined. That means you can override parts of the database module with your own methods to return test data. There are libraries designed to make this process easier and more expressive. One module that does this exceptionally well is Sinon.JS. The next example uses Sinon.JS along with Mocha to stub the database module.

Listing 10.21 presents an example that stubs a class that uses Redis for a user account database. The goal of the test is to check that password encryption works correctly.

Listing 10.21 Stubbing a database

```
var assert = require('assert');
var sinon = require('sinon');
var db = sinon.mock(require('./../db'));
var User = require('./../user');

describe('Users', function() {
  var fields = {
    name: 'Huxley',
    hashedPassword: 'a94a8fe5ccb19ba61c4c0873d391e987982fbbd3'
  };
  var user;

  before(function() {
    user = new User(1, fields);
    var stub = sinon
      .stub(user.db, 'hmget')
      .callsArgWith(2, null, JSON.stringify(fields));
  });

  it('should allow users to sign in', function(done) {
    user.signIn('test', function(err, signedIn) {
      assert(signedIn);
      done(err);
    });
  });

  it('should require the correct password', function(done) {
    user.signIn('wrong', function(err, signedIn) {
      assert(!signedIn);
      done(err);
```

1 Mock the database module.

2 This hashed password will be used when the user signs in.

3 Stub the Redis hmget method.

4 Make hmget call the passed-in callback, which is the third argument (index two), and pass the callback null and the fields we want to use.

```
    });
  });
});
```

This example is part of a large project that includes a package.json file and the `User` class being tested—it's available in the code samples, under `testing/mocha-sinon`.

On the third line you'll notice something new: `sinon.mock` wraps the whole database module (1). The database module is one we've defined that loads the `node-redis` module, and then connects to the database. In this test we don't want to connect to a real database, so we call `sinon.mock` to wrap it instead. This approach can be applied to other projects that use MySQL, PostgreSQL, and so on. As long as you design the project to centralize the database configuration, you can easily swap it for a mock.

Next we set up some fields that we want to use for this user (2). In an integration test, these fields would be returned by the database. We don't want to do that here, so in the `before` callback, we use a stub to redefine what Redis `hmget` does (3). The stubbing API is chainable, so we chain on the definition of what we want *our* version of `hmget` to do by using `.callsArgWith` (4).

The semantics of `.callsArgWith` can be confusing, so here's a breakdown of how it works. In the `User` class, `hmget` is called like this:

```
this.db.hmget('user:' + this.id, 'fields', function(err, fields) {
  this.fields = JSON.parse(fields);
  cb(err, this);
}.bind(this));
```

As you can see, it takes three arguments: the record key, the hash value to fetch, and then a callback that receives an optional error object and the loaded values. When we stub this, we need to tell Sinon.JS how to respond. Therefore, the first argument to `callsArgWith` is the index of the callback, which is `2`, and then the arguments that the callback should receive. We pass `null` for the error, and the user's fields serialized as a `strong`. That gives us `callsArgWith(2, null, JSON.stringify(fields))`.

This test is useful because the intent of the test is to ensure users can sign in, but only with the correct password. The sign-in code doesn't really require database access, so it's better to pass in predefined values rather than going to the trouble of accessing the database. And, because the code serializes JSON to Redis, we don't need a special library for serializing and decoding JSON—we can use the built-in `JSON` object.

Now you should know when and how to use integration tests, and mocks and stubs. All of these techniques will help you write better tests, but only if you use them in the correct circumstances. Table 10.2 provides a summary of these techniques and explains when to use each one.

The next time you want to test code that connects to a remote web service, or you need to write tests that run against a database, you should know what to do. If you've found this section interesting and you want to find out more, continue reading for some ideas on what to learn next.

Table 10.2 When to use integration tests, mocks, and stubs

Technique	When to use it
Integration testing	This means testing groups of modules. Here we've used the term to distinguish between tests that access a real database, and tests that somehow replace database access with a compatible API. You should use integration tests to ensure your database behaves the way you expect. Integration tests can help verify performance, but this is highly dependent on your test data. It may cause your tests to be more closely coupled to the database, which means that if you change the database or database API, you may need to change your test code as well.
Database dump	This is one way to preload data (before tests) into a test database. It requires a lot of work up front to prepare the data, and the data has to be maintained if you ever change the database schema. The added maintenance work is offset by the simplicity of the approach—you don't need any special tools to create SQL, Mongo, or other data files. You should use this technique when you're writing tests for a project that already has a database. Perhaps you're moving to Node from another programming language or platform, and you're using the existing database. You can take production data—being careful to remove or obscure any personal information, or other sensitive information—and then drop the resulting database export into your project's repository.
ORM fixture	Rather than creating a file to import before the tests are run, you can use your ORM module to create and store data in your test code. This can make it hard to maintain over time—any schema changes mean tests have to be carefully updated. You should use this technique for tests where algorithms are closely tied to the underlying data. By keeping the data near the code that uses it, any relating issues can be easier to understand and fix.
Mocks and stubs	Mocks are objects that simulate other objects. In this chapter you saw Sinon.JS, a library for handling mocks and stubs for tests. You should use mocks when you don't want to access an I/O resource. For example, if you're writing tests for code that talks to a payment provider like WorldPay or Stripe, then you'd create objects that behave like Stripe's API without actually communicating with Stripe. It's generally safer to ensure tests never need to access the internet, so anything that hits the network should be mocked.

10.6 Further reading

Testing is a big topic, and although this chapter has been long, there are still important topics to consider. The Node community continues to explore ways to write better tests, and it has started to bring its ideas to client-side development. One such development is Browserify (http://browserify.org)—this allows Node's module pattern and core modules like `EventEmitter` and `stream.Readable` to be used in the browser.

Some Node developers are taking advantage of Browserify to write better client-side tests. Not only can they take advantage of streams and Node's module pattern for cleaner dependency management, but they can also write Mocha or TAP tests the way they do on the server. James Halliday, the author of Browserify, created Testling, which is a browser automation module for running client-side tests.

Along with continuous integration servers, another useful test-related tool is coverage reports. These analyze code to see how much of a project is hit when the tests are

run. There may be functions, methods, or even clauses in `if` statements that never get executed, which means untested and potentially buggy code could be released to the production environment.

10.7 Summary

In this chapter you've learned how to write assertions and extend them, and how to use two popular test frameworks. When writing tests for your Node projects, you should always err on the side of readability—tests should be fast, but if they don't communicate intent, they can cause maintenance issues in the future.

Here's a recap of the main points we covered:

- Master the `assert` module by learning each method and how to ensure errors are correctly handled.
- Use test harnesses like Mocha and `node-tap` to help make tests readable and maintainable.
- Write tests for code that uses a database by loading data or using mocks and stubs.
- Improve mocks and stubs by using third-party modules like Sinon.JS.
- Develop your own domain-specific languages for tests—write functions and classes that help keep test cases lean and succinct.

One aspect of development that we haven't covered yet is debugging Node programs. This can be an important part of writing software, depending on your development style and background. If you're interested in learning the basics of the Node debugger, or want to learn more about it, then read on to dive into debugging with Node.

11 Debugging: Designing for introspection and resolving issues

This chapter covers

- Handling uncaught exceptions
- Linting Node applications
- Using debugging tools
- Profiling applications and investigating memory leaks
- Using a REPL to investigate a running process
- Tracing system calls

Understanding how errors are generated and handled in any given platform is paramount to building stable applications. Good error introspection and tests that are built-in are the best offense for debugging issues later on. In this chapter we focus on *how to prepare for* and *what to do* when things go south.

Maybe your process keeps crashing or it's using more memory than you expected. Perhaps it's stuck at 100% CPU usage. We'll look at debugging solutions for these and other problems you may encounter in your Node applications.

In the first part we'll cover Node application design for error handling and detection. In the second half we'll look at debugging specific types of problems.

11.1 Designing for introspection

When we design applications, we need to be thinking about how we'll handle errors. Relevant error logging and intervention takes thought. It also takes a good understanding of where errors can occur in order to trap them. Throughout this book, we've covered various forms of errors that can happen in Node applications. Let's cover all the types here.

11.1.1 Explicit exceptions

Explicit exceptions are those *explicitly* triggered by the `throw` keyword. They clearly indicate that something has gone wrong:

```
function formatName (name) {
  if (!name) throw new Error("name is required");
  ...
}
```

Explicit exceptions are handled by a `try/catch` block:

```
try {
  formatName();
} catch (err) {
  console.log(err.message, err.stack);
}
```

If you `throw` your own exceptions, keep these guidelines in mind:

- `throw` should be used only in synchronous functions; or in some cases, it makes sense before the asynchronous action has occurred in asynchronous functions (like API misuse).
- Always `throw` an `Error` object or something that inherits from `Error`. Using simple strings (like `throw "Oh no!"`) won't generate a stack trace, so you'll have no information as to where the error occurred.
- Don't `throw` inside Node-style callback functions; nothing exists on the stack to catch it! Instead, deal directly with the error or pass the error off to another function that can properly handle the error.

REGAINING THROW You can regain the use of `throw` for asynchronous blocks if the structures support it; some notable ones are domains, promises, or generators.

11.1.2 Implicit exceptions

Implicit exceptions are any runtime JavaScript errors *not* triggered by the `throw` keyword. Unfortunately, these exceptions can sneak into our code too easily.

One common implicit exception is `ReferenceError`, which is caused when a reference to a variable or property can't be found.

Here, we see an innocent misspelling of `data` causes an exception:

```
function (err, data) {
  res.write(dat); // ReferenceError: dat is not defined
}
```

Another common implicit exception is `SyntaxError`, most famously triggered using `JSON.parse` on invalid JSON data:

```
JSON.parse("undefined"); // SyntaxError: Unexpected token u
```

It's a good idea to wrap `JSON.parse` with a `try/catch` block, especially if you aren't in control of the input JSON data.

> **CATCH IMPLICIT EXCEPTIONS EARLY** A great way to catch implicit exceptions early is to utilize linting tools like JSHint or JSLint. Adding them to your build process helps keep your code in check. We'll talk more on subject this later in the chapter.

11.1.3 The error event

The `error` event can be emitted from any `EventEmitter` in Node. If left unhandled, Node *will* `throw` the error. These events can be the most difficult to debug if not handled, since many times they're triggered during asynchronous operations like streaming data where the call stack is minimal:

```
var EventEmitter = require('events').EventEmitter;
var ee = new EventEmitter();
ee.emit('error', new Error('No handler to catch me'));
```

This will output the following:

```
events.js:72
  throw er; // Unhandled 'error' event
 ^
Error: No handler to catch me
  at Object.<anonymous> (/debugging/domain/ee.js:5:18)
  at Module._compile (module.js:456:26)
  at Object.Module._extensions..js (module.js:474:10)
  at Module.load (module.js:356:32)
  at Function.Module._load (module.js:312:12)
  at Function.Module.runMain (module.js:497:10)
  at startup (node.js:119:16)
  at node.js:902:3
```

Luckily, we know where this error came from; we just wrote the code, after all! But in larger applications, we may have errors triggered at the DNS layer and we have no idea which module utilizing DNS just had a problem.

So, when possible, handle error events:

```
ee.on('error', function (err) {
  console.error(err.message, err.stack);
});
```

When writing your own `EventEmitters`, do yourself and your API consumers a favor and give them context to any errors in your dependencies that you're propagating upward. Also, use `Error` objects over plain strings when emitting errors so a stack trace can be found.

11.1.4 The error argument

Errors that occur during an asynchronous operation are provided as the first argument in a callback function. Unlike the previous errors we've talked about, these never cause exceptions *directly*. But they can be the source of many implicit exceptions:

```
fs.readFile('/myfile.txt', function (err, buf) {
  var data = buf.toString();
  ...
});
```

Here, we ignore the error returned from `readFile`, perhaps assuming we'll always have a buffer of the file data to continue working with. Unfortunately, the day comes when we can't read the file and we have a `ReferenceError` because `buf` is not defined.

It's more robust to just handle the asynchronous errors. A lot of times this can mean simply passing the error to another function that can gracefully handle the error:

```
function handleError (err) {
  console.error('Failed:', err.message, err.stack);
}
fs.readFile('/myfile.txt', function (err, buf) {
  if (err) return handleError(err);
  var data = buf.toString();
  ...
});
```

Handling each of these four types of errors effectively will give you much better data to work with when you're debugging issues in the future!

Even with our best efforts and tooling, though, we can still miss exceptions and have a crashed server on our hands. Let's look at designing our applications to handle these situations so we can quickly address and fix uncaught exceptions.

TECHNIQUE 88 Handling uncaught exceptions

How do you effectively handle Node crashes? One of the first things you discover when working with Node is that it terminates a process whenever an exception is

uncaught. It's important to understand why this behavior exists and how you handle uncaught exceptions in order to build robustness into your programs.

■ **Problem**

You have an uncaught exception taking down your process.

■ **Solution**

Log the exception, and shut down gracefully.

■ **Discussion**

Sometimes exceptions go uncaught. When this happens, Node by default will terminate the process. There's a good reason for this, which we'll come back to, but let's first talk about how we can change this default behavior.

With an `uncaughtException` handler set on the `process` object, Node will execute the handler *instead* of terminating your program:

```
process.on('uncaughtException', function (err) {
  console.error(err);
});
```

Yeah! Now your Node application will never crash! Although it's true that exceptions won't take down your process anymore, the drawbacks of leaving the Node program running will most likely outweigh the benefits. If you choose to keep the application running, the application could leak resources and possibly become unstable.

How does that happen? Let's look at an example of an application we intend to run for a long time: a web server. We won't allow Node to terminate the process by adding an `uncaughtException` handler that just logs the error. What do you think will happen when we have an uncaught exception while we're handling a user's request?

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

var server = http.createServer(req, res) {
  response.end('hello world');
});
server.listen(3000);

process.on('uncaughtException', function (err) {
  console.error(err);
});
```

Throws a ReferenceError since response is not defined

When a request comes in, an exception is thrown and then caught by the `uncaughtException` handler. What happens to the request? It is *leaked*, as that connection will remain open until the client times out (we also no longer have access to `res` to give a response back).

In figure 11.1, you can see an illustration of this leak happening. If we had no exception, we'd be fine, but since we had an exception, we leaked a resource.

Although this example is simplified to be clear, uncaught exceptions are a reality. Most of the time it will be open handles to sockets or files that aren't able to be closed properly. Uncaught exceptions are usually buried much deeper in the code, which makes determining what resources are being leaked even harder.

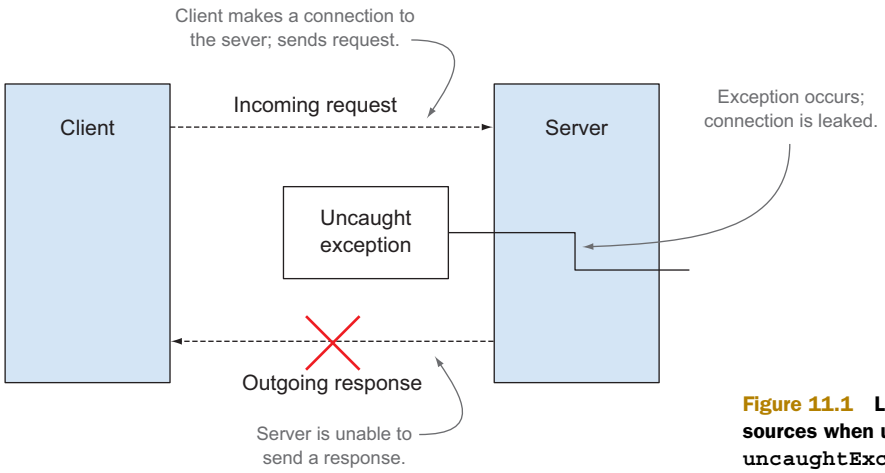


Figure 11.1 Leaking resources when using `uncaughtException`

State can also be affected, since an uncaught exception takes you out of your current context and places you in a completely different context (the `uncaughtException` handler) where you don't have references to objects in order to clean things up. In our example, we didn't have access to the `res` object in order to send a response back to the client.

So what good is the `uncaughtException` handler? It enables your application to log and restart gracefully. It's wise to treat an `uncaughtException` handler as a last chance to say your goodbyes before going down. Write out the error, perhaps send an email or do some other notification, and then gracefully kill the app:

```
process.on('uncaughtException', function (err) {
  console.error(err);
  server.close();
  setTimeout(process.exit, 5000, 1);
});
```

Stop incoming connections

Log error

Give any existing connections 5 more seconds and then kill process

The `uncaughtException` handler is a last defense. Ideally exceptions should be handled closer to the source so action can be taken to prevent leaks and instability. For that, you can use domains.

Using domains for uncaught exceptions

Whereas an `uncaughtException` casts a net over the entirety of your application code base to catch errors, domains allow you to control the *portions of code* that are monitored by the domain and handle exceptions closer to the source (more on this back in chapter 4). Let's implement the same `uncaughtException` example from earlier, but using domains instead:

```
var domain = require('domain');
var http = require('http');

var d = domain.create();
  d.run(function () {
```

Run the following code inside the domain.

Set up a new domain.

```
var server = http.createServer(req, res) {
  d.on('error', function (er) {
    res.statusCode = 500; )
    res.end('internal server error');
    server.close();
    setTimeout(process.exit, 5000, 1);
  })
  response.end('hello world');
})
server.listen(3000);
});
```

Respond to the user with an error response.

Handle any uncaught exception that occurs in the domain.

Using domains allowed us to sandbox our server code and still have access to the `res` object to give the user a response, which is an improvement on the previous example. But even though we're able to give the user a response and close the connection, it's still best practice to close out the process.

If you utilize domains, it's not a bad idea to keep an `uncaughtException` handler as a catchall for those cases where an error slips by one of your domains, or your domain's error handler throws an exception where no other domain is there to catch it.

Let's switch to a helpful way to build introspection into your application and prevent errors before they happen: linting!

TECHNIQUE 89 Linting Node applications

Lint tools can help catch a multitude of application errors when properly tuned. In our previous example, we misspelled `res`, which led to an uncaught exception. Although the code was *valid* JavaScript, we had an undefined variable being accessed. A lint tool would've caught this error.

Problem

You want to catch potential coding errors and exceptions.

Solution

Use a lint tool.

Discussion

Let's talk about setting up an application to use JSHint with Node. JSHint is an actively maintained lint tool that includes a number of customizable options for JavaScript code bases.

First, we assume you already have a package.json file set up (if not: `npm init`) for your project. Next, let's add `jshint` to our development dependencies:

```
npm install jshint --save-dev
```

Now let's configure JSHint so it knows what it's working with. We just throw a .jshintrc file—which takes a JSON configuration—in our project root. Let's look at a basic configuration for Node projects:

```
{
  "node": true,
  "undef": true
}
```

Catch any undefined variable usage.

Make JSHint understand it's working with Node. This avoids errors with known global variables and other Node-specific intelligence.

JSHint has a lot of options (http://jshint.com/docs/options/) that bend rules to match your coding style and intent, but these just shown are some good basic defaults.

To run JSHint, add the following line to the `"scripts"` block in your package.json file (if you don't have a `"scripts"` block, just add one):

```
"scripts": {
  "lint": "jshint *"
}
```

Run JSHint against all JavaScript files in your project

You can then run JSHint from your project root this way:

```
npm run lint
```

JSHint will give you output that tells you what errors it found, and you can either correct or update the options to better fit your coding style and intent. Similarly to tests, it's helpful to have your build tools run the lint tools as you push code, since it's easy to forget to run and can be automated away.

Now that we've looked at ways to prevent and effectively handle application errors, let's switch over to look at tools we can use to debug issues when they occur.

11.2 Debugging issues

We have our tests, our logging, and our linting. How do we actually debug and fix issues when they occur? Thankfully, there are a number of tools for a number of different situations. In this section we'll take a look at various and likely unrelated problems you can encounter when running your applications, and techniques to solve them. We'll start with using debuggers and move on to profiling, memory leaks, production debugging, and tracing.

TECHNIQUE 90 Using Node's built-in debugger

Whenever you need step-by-step analysis of the state of your application, a debugger can be an invaluable tool, and Node's built-in debugger is no exception. Node's built-in tooling allows you to watch variables, pause execution through breakpoints, step in and out of parts of your application, see backtraces, run an interactive context-aware REPL, and more.

Unfortunately, many shy away from the command-line tool, as it may seem intimidating at first. We want to debunk that and show how powerful it can be by walking you through most of the functionality it has to offer.

Problem

You want to run a debugger to set breakpoints, watch variables, and step through your application.

Solution

Use `node debug`.

Discussion

Let's take a simple program to debug in order to demonstrate some of the features of the debugger:

```
var a = 0;
function changeA () {
  a = 50;
}
function addToA (toAdd) {
  a += toAdd;
}
changeA();
addToA(25);
addToA(25);
```

To run the built-in debugging tool, simply use the `debug` command:

```
node debug myprogram
```

It will start the application with the debugger breaking on the first executable line:

```
< debugger listening on port 5858
connecting... ok
break in start.js:1
  1 var a = 0;
  2
  3 function changeA () {
debug>
```

To view all the available commands and debugging variables, you can type `help`:

```
debug> help
Commands: run (r), cont (c), next (n), step (s), out (o),
backtrace (bt), setBreakpoint (sb), clearBreakpoint (cb),
watch, unwatch, watchers, repl, restart, kill, list, scripts,
breakOnException, breakpoints, version
```

To continue from the default starting breakpoint, just type `cont`, or just `c` for short. Since we don't have any other breakpoints, the application will terminate:

```
debug> cont
program terminated
debug>
```

But we're *still* in the debugger and can restart the application again by using the `run` command (`r` for short):

```
debug> run
< debugger listening on port 5858
connecting... ok
break in start.js:1
  1 var a = 0;
  2
  3 function changeA () {
debug>
```

And we're back in business. We can also restart the application with the `restart` command or manually kill the application with the `kill` command if we need to.

The application is all about the letter *A*, so let's take a peek at how that changes as our application executes by making a `watch` expression for that. The `watch` function takes an expression to watch as an argument:

```
debug> watch('a')
```

We can view the state of all that we're watching using the `watchers` command:

```
debug> watchers
  0: a = undefined
```

Currently we're paused before the assignment to 0 has even been made, so we're `undefined`. Let's step into the next line with `next` (or `n` for short):

```
debug> next
break in start.js:11
Watchers:
  0: a = 0

  9 }
  10
  11 changeA();
  12 addToA(25);
  13 addToA(25);
debug>
```

Well, that's convenient: the debugger outputs our watchers for us as we step to the next section. If we were to type `watchers` again, we'd see similar output:

```
debug> watchers
  0: a = 0
```

If we ever want to remove a watch expression, we can use the `unwatch` command given the same expression we used to create it.

By default, the debugger will print just a couple lines before and after to give a sense of context. But sometimes we want to see more of what's going on. We can use the `list` command, giving it the number of lines around the current line where we're paused:

```
debug> list(5)
  6
  7 function addToA (toAdd) {
  8   a += toAdd;
  9 }
  10
  11 changeA();
  12 addToA(25);
  13 addToA(25);
  14
  15 });
debug>
```

We're currently at line 11, the `changeA` function. If we were to type `next`, we'd move to the next line, which is the `addToA` function, but let's investigate our `changeA` function more by stepping into it. To do that we just use the `step` command (or `s` for short):

```
debug> step
break in start.js:4
Watchers:
  0: a = 0

  2
  3 function changeA () {
  4   a = 50;
  5 }
  6
debug>
```

Now that we're in this function, we can step out of it at any time using the `out` command. We'll automatically step out of it once we reach the end, so we can also use `next`; let's try it:

```
debug> next
break in start.js:5
Watchers:
  0: a = 50

  3 function changeA () {
  4   a = 50;
  5 }
  6
  7 function addToA (toAdd) {
debug>
```

As you can see, our watchers updated to show that *a* is now `50`. Let's go to the next line:

```
debug> next
break in start.js:12
Watchers:
  0: a = 50

  10
  11 changeA();
  12 addToA(25);
  13 addToA(25);
  14
debug>
```

Now we're back to the line after our `changeA` function. Let's step into this next function again. Remember what command that was?

```
debug> step
break in start.js:8
Watchers:
  0: a = 50
```

```
  6
  7 function addToA (toAdd) {
  8   a += toAdd;
  9 }
  10
debug>
```

Let's explore another neat aspect of the debugger: the built-in REPL! We can access it by using the `repl` command:

```
debug> repl
Press Ctrl + C to leave debug repl
>
```

This is a standard REPL that's aware of the context that surrounds it when you used the `repl` command. So we can, for instance, output the value of the `toAdd` argument:

```
> toAdd
25
```

We can also introduce state into the application. Let's create a global `b` variable:

```
> b = 100100
```

In many ways, this behaves just like the standard Node REPL, so a lot of what you can do there, you can do here.

You can exit the REPL mode at any time with Ctrl-C. Let's do that now. You'll know you've exited because you'll get your debug prompt back:

```
debug>
```

We were in a REPL for a while, so we likely lost context when we were paused. Let's use `list` again to get our bearings:

```
debug> list()
  3 function changeA () {
  4   a = 50;
  5 }
  6
  7 function addToA (toAdd) {
  8 a += toAdd;
  9 }
  10
  11 changeA();
  12 addToA(25);
  13 addToA(25);
```

Ah yes, that's right, we were on line 8. Well, you know what, we really wanted the `changeA` function to assign `a` to 100. It's such a nice number to accompany such a nice letter! But we forgot to do that when we started the debugger. No problem! We can set a breakpoint here to save our spot by using the `setBreakpoint` function (or `sb` for short):

```
debug> setBreakpoint()
  3 function changeA () {
  4   a = 50;
```

```
  5 }
  6
  7 function addToA (toAdd) {
 *8   a += toAdd;
  9 }
  10
  11 changeA();
  12 addToA(25);
  13 addToA(25);
debug>
```

Note that our line 8 now has a star (*) next to it indicating we have a breakpoint set there. Let's change that function in our code file and save it:

```
function changeA () {
  a = 100;
}
```

Back in our debugger, we can restart the app:

```
debug> restart
program terminated<
debugger listening on port 5858
connecting... ok
Restoring breakpoint debug.js:8
break in start.js:1
  1 var a = 0;
  2
  3 function changeA () {
debug>
```

Looks like our program was restarted and the breakpoint we set is still intact. Did it get our changes? Let's see:

```
debug> list(20)
  1 var a = 0;
  2
  3 function changeA () {
  4   a = 100;
  5 }
  6
  7 function addToA (toAdd) {
  8   a += toAdd;
  9 }
  10
  11 changeA();
  12 addToA(25);
  13 addToA(25);
  14
  15 });
debug>
```

Another way we can set breakpoints right from our application code is to use the `debugger` keyword:

```
function changeA () {
  debugger;
  a = 100;
}
```

If we restart our application again, we'll always stop on any `debugger` lines. We can clear breakpoints as well using `clearBreakpoint` (or `cb` for short).

Let's look at one more topic: uncaught exceptions. Let's introduce a nasty `ReferenceError` in our `changeA` function:

```
function changeA () {
  a = 100;
  foo = bar;
}
```

If we restart our application using `restart` and then `cont` to skip the initial breakpoint, our application will crash due to an uncaught exception. We can break on these exceptions instead using `breakOnException`:

```
debug> breakOnException
debug>
```

Now, instead of crashing, we'll break first, allowing us to inspect the state of the application and use the REPL before the program terminates.

> **HELPFUL MULTIFILE DEBUGGER COMMANDS** This scenario only looked at a single file that included no other modules. The debugger also has a couple of commands that are helpful when you're within multiple files. Use `backtrace` (or `bt` for short) to get a call stack on whatever line you're currently paused at. You can also use `scripts` to get a list of loaded files and an indicator of what file you're currently in.

The built-in debugger may feel odd at first if you're used to a GUI tool for debugging applications, but it's actually pretty versatile once you get the hang of it! Just throw a quick `debugger` statement where you're working and fire it up.

TECHNIQUE 91 Using Node Inspector

Want to do everything you can with the built-in debugger, but using the Chrome DevTools interface instead? There's a module for that! It's called `node-inspector`. In this technique we'll look at how to set it up and start debugging.

Problem

You want to debug a Node application using Chrome DevTools.

Solution

Use `node-inspector`.

Discussion

Node allows remote debugging by exposing a debugging port that third-party modules and tools can hook into (including the built-in debugger). One popular module is `node-inspector`, which ties in debugging information from Node into the Chrome DevTools interface.

To set up `node-inspector`, simply install it:

```
npm install node-inspector -g
```

Don't forget the `-g` flag to install it globally. Once you have it, you can fire it up by running the following command:

```
node-inspector
```

Now `node-inspector` is ready to roll and will tell you where to reach it:

```
$ node-inspector
Node Inspector v0.7.0-2
  info - socket.io started
Visit http://127.0.0.1:8080/debug?port=5858 to start debugging.
```

You can then visit that URL in any Blink-enabled browser like Chrome or Opera. But we don't have any Node program that has an open debugging port to start debugging, so we receive an error message, as shown in figure 11.2.

Let's leave that running for now and write a little application to debug:

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

var server = http.createServer();
server.on('request', function (req, res) {
  res.end('Hello World');
});
server.listen(3000);
```

Now we can run this application exposing the debugging port:

```
$ node --debug test.js
debugger listening on port 5858
```

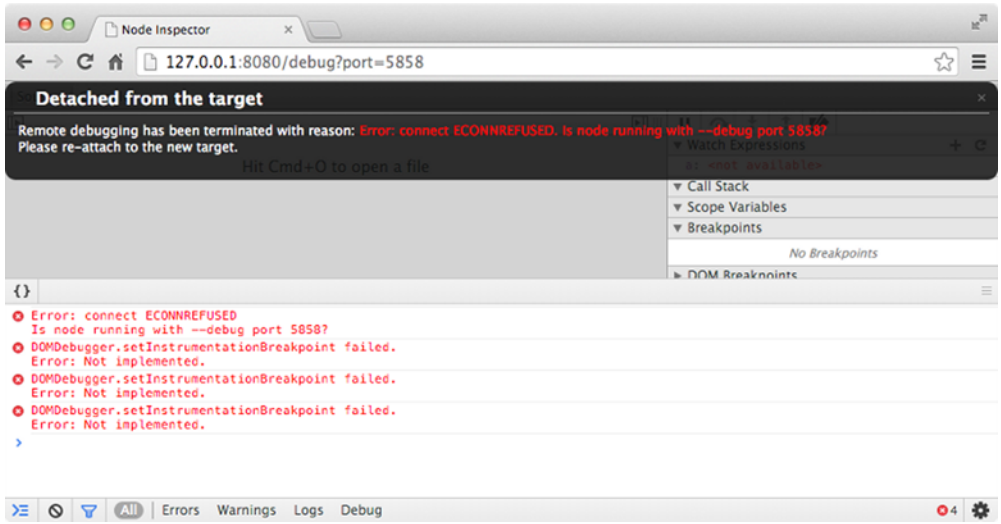


Figure 11.2 Error screen when no debugging agent is found

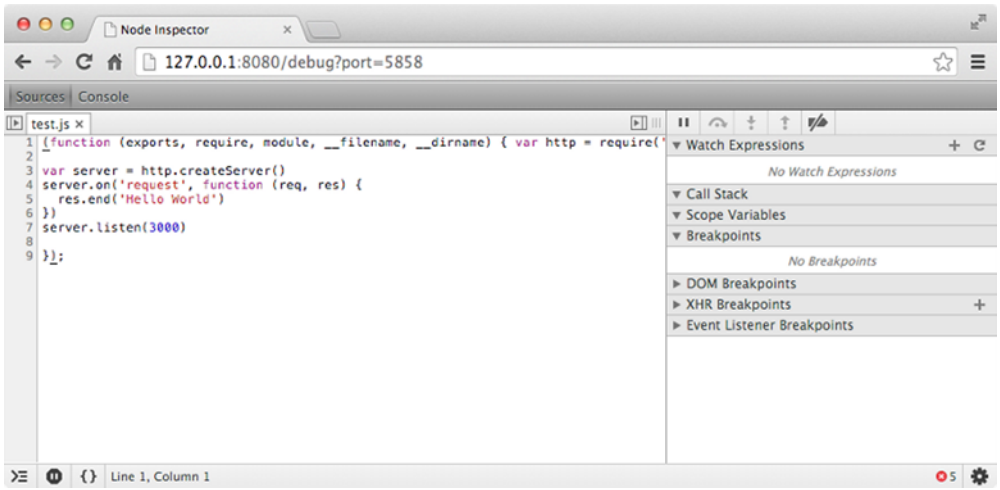


Figure 11.3 Node inspector connected to the debugger

Our application now lets us know that the debugger is listening on port 5858. If we refresh our Node inspector web page, it'll look more interesting, as shown in figure 11.3.

We can use the inspector much like the built-in debugger to set breakpoints and watch expressions. It also includes a console that's similar to the REPL to allow you to poke around at the state of your application while it's paused.

One difference between `node-inspector` and the built-in debugger is that Node doesn't automatically break on the first expression. To enable that, you have to use the `--debug-brk` flag:

```
node --debug-brk test.js
```

This tells the debugger to break on the first line until the inspector can step through or continue execution. If we reload the inspector, we can see it's paused on the first line, as shown in figure 11.4.

`node-inspector` is continually being developed to support more of Chrome DevTools' functionality.

We've looked at two ways to use debugging tools in Node: the command-line debugger and `node-inspector`. Now, let's switch to another tool for resolving performance-related issues: the profiler.

TECHNIQUE 92 Profiling Node applications

Profiling aims to answer this question: *Where is my application spending its time?* For instance, you may have a long-running web server that gets stuck at 100% CPU usage when you hit a particular route. At first glance, you might view the various functions that touch that route to see if anything stands out, or you could run a profiler and let Node tell you where it's stuck. In this technique you'll learn how to use the profiler and interpret the results.

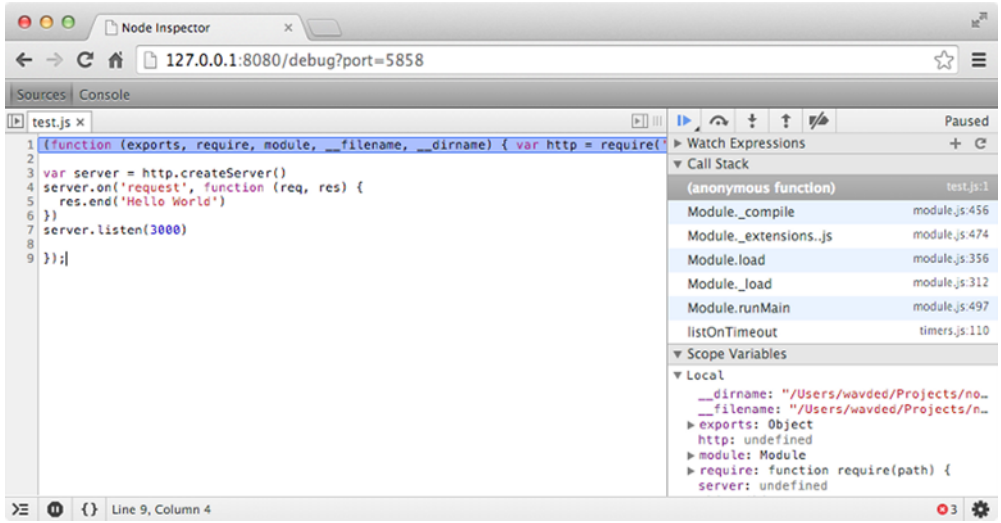


Figure 11.4 Using the `--debug-brk` flag

Problem

You want to find out where your application is spending its time.

Solution

Use `node --prof`.

Discussion

Node taps into the underlying V8 statistical profiler by the use of the `--prof` command-line flag. It's important to understand how it works in order to interpret the data.

Every two milliseconds, the profiler looks at the running application and records the function executing at that moment. The function may be a JavaScript function, but it also can come from C++, shared libraries, or V8 garbage collection. The profiler writes these "ticks" to a file named `v8.log`, where they're then processed by a special V8 tick-processor program.

Let's look at a simple application to see how this works. Here we have an application doing two different things—running a slower computational task every two seconds, and running a quicker I/O task more often:

```
function makeLoad () {
  for (var i=0;i<100000000000;i++);
}
function logSomething () {
  console.log('something');
}

setInterval(makeLoad, 2000);
setInterval(logSomething, 0);
```

We can profile this application like so:

```
node --prof profile-test.js
```

If we let it run for 10 seconds or so and kill it, we'll get a `v8.log` in that same directory. The log isn't too helpful by itself. Let's process the log by using the V8 tick-processor tools. Those tools require that you build V8 from source on your machine, but there's a handy third-party module that allows you to skip that. Just run the following command to install:

```
npm install tick -g
```

This will install the appropriate tick processor for your operating system in order to view the data. You can then run the following command in the same directory as your `v8.log` file to get some more helpful output:

```
node-tick-processor
```

You'll get output that looks similar to the following (abbreviated to show structure):

```
Statistical profiling result from v8.log,
  (6404 ticks, 1 unaccounted, 0 excluded).

[Unknown]:
  ticks total nonlib name
      1  0.0%

[Shared libraries]:
  ticks total nonlib name
   4100 64.0%   0.0% /usr/lib/system/libsystem_kernel.dylib
   211   3.3%   0.0% /Users/wavded/.nvm/v0.10.24/bin/node
  ...

[JavaScript]:
  ticks total nonlib name
   1997 31.2%  96.4% LazyCompile: *makeLoad profile-test.js:1
   7     0.1%   0.3% LazyCompile: listOnTimeout timers.js:77
   5     0.1%   0.2% RegExp: %[sdj%]
  ...

[C++]:
  ticks total nonlib name

[GC]:
  ticks total nonlib name
      1  0.0%

[Bottom up (heavy) profile]:
  Note: percentage shows a share of a particular caller in
  the total amount of its parent calls.
  Callers occupying less than 2.0% are not shown.

  ticks parent name
  4100   64.0% /usr/lib/system/libsystem_kernel.dylib

  1997   31.2% LazyCompile: *makeLoad profile-test.js:1
  1997  100.0%   LazyCompile: ~wrapper timers.js:251
  1997  100.0%     LazyCompile: listOnTimeout timers.js:77
```

Let's look at what each section means:

- *Unknown*—For that tick, the profiler couldn't find a meaningful function attached to the pointer. These are noted in the output but aren't much help beyond that and can be safely ignored.
- *Shared libraries*—These are usually underlying C++/C shared libraries; a lot of the I/O stuff happens here as well.
- *JavaScript*—This is typically the most interesting part; it includes your application code as well as Node and V8 internal native JavaScript code.
- *C++*—This is C++ code in V8.
- *GC*—This is the V8 garbage collector.
- *Bottom up (heavy) profile*—This shows a more detailed stack for the highest hitters found by the profiler.

In our case, we can see that `*makeLoad` is the hottest JavaScript function, with 1997 ticks accounted for:

```
[JavaScript]:
  ticks total nonlib name
   1997 31.2%  96.4% LazyCompile: *makeLoad profile-test.js:1
   7     0.1%   0.3% LazyCompile: listOnTimeout timers.js:77
   5     0.1%   0.2% RegExp: %[sdj%]
```

This makes sense since it has some heavy computation. Another interesting section to note is `RegExp: %[sdj%]`, which is used by `util.format`, which is used by `console.log`.

The profiler's job is to show you what functions are running most often. It doesn't necessarily mean that the function is slow, but it does mean either a lot happens in the function or it's called often. The results should serve as clues to help you understand what can be done to improve the performance. In some cases it may be surprising to find out certain functions are running hot; other times it may be expected. Profiling serves as one piece of the puzzle to help solve performance-related issues.

Another potential source of performance-related issues is memory leaks, although, obviously they're *first* a memory concern that may have performance ramifications. Let's look at handling memory leaks next.

TECHNIQUE 93 Debugging memory leaks

Before the days of Ajax and Node, there wasn't much effort put into debugging JavaScript memory leaks, since page views were short-lived. But memory leaks can happen, especially in Node programs where a server process is expected to stay up and running for days, weeks, or months. How do you debug a leaking application? We'll look at a technique that works locally or in production.

Problem

You want to debug a program leaking memory.

Solution

Use `heapdump` and Chrome DevTools.

Discussion

Let's write a leaky application to demonstrate how to use a couple of tools to debug a memory leak. Let's make a leak.js program:

```
var string = '1 string to rule them all';

var leakyArr = [];
var count = 2;
setInterval(function () {
  leakyArr.push(string.replace(/1/g, count++));
}, 0);
```

Since strings are immutable in JavaScript, we push a unique string every time into the array to intentionally grow memory and not allow the garbage collector to clean.

How do we know this application is growing in memory? We could sit and watch `top` or some other process-monitoring application. We can also test it by logging the memory used. To get an accurate read, let's force a garbage collection before logging out the memory usage. Let's add the following code to our leak.js file:

```
setInterval(function () {
  gc();
  console.log(process.memoryUsage());
}, 10000)
```

In order to use the `gc()` function, we need to expose it by running our application with the `--expose-gc` flag:

```
node --expose-gc leak.js
```

Now we can see some output showing clearly that we're growing in memory usage:

```
{ rss: 15060992, heapTotal: 6163968, heapUsed: 2285608 }
{ rss: 15331328, heapTotal: 6163968, heapUsed: 2428768 }
{ rss: 15495168, heapTotal: 8261120, heapUsed: 2548496 }
{ rss: 15585280, heapTotal: 8261120, heapUsed: 2637936 }
{ rss: 15757312, heapTotal: 8261120, heapUsed: 2723192 }
{ rss: 15835136, heapTotal: 8261120, heapUsed: 2662456 }
{ rss: 15982592, heapTotal: 8261120, heapUsed: 2670824 }
{ rss: 16089088, heapTotal: 8261120, heapUsed: 2814040 }
{ rss: 16220160, heapTotal: 9293056, heapUsed: 2933696 }
{ rss: 16510976, heapTotal: 10324992, heapUsed: 3085112 }
{ rss: 16605184, heapTotal: 10324992, heapUsed: 3179072 }
{ rss: 16699392, heapTotal: 10324992, heapUsed: 3267192 }
{ rss: 16777216, heapTotal: 10324992, heapUsed: 3293760 }
{ rss: 17022976, heapTotal: 10324992, heapUsed: 3528376 }
{ rss: 17117184, heapTotal: 10324992, heapUsed: 3635264 }
{ rss: 17207296, heapTotal: 10324992, heapUsed: 3728544 }
```

Although we know we're growing pretty steadily, we don't really know "what" is leaking from this output. For that we need to take some heap snapshots and compare them to see what's changing in our application. We'll use the third-party `heapdump` module (https://github.com/bnoordhuis/node-heapdump). The `heapdump` module allows us to take snapshots either programmatically or by sending a signal to the process (UNIX only). These snapshots can be processed using the Chrome DevTools.

Let's install the module first:

```
npm install heapdump --save-dev
```

Then include it in our leak.js file and instrument it to output a heap snapshot every 10 seconds:

```
var heapdump = require('heapdump');
var string = '1 string to rule them all';

var leakyArr = [];
var count = 2;
setInterval(function () {
  leakyArr.push(string.replace(/1/g, count++));
}, 0);

setInterval(function () {
  if (heapdump.takeSnapshot()) console.log('wrote snapshot');
}, 10000);
```

Now, every 10 seconds a file is written to the current working directory of the process that contains the snapshot. A garbage collection is automatically performed whenever a snapshot is taken. Let's run our application to write a couple snapshots and then terminate it:

```
$ node leak3.js
wrote snapshot
wrote snapshot
```

Now we can see what was written:

```
$ ls
heapdump-29701132.649984.heapsnapshot
heapdump-29711146.938370.heapsnapshot
```

The files are saved with their respective timestamps. The larger the number, the more recent the snapshot. Now we can load these files into Chrome DevTools. Open Chrome and then the Developer Tools, go to the Profiles tab, and right-click on Profiles to load a snapshot file (see figure 11.5).

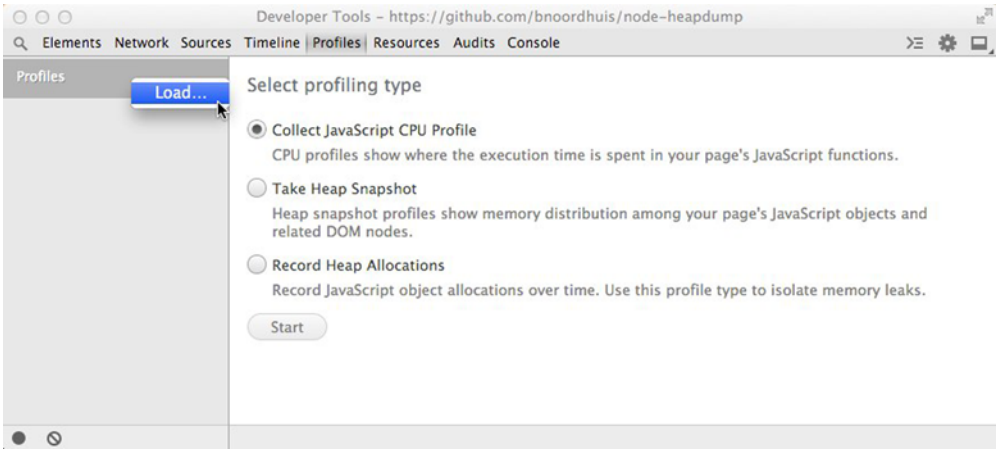


Figure 11.5 Loading a heap snapshot into the Chrome DevTools

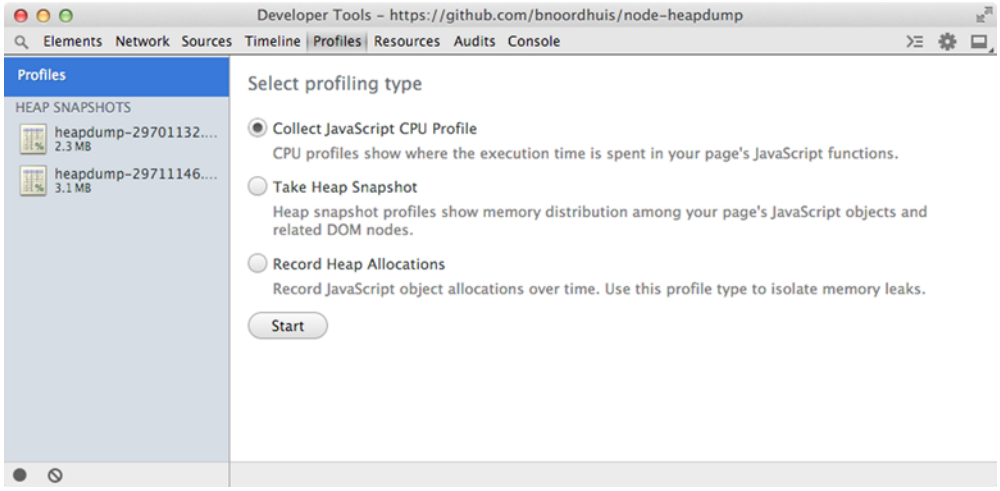


Figure 11.6 **Loading a second snapshot for comparison**

To compare our two snapshots, let's load them in the order we took them (see figure 11.6).

Now that we have them loaded, we can do some investigation. Let's select the second one and then choose the Comparison option. Chrome will automatically select the previous snapshot to compare to (see figure 11.7).

Now we can see something immediately interesting in our view—a lot of strings are being created and not being garbage collected (see figure 11.8).

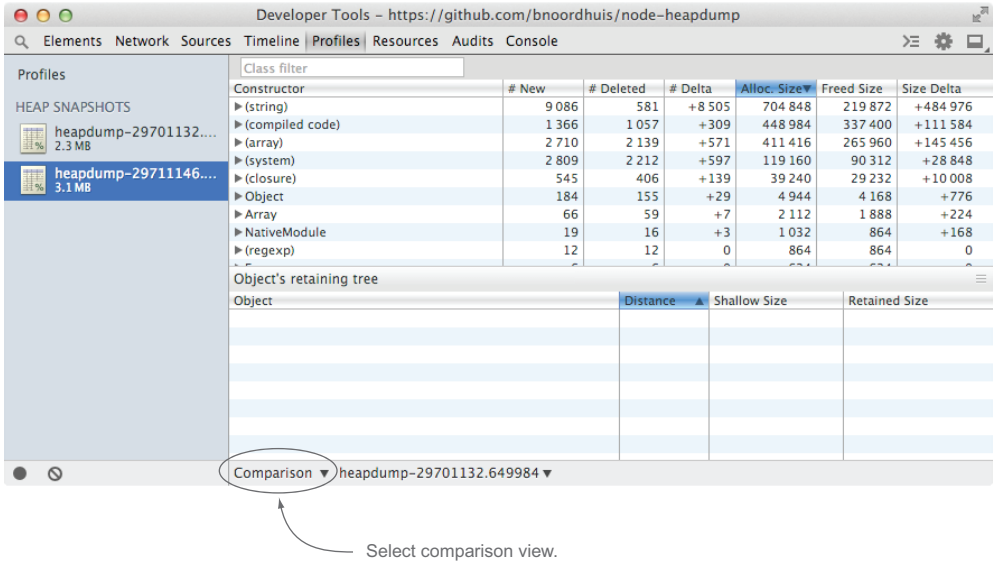


Figure 11.7 **Using the comparison view**

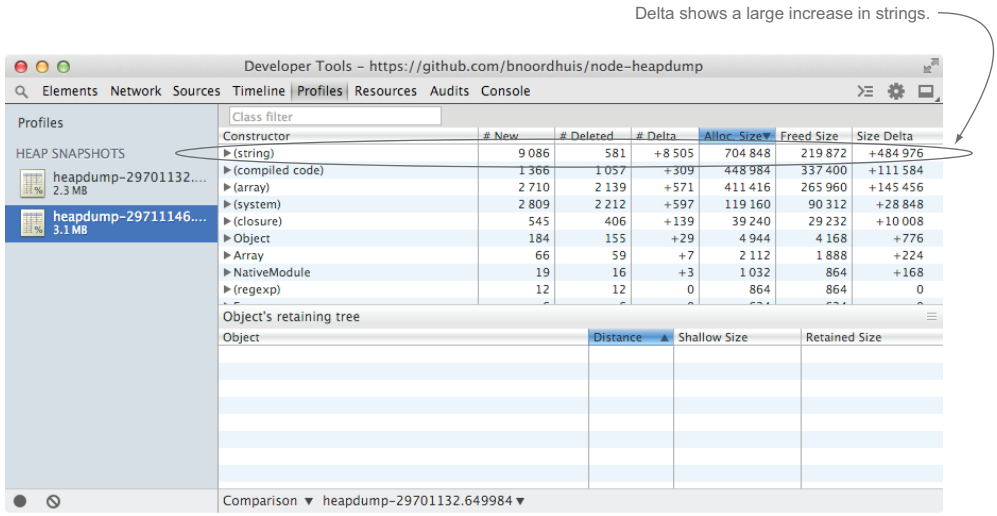


Figure 11.8 Examining memory allocations between the snapshots

So we can see that strings could be a problem here. But what strings are getting created? Here we have to do some investigation. Expanding the (string) tree will show us the largest strings first—typically application source code and some larger strings used in Node core and V8. But when we scroll down, we start to see strings generated in our application, and lots of them. By clicking one, we can see the retaining tree, or its relationship to other objects (see figure 11.9).

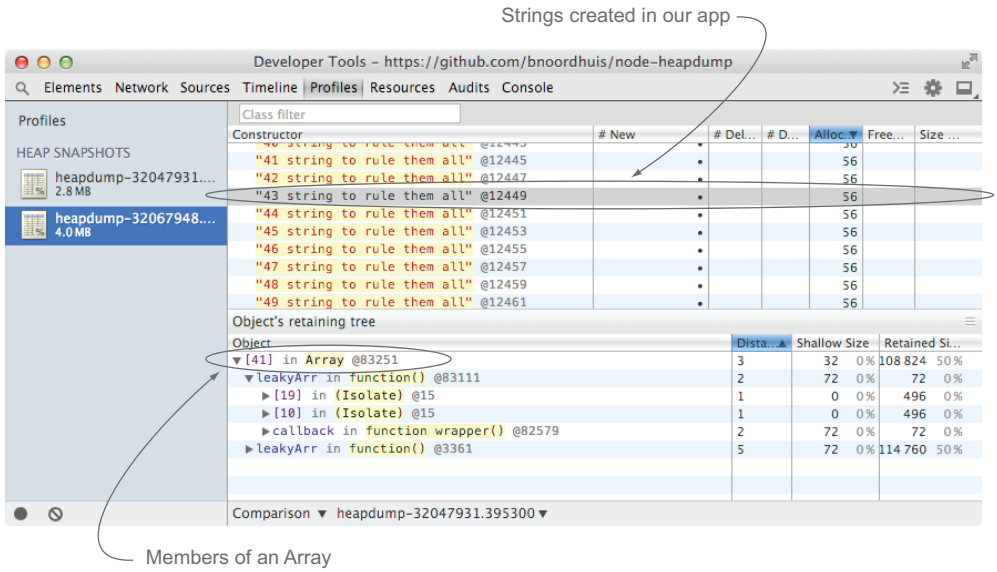


Figure 11.9 Drilling down to the types of data being created in memory

In this exercise, we had a hunch we were going to leak strings stored inside the `leakyArr` variable. But this exercise shows the relationship between the code and the tools to inspect memory usage. As a developer, you'll know your source code, and the clues you get inside DevTools will be specific to your code and modules. The Comparison view can give a great snapshot of what's changing.

We only talked about one way of creating snapshots. You can also send a SIGUSR2 (on *NIX systems) to a process with `heapdump` to take a snapshots at will:

```
kill -USR2 1120
```

Just remember that it'll write the snapshot to the CWD of the process and will fail silently if the CWD isn't writable by the process user.

You can also be programmatically clever, depending on your needs. For example, you could set up `heapdump` to take a snapshot after a certain memory threshold is breached, or if it grows faster than some expected limit given an interval.

Taking heap snapshots is something you can do in production for a small performance penalty while the snapshot is being written to disk. Let's turn our attention to another technique you can use in production that has a minimal penalty and lets you poke around at the application state: using a REPL.

TECHNIQUE 94 Inspecting a running program with a REPL

Attaching a debugger to a production process isn't a viable option, as we don't want to be pausing execution or adding a performance tax of running the V8 debugger. So how can we debug live or performance-sensitive issues? We can use a REPL to dive into the process and inspect or change state. In this technique we'll first look at how a REPL works in Node, and how to set up your own REPL server and client. Then we'll turn to inspecting a running process.

Problem

You want to interact with a running process to inspect or change its state.

Solution

Set up a REPL in the process and a REPL client to access.

Discussion

The Node REPL is a great way to play around and experiment with JavaScript and Node. The simplest way to play around with a REPL is to run Node without any arguments, as shown in figure 11.10.

But you can create your own REPLs using the built-in `repl` module. In fact, Node uses the same module when you type `node`. Let's make our own REPL:

```
var repl = require('repl');
repl.start({
  input: process.stdin,
   output: process.stdout
});
```

Use stdin stream as input stream to the REPL

Use stdout stream as output stream from the REPL

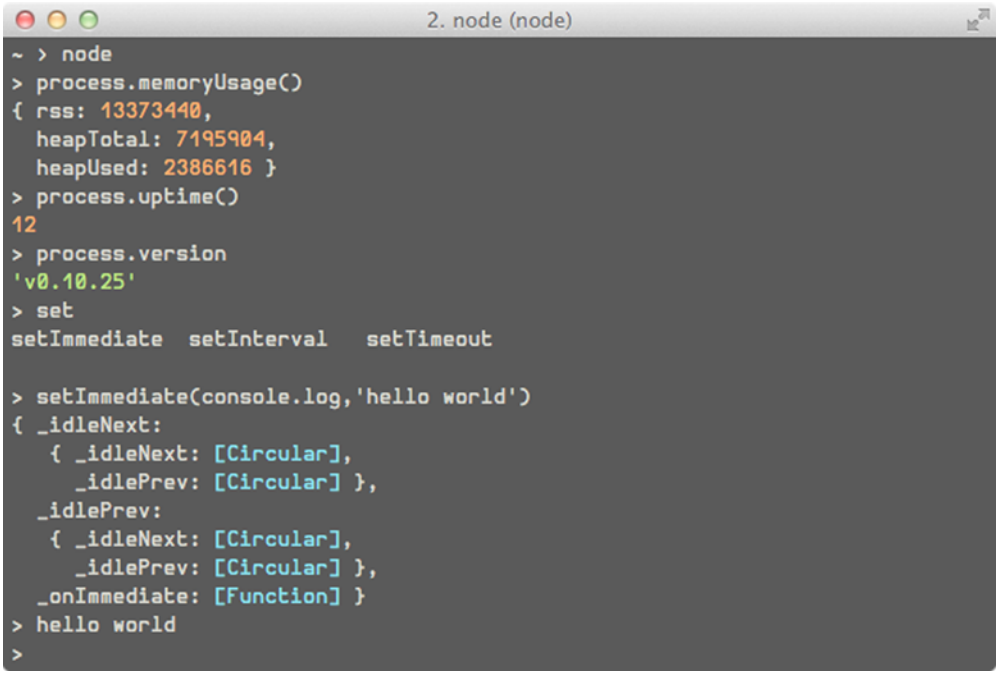


Figure 11.10 Sample Node REPL session

Executing this program creates a REPL that looks and functions much like `node` does:

```
$ node repl-basic.js
> 10 + 20
30
>
```

But we don't need to use the process's `stdio` for input and output; we can use a UNIX or a TCP socket! This allows us to connect to a long-running process from the outside. Let's make a TCP REPL server:

```
var net = require('net');
var repl = require('repl');

net.createServer(function (socket) {
  var r = repl.start({
    input: socket,
    output: socket
  });
  r.on('exit', function() {
    socket.end();
  });
}).listen(1337);

console.log('node repl listening on 1337');
```

Use incoming socket (a Duplex stream) as the input and output stream for REPL

When REPL is exited, end connection

Now if we fire up our REPL server, it'll be listening on port 1337:

```
$ node repl-tcp.js
node repl listening on 1337
```

We can then connect to it with a TCP client like telnet or Netcat. You can do this in a separate terminal window:

```
$ nc localhost 1337
> 10 + 20
30
> exit
$
```

That's cool! But it doesn't behave like our basic REPL (see figure 11.11) or the `node` command:

- The Tab key doesn't autocomplete available properties and variables.
- We don't have any readline support, so the Up Arrow key doesn't give us any command history.
- No color or bold output.

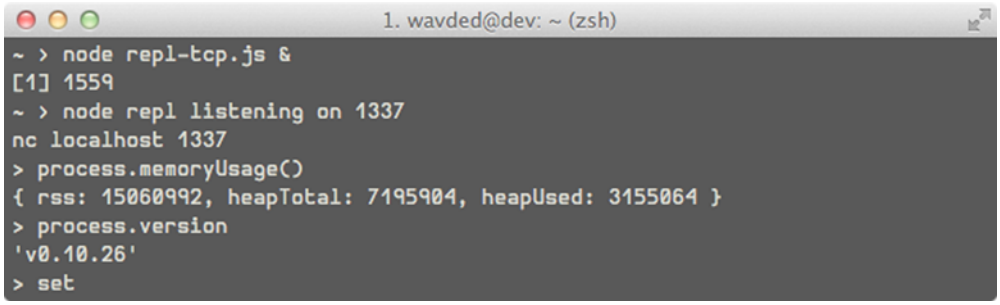


Figure 11.11 Using Netcat against a REPL server

The reasons for this are twofold. First, the `repl` module can't determine that we're running a TTY (terminal) session, so it provides a minimal interface avoiding the use of ANSI/VT100 escape codes for color and formatting. These escape codes end up being noise to clients like Netcat. Second, our client isn't behaving like a TTY. It isn't sending the proper input codes to get nice perks like autocomplete behavior or history.

In order to change this behavior, we need to modify both the server and client. First, to send proper ANSI/VT100 escape codes for things like color and bold output, we need to add the terminal option to our REPL configuration:

```
var net = require('net');
var repl = require('repl');

net.createServer(function (socket) {
  var r = repl.start({
```

```
    input: socket,
    output: socket,
    terminal: true        <-- Treat output as a TTY stream
  });
  r.on('exit', function() {
    socket.end();
  });
}).listen(1337);

console.log('node repl listening on 1337');
```

Second, to get the input tab completion and readline, we need to create a REPL client that can send the raw TTY input to the server. We can create that using Node:

```
var net = require('net');
var socket = net.connect(1337);

process.stdin.setRawMode(true);
process.stdin.pipe(socket);
socket.pipe(process.stdout);

socket.once('close', function () {
  process.stdin.destroy();
});
```

Connect to the REPL TCP server.

Pipe input from stdin to the socket.

Pipe output from the socket to stdout.

Treat stdin as a raw TTY input stream. This allows, for example, the Tab key and Up Arrow key to behave as you'd expect in a modern terminal session.

When the connection is terminated, destroy the stdin stream, allowing the process to exit.

Now we can start our REPL server with terminal support:

```
$ node repl-tcp-terminal.js
node repl listening on 1337
```

We can connect to the server with our REPL client in another terminal session:

```
$ node repl-client.js
> 10 + 20
30
> .exit
$
```

Now our REPL session behaves as if we were running the `node` or a basic REPL. We can use autocomplete, access our command history, and get colored output. Sweet!

Inspecting a running process

We've discussed how to use the `repl` module to create connection points and use various clients to access it. We did this to get you comfortable setting up REPL instances on your applications so you can use them to inspect a running process. Now, let's get practical and instrument an existing application with a REPL server, and interact with it using the REPL client we created.

First, let's create a basic HTTP server:

```
var http = require('http');
var server = http.createServer();
server.on('request', function (req, res) {
  res.end('Hello World');
});
server.listen(3000);
console.log('server listening on 3000');
```

This should look familiar. But let's expose this server to our REPL server by adding the following code:

```
var net = require('net');
var repl = require('repl');
net.createServer(function (socket) {
  var r = repl.start({
    input: socket,
    output: socket,
    terminal: true,
    useGlobal: true
  });
  r.on('exit', function() { socket.end() });
  r.context.server = server;
}).listen(1337);
console.log('repl listening on 1337');
```

Allow scripts to be executed in global context versus a separate context

Expose our server instance to REPL

A NOTE ABOUT USEGLOBAL When enabled, whenever you create a new variable (like `var a = 1`), it will be put in the global context (`global.a === 1`). But `a` now will also be accessible in functions run in a later turn in the event loop.

We exposed the server by setting a property on `r.context`. We can expose anything we want to the REPL in order to interact with it. It's important to note that we also can *override* anything already existing in the context. This includes all the standard Node global variables like `global`, `process`, or `Buffer`.

Now that we have our server exposed, let's see how we can inspect and debug our HTTP server. First let's fire up our HTTP and REPL servers:

```
$ node repl-app.js
server listening on 3000
repl listening on 1337
```

Now let's use our REPL client to tap into the server:

```
$ node repl-client.js
>
```

We can tap into useful bits of information right away. For instance, we can see how long our process has been running, or how its memory usage is:

```
> process.uptime()
115
 > process.memoryUsage()
{ rss: 17399808,
  heapTotal: 7195904,
  heapUsed: 4146840 }
```

Uptime in seconds

Memory in bytes

We also exposed the `server` object, and we can access that by just typing `server`:

```
> server
{ domain: null,
  _events:
  ...
  _connectionKey: '4:0.0.0.0:3000' }
```

Let's see how many connections are currently active:

```
> server.connections
0
```

Clearly this would be more interesting in a production context, since we are the only ones using the server and we haven't made a connection yet! Let's hit http://localhost:3000 in our browser and inspect the connections again and see if they've changed:

```
> server.connections
6
```

Connections vary on browser/client

That works. Let's instrument something more complex. Can you think of a way to start tallying the number of requests coming in to our server using the REPL?

> **ADDING INSTRUMENTATION** One powerful aspect of a REPL is the ability to add instrumentation to help us understand behavior in our application as it's happening. This is especially handy for tricky problems where restarting the application loses our precious state and we have no idea how to duplicate the issue except to wait for it to happen again.

Since our HTTP server is an `EventEmitter`, we can add another request handler that will be called on every request to instrument it with the behavior we want using the REPL:

```
> var numReqs = 0
 undefined
> function trackReqs (req, res) {
 ..... numReqs++
..... }
undefined
> server.on('request', trackReqs)
 { domain: null,
  _events:
  ...
  _connectionKey: '4:0.0.0.0:3000' }
>
```

Create variable to store the number of requests

Create handler function for incoming requests that increments the request count

Add handler to the request event

Now we're tracking incoming requests. Let's hit Refresh a few times on our browser and see if it worked:

```
> numReqs
8
```

Excellent. Since we have access to the request objects, we can inspect any information about requests available to us: IP addresses, headers, paths, and so on. In this example

we exposed an HTTP server, but any objects can be put on the context where it makes sense in your application. You may even consider writing a module exposing commonly used methods in the REPL.

Some issues can't be resolved at an application level and need deeper system introspection. One way to gain deeper understanding is by tracing.

TECHNIQUE 95 Tracing system calls

Understanding how the underlying system calls work can really help you understand a platform. For example, Python and Node both have functionality to perform DNS lookups, but they go about it differently at a lower level. And if you're wondering why one is behaving differently than the other, tracing tools will show you that!

At their core, tracing tools monitor underlying system calls (typically C function names, arguments, and return values) that an application or multiple applications are making, and do interesting things with the data (like logging or statistics).

Tracing helps in production. If you have a process stuck at 100% and are unsure why, a tracer can help expose the underlying state at the system level. For example, you may discover in this instance that you exceeded the allowed open files for a process, and all I/O attempts are being rejected, causing the problem. Since tracing tools aren't performance intrusive like a profiler, they can be valuable assets.

Problem

You want to understand what's happening in your application at the system level.

Solution

Use tracing tools specific to the operating system to gain introspection.

Discussion

All the techniques we've discussed so far have been system-agnostic. This one is OS-specific. There are a lot of different tools, but most are unique to an operating system. For our example, we'll use the Linux-specific tool called `strace`. Similar tools exists for Mac OS X/Solaris (`dtruss`) and Windows (`ProcessMonitor`: http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx).

A tracing program is essentially a dump of system calls as they happen in a process. If you're unfamiliar with the underlying OS, prepare to learn! We'll walk through tracing a simple application to see what's happening at the OS level when we run it to learn how to read trace logs.

Let's write an extremely simple program to trace:

```
console.log('hello world');
```

This seems innocent enough. To see what's going on behind the scenes, let's trace this:

```
sudo strace -o trace.out node hello
```

You'll see the program output "hello world" and exit as expected. But we also got a dump of every system call in trace.out. Let's examine that file.

Right at the top we can see our first call, which makes sense. We're executing `/usr/bin/node`, passing it the arguments `node` and `hello`:

```
execve("/usr/bin/node", ["node", "hello"], [/* 24 vars */]) = 0
```

If you ever wondered why `process.argv[0]` is `node` and `process.argv[1]` is the path to our Node program, now you can see how the underlying call is being made! The `strace` output tells us the arguments passed and the return value.

To find more information about what `execve` is (and any other system call), we can just look at the `man` pages on the host if available (best option), or if not, look online:

```
man execve
```

MORE ON MAN COMMAND Manual pages also include error codes that are helpful to get more details on, for example, what ENOENT or EPERM mean on an operating system. Many of these error codes can be found in the `openman` page.

Let's examine more of this file. Many of the initial calls are loading the shared libraries `libuv` needs. Then we get to our application:

```
getcwd("/home/wavded", 4096) = 13
...
stat("/home/wavded/hello", 0x7fff082fda08) = -1
  ENOENT (No such file or directory)
stat("/home/wavded/hello.js",
  {st_mode=S_IFREG|0664, st_size=27, ...}) = 0
```

We can see Node grabbing the current working directory and then looking up our file to run. Note that we executed our application *without* the .js extension, so Node first looks for a program called "hello" and doesn't find it, and then looks for hello.js and is successful. If we were to run it with the .js extension, you wouldn't see the first stat call.

Let's look at the next interesting bit:

```
open("/home/wavded/hello.js", O_RDONLY) = 9
fstat(9, {st_mode=S_IFREG|0664, st_size=27, ...}) = 0
...
read(9, "console.log('hello world')\n", 27) = 27
close(9) = 0
```

Here we `open` the hello.js file in read-only mode and get assigned a file descriptor. File descriptors are just integers assigned by the OS. But to understand the subsequent calls, we should take note that `9` is the number assigned for hello.js until we see a subsequent `close` call.

After `open`, we then see an `fstat` to get the file's size. Then we read the contents of the file in the `read` line. The `strace` output also shows us the contents of the buffer we used to store the file. We then `close` the file descriptor.

A trace output file won't show us any application code being run. We just see the system *effects* of what's being run. That is, we won't see V8 parsing or executing our `console.log` but we'll see the underlying write out to `stdout`. Let's look at that next:

```
write(1, "hello world\n", 12) = 12
```

Recall from chapter 6 that every process has three file descriptors automatically assigned for stdin (0), stdout (1), and stderr (2). We can see that this `write` call uses stdout (1) to write out `hello world`. We also see that `console.log` appends a newline for us.

Our program eventually exits on the last line of `trace.out`:

```
exit_group(0)
```

The zero (0) here represents the process exit code. In this case it's successful. If we were to exit with `process.exit(1)` or some other status, we'd see that number reflected here.

Tracing a running process

So far we've used `strace` to start and trace a program till it exits. How about tapping into a running process?

Here we can just grab the PID for the process:

```
ps ax | grep node
```

The first number in the row is our PID:

```
32476 ? Ssl 0:08 /usr/bin/node long-running.js
```

Once we have our PID, we can run `strace` against it:

```
sudo strace -p 32476
```

All the currently running system calls will output to the console.

This can be a great first line of defense when debugging live issues where CPU is pegged. For example, if we've exceeded our `ulimit` for a process, this will typically peg our CPU, since `open` system calls continually will fail. Running `strace` on the process would quickly show a bunch of `ENFILE` errors occurring. And from the `openman` page, we can see a nice entry for the error:

```
ENFILE The system limit on the total number of
open files has been reached.
```

> **LISTING OPEN FILES** In this case, we can use another handy Linux tool called `lsof` to get a list of open files for a process given a PID to further investigate what we have open right now.

We can also get a CPU pegged at 100% and open up `strace` and see just the following repeating over and over:

```
futex(0x7ffbe00008c8, FUTEX_WAKE_PRIVATE, 1) = 1
```

This, for the most part, is just event loop noise, and it's likely that your application code is stuck in an infinite loop somewhere. Tools like `node --prof` would help at this point.

About other operating system tools

The actual system calls we looked at will be different on other operating systems. For example, you'll see `epoll` calls being made on Linux that you won't ever see on Mac OS X because `libuv` uses `kqueue` for Mac. Although most OSes have POSIX methods like `open`, the function signatures and error codes can vary. Get to understand the machines you host and develop your Node applications on to make best use of the tracing tools!

> **HOMEWORK!** Make a simple HTTP server and trace it. Can you find out where the port is being bound, where connections are being accepted, and where responses are being written back to the client?

11.3 Summary

In this chapter we looked at debugging Node applications. First, we focused on error handling and prevention:

- How do you handle errors that your application generates?
- How are you notified about crashes? Do you have domains set up or an `uncaughtException` handler?
- Are you using a lint tool to help prevent exceptions?

Then, we focused on debugging specific problems. We used various tools available in Node and third-party modules. The big thing to take away is knowing the right tool for the job, so when a problem arises, you can assess it and gain useful information:

- Do you need to be able to set breakpoints, watch expressions, and step through your code? Use the built-in `debug` command or `node-inspector`.
- Do you need to see where your application is spending its time? Use the Node built-in profiler (`node --prof`).
- Is your application using more memory than expected? Take heap snapshots and inspect the results.
- Do you want to investigate a running process without pausing it or incurring a performance penalty? Set up and use a REPL server.
- Do you want to see what underlying system calls are being made? Use your operating system's tracing tools.

In the next chapter we'll dive into writing web applications with Node!

12 Node in production: Deploying applications safely

This chapter covers

- Deploying Node applications to your own server
- Deploying Node applications to cloud providers
- Managing packages for production
- Logging
- Scaling with proxies and cluster

Once you've built and tested a Node application, you'll want to release it. Popular PaaS (platform as a service) providers like Heroku and Nodejitsu make deployment simple, but you can also deploy to private servers. Once your code is out there, you'll need to cope with unexpected errors, service outages, and bugs, and monitor performance.

This chapter shows you how to safely release and maintain Node programs. It covers privately hosted servers that use Apache and nginx, WebSockets, horizontal scaling, automated deployment, logging, and ways to boost performance.

12.1 Deployment

In this section you'll learn how to deploy Node applications to popular cloud providers and your own private servers. It's likely that you'll only typically use one of these approaches, depending on the requirements of your application or employer, but being familiar with both is instructive. For example, the Git-based workflow employed by Heroku has influenced how people deploy applications to servers they control, and with a bit of knowledge you can set up a server without having to call for help from a DevOps specialist.

The first technique we cover is based on Windows Azure, Heroku, and Nodejitsu. This is probably the easiest way to deploy web applications today, and cloud providers have free plans that make it cheap and painless to share your work.

TECHNIQUE 96 Deploying Node applications to the cloud

This technique outlines how to use Node with PaaS providers, and has tips on how to configure and maintain applications in production. The focus is on the practical aspects of deployment and maintenance, rather than pricing or business models.

You can try out Heroku and Azure for free, so follow along if you've ever wanted to run a Node application in the cloud.

Problem

You've built a Node web application and want to run it on servers so people can use it.

Solution

Use a PaaS provider like Heroku or Nodejitsu.

Discussion

We'll look at three options for cloud deployment: Nodejitsu, Heroku, and Windows Azure. All of these services allow you to deploy Node web applications, but they all handle things slightly differently. The methods for uploading an application and configuring it vary, even though the fundamental concepts are the same.

Nodejitsu is an interesting case because it's dedicated to Node. On the other hand, Windows Azure supports Microsoft's software development tools, programming languages, and databases. Azure even has features beyond web application hosting, like databases and Active Directory integration. Heroku draws on a rich community of partners that offers add-ons, whereas Azure is more of a full-service offering.

If you look in the source code provided with this book, you should find a small Express application in production/inky. This is the application we used to research this technique, and you can use it as a sample application to try each service provider. Nodejitsu and Azure's documentation includes examples based on Node's `http` module, but you really need something with a package.json to see how things work for typical Node applications.

The first service provider we'll look at is Nodejitsu (https://www.nodejitsu.com/). Nodejitsu is based in New York, and has data centers in North America and Western Europe. Nodejitsu was founded in 2010, and has funding from the Bloomberg Beta fund.

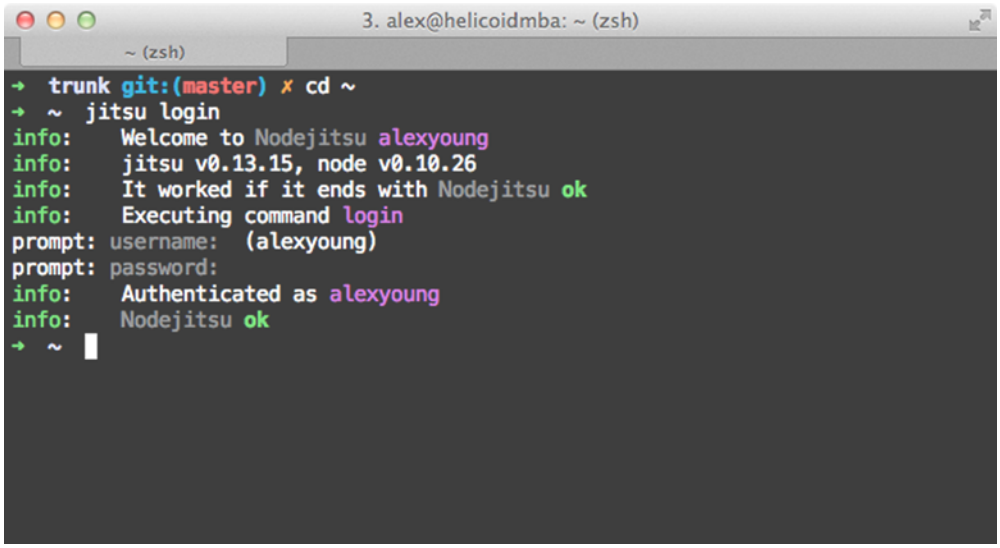


Figure 12.1 **The `jitsu` command-line client allows you to sign in.**

To get started with Nodejitsu, you'll need to register an account. Go to Nodejitsu.com and sign up. You can sign up without selecting a pricing plan if you intend to release an open source project through Nodejitsu.

Nodejitsu has a command-line client called `jitsu`. You can install it with `npm install -g jitsu`. Once npm has finished, you'll need to sign in—type `jitsu login` and enter your username and password. This will save an API token to a file called ~/.jitsuconf, so your password won't be stored locally. Figure 12.1 shows what this process looks like in the terminal.

To deploy an application, type `jitsu deploy`. The `jitsu` command will prompt with questions about your application, and then set it up to run on a temporary subdomain. If you're using an Express application, it'll automatically set `NODE_ENV` to production, but you can edit this setting along with other environmental variables in the web interface. In fact, the web interface can do most of the things the `jitsu` command does, which means you don't necessarily need a developer on hand to do basic maintenance chores like restarting applications.

Figure 12.2 shows a preview of Nodejitsu's web interface, which is called *WebOps*. It allows you to stop and start applications, manage environmental variables, roll back to earlier versions of your application, and even stream logs in real time.

Unsurprisingly Nodejitsu is heavily tailored toward Node applications, and the deployment process is heavily influenced by npm. If you have a strong grasp of npm and package.json files, and your projects are all Node applications, then you'll feel at home with Nodejitsu.

Another PaaS solution that's popular with Node developers is Heroku. Heroku supports several programming languages and platforms, including Node, and was

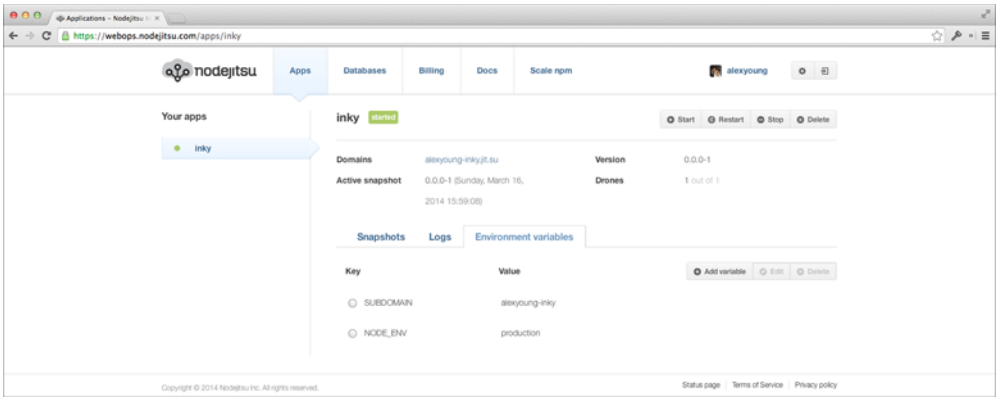


Figure 12.2 The WebOps management interface

founded in 2007. It has since been acquired by Salesforce.com, and uses a virtualized solution based on Ubuntu servers. To use Heroku, you'll need to sign up at heroku.com. It's easy to create a free account, and you can even run production applications on the free tier. Essential features like domain aliases and SSL are paid, so it doesn't take many requirements to hit around $20 a month, but if you don't mind using a Heroku subdomain, you can keep things running for free.

Once you've created an account, you'll need to install the Heroku Toolbelt from toolbelt.heroku.com. There are installers for Linux, Mac OS X, and Windows. Once you've installed it, you'll have a command-line client called `heroku` that can be used to create and manage applications. Before you can use it, you'll have to sign in; `heroku login` can be used to do this, and functions in much the same way as Nodejitsu's `jitsu` command. You only need to log in once because it stores a token that will be used for subsequent requests. Figure 12.3 shows what this should look like.

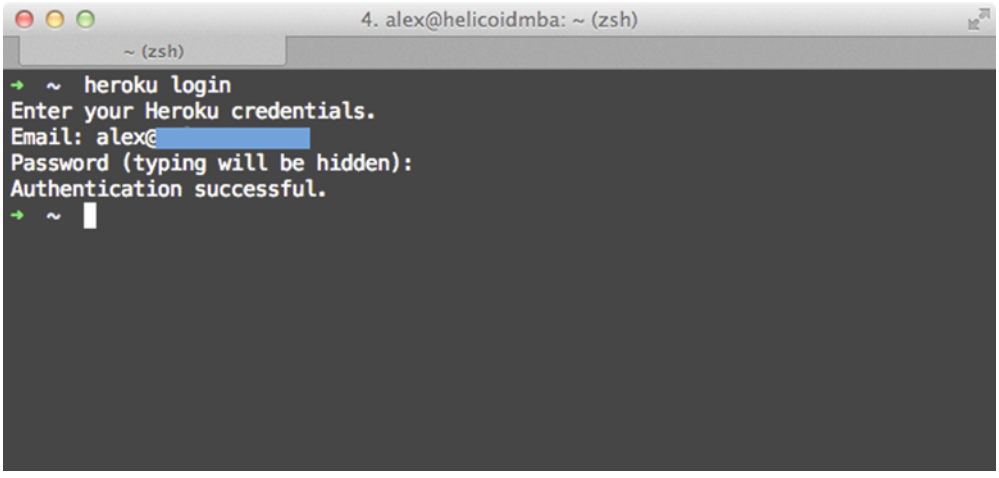


Figure 12.3 Signing in with Heroku

The next step with a Heroku deploy is to prepare your repository. You'll need to `git init` and commit your project. If you're using our code samples and have checked them out of Git, then you should copy the specific project that you want to deploy out of our working tree. The full steps are as follows:

```
1 git init
2 git add .
3 git commit -m 'Create new project'
4 heroku create
5 git push heroku master
```

The `heroku create` command sets up a remote repository called `heroku`, and the first `git push` to it will trigger the creation of a temporary `herokuapp.com` subdomain.

If your application can be started with `npm start`, it should just work. If not, you might need to add a file called Procfile to your application that contains `web: node yourapp.js`. This file lists the processes that your application needs to run—it could include background workers as well.

If you're using an Express application that expects `NODE_ENV` to be set, then you'll need to do this manually with Heroku. The command is just `heroku config:set NODE_ENV=production`, but notice that this is automatic with Nodejitsu.

The last PaaS provider we'll discuss is Windows Azure. Microsoft's Azure platform can be used entirely through the web interface, but there's also a command-line interface that you can install with `npm install -g azure-cli`. Figure 12.4 shows what the command-line tool looks like.

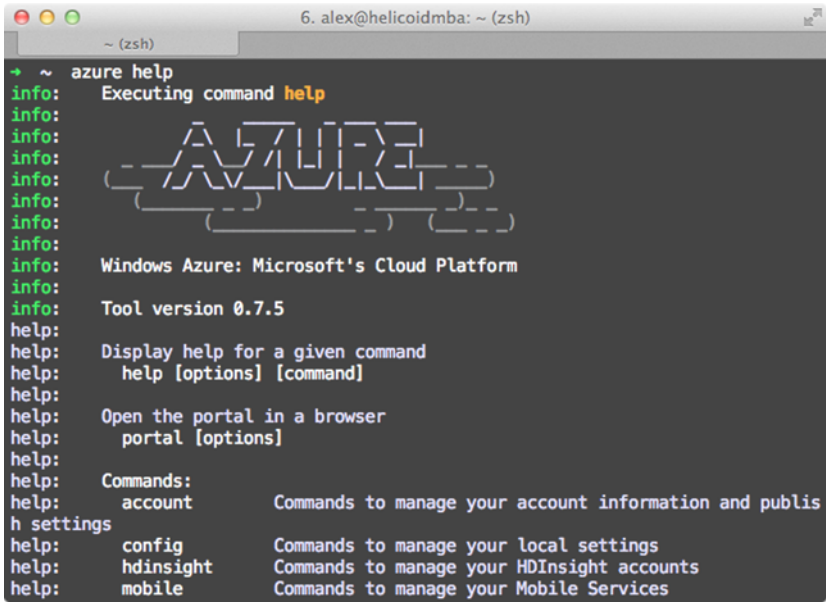


Figure 12.4 The Azure CLI tool

Azure also has an SDK that you can download for Linux, Mac OS X, and Windows. The downloads are available at www.windowsazure.com/en-us/downloads/.

To start using Azure, you'll need to sign in to www.windowsazure.com with a Microsoft account. This is the same account that you can use for other Microsoft services, so if you already have an email account with Microsoft, you should be able to sign in. Azure's registration process has extra security steps: a credit card and phone number are used to validate your account, so it's a bit more tedious than Heroku or Nodejitsu.

Once you've created your Windows Azure account, you'll want to go to the Portal page. Next go to *Compute*, *Web Site*, and then *Quick Create*. Just keep in mind that you're creating a "Web Site" and you should be fine—Microsoft supports a wide range of services that are partly tailored to .NET development with their existing tools like Visual Studio, so it can be bewildering for Mac and Unix developers.

Once your application has been created, you'll need to tie it to a source control repository. Don't worry, you can use GitHub! Before we go any further, check that you're looking at a page like the one in figure 12.5.

Cloud configuration

PaaS providers all seem to have their own approaches to application configuration. You can, of course, keep configuration settings in your code, or JSON files, but there are times when it's useful to store them outside of your repository.

For example, we build open source web applications that we also run on Heroku, so we keep our database passwords outside of our open source repository and use `heroku config:set` instead.

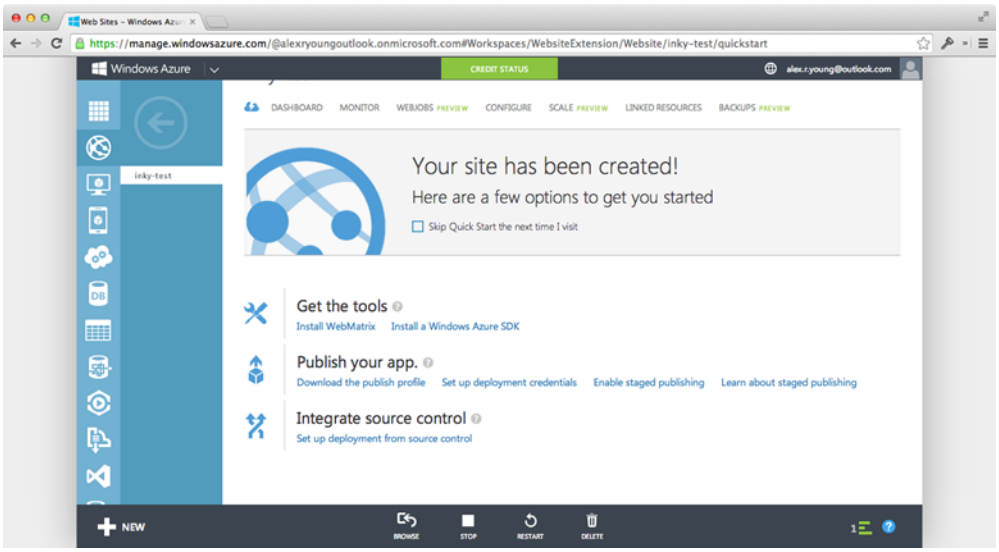


Figure 12.5 Azure's web interface after creating a website

Click your application's name, select *Set up deployment from source control*, and then look for *site URL* on the right side. From here you'll be able to choose from a huge range of repositories and service providers, but we tested our application with GitHub. Azure fetched the code and set up a Node application—it was the same Express code that we used for Heroku (listings/production/inky), and worked the first time.

Table 12.1 shows how to get and set configuration values on each of the cloud providers we've discussed here.

Table 12.1 Setting environmental variables

Provider	Set	Remove	List
Nodejitsu	`jitsu env set name value`	`jitsu env delete name`	`jitsu env list`
Heroku	`heroku config:set name=value`	`heroku config:unset name`	`heroku config`
Azure	`azure site appsetting add name=value`	`azure site appsetting delete name`	`azure site appsetting list`

Although Azure's registration requirements might seem less convenient than Heroku and Nodejitsu, it does have several benefits: if you're working with .NET, then you can use your existing tools. Also, Microsoft's documentation is excellent, and includes guides on setup and deploying for Linux and Mac OS X (http://www.windowsazure.com/en-us/documentation/articles/web-sites-nodejs-develop-deploy-mac/).

Your own servers, rented servers, or cheap virtual hosts all have their own advantages. If you want complete control over your server, or if your business already has its own servers or data centers, then read on to learn how to deploy Node to your own servers.

TECHNIQUE 97 Using Node with Apache and nginx

Deploying Node to private servers running Apache or nginx is entirely possible, and recommended for certain situations. This technique demonstrates how to run a Node program behind Apache and nginx.

Problem

You want to run a Node web application on your own server.

Solution

Use Apache or nginx proxying and a service supervisor like runit.

Discussion

While PaaS solutions are easy to use, there are times when you have to use dedicated hardware, or virtual machines that you have full control over. Larger businesses often have investments in their own data centers, so it doesn't make sense to switch to an external service provider.

Virtualization has transformed web hosting. Linux virtual machines have been a key solution for hosting web applications for several years, and services like Amazon Elastic Compute Cloud make it easy to create and destroy services on demand.

It's therefore likely that at some point you'll be faced with deploying Node applications to servers that need configuration and maintenance. If you're already experienced with basic systems administration tasks, then you can reuse your existing skills and software. Otherwise, you'll have to become familiar with web server daemons and the tools used to keep Node programs running and recovering from errors.

This technique presents examples for Apache and nginx. They're both web servers, but their configuration formats are very different, and they're built in different ways. Figure 12.6 shows the basic server architecture that we'll create in this section.

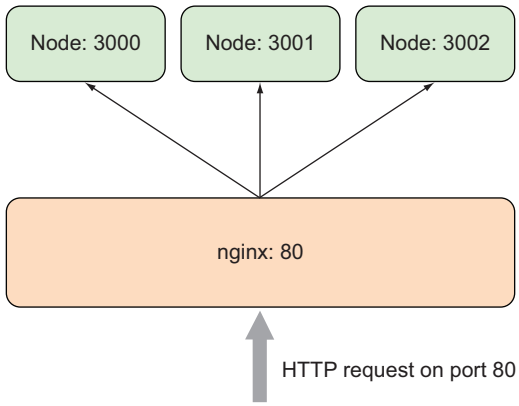


Figure 12.6 A Node program running alongside Apache or nginx

It's not actually necessary to run a web server—there are ways to make Node programs safely access port 80. But we assume that you're deploying to a server that has existing websites. Also, some people prefer to serve static assets from Apache or nginx.

The same technique is used for both servers: proxying. The following listing shows how to do this with Apache.

Listing 12.1 Proxying requests to a Node application with Apache

```
ProxyPass / http://localhost:3000/
LoadModule proxy_module /lib/apache2/modules/mod_proxy.so
LoadModule proxy_http_module /lib/apache2/modules/mod_proxy_http.so
```

1 Proxy requests starting at / to localhost:3000

Load proxy module 2

3 Load HTTP proxy module

The directives in listing 12.1 should be added to your Apache configuration file. To find the right file, type `apache2 -V` on your server, and look for the `HTTPD_ROOT` and `SERVER_CONFIG_FILE` values—joining them will give you the right path and file. It's likely that you won't want to redirect *all* requests to your Node application, so you can add the proxy settings to a `VirtualHost` block.

With these three lines, requests to / will now be proxied to a process listening on port 3000 1. In this case, the process is assumed to be a Node program that you've run with `node server.js` or `npm start`, but it could technically be any HTTP server. The `LoadModule` directives tell Apache to use the proxy 2 and HTTP proxying 3 modules.

If you forget to start the Node process, or quit it, then Apache will return a 503 error. To avoid errors like this, you need a way to keep the Node process running, and to also run it when the server boots. One way to do this is with runit (http://smarden.org/runit/).

If you're using Debian or Ubuntu, you can install runit with `apt-get install runit`. Once it's ready, create a shell script that can start your Node process. First, create a directory for your project: sudo mkdir /etc/service/nodeapp. Next, create a file that will be used for the script: sudo touch /etc/service/nodeapp/run. Then edit the file to make it look like the next listing.

Listing 12.2 Running a program with `runit`

```
#!/bin/sh
export PATH=$PATH:/home/vagrant/.nvm/v0.10.26/bin
cd /home/vagrant/inky
exec npm start
```

1 Set PATH so the shell can find your Node installation

2 Change directory to location of your Node project

Our server was using nvm (https://github.com/creationix/nvm) to manage the installed versions of Node, so we added its location to `$PATH` 1; otherwise the shell couldn't find where `node` and npm were installed. You may have to modify this based on the output of `which node`, or remove it entirely. The last two lines 2 just change the directory to the location of your Node project, and then start it with `npm start`.

The application can be started with `sudo sv start /etc/service/nodeapp` and stopped with `sudo sv stop /etc/service/nodeapp`. Once the Node process is running, you can test it by killing it, and then checking to see that it automatically gets restarted by runit.

Now that you know how Apache handles proxies, and how to keep a process running, let's look at nginx. Nginx is often used as a web server, but it's technically a reverse proxy server that supports HTTP, HTTPS, and email. To make nginx proxy connections to Node applications, you can use the `Proxy` module, which uses a `proxy_pass` directive in a way similar to Apache.

Listing 12.3 has the settings needed by nginx. Like Apache, you could also put the `server` block in a virtual host file.

Listing 12.3 Proxying requests to a Node application with nginx

```
http {
  server {
    listen 80;

    location / {
      proxy_pass http://localhost:3000;
      proxy_http_version 1.1;
    }
  }
}
```

1 This proxies to port 3000, so you can change it to other ports if you use multiple applications.

If you have multiple applications on the same server, then you can use a different port, but we've used 3000 here 1. This example is basically the same as Apache—you tell the server what location to proxy, and then the port. And of course, this example could be combined with runit.

If you don't want to run Apache or nginx, you can run Node web applications without a web server. Read on to learn how to do this using firewall rules and other techniques.

TECHNIQUE 98 Safely running Node on port 80

You can still run Node without a web server daemon like Apache. To do this, you basically need to forward the external port 80 to an internal, unprivileged port. This technique presents some ways to do this in Linux.

Problem

You don't want to use Apache or nginx.

Solution

Use firewall rules to redirect port 80 to another, unprivileged port.

Discussion

In most operating systems, binding to port 80 requires special privileges. That means that if you try to use `app.listen(80)` instead of port 3000 as we've used in most of our examples, you'll see `Error: listen EACCES`. This happens because your current user account doesn't have permission to bind to port 80.

You could get around this restriction by running `sudo npm start`, but this is dangerous. Ideally you want your Node program to run as a nonroot user.

In Linux, traffic can be redirected from port 80 to a higher port number by using `iptables`. Linux uses `iptables` to manage firewall rules, so you just need a rule that maps from port 80 to 3000:

```
iptables -t nat -I PREROUTING -p tcp --dport\
    80 -j REDIRECT --to-port 3000
```

To make this change permanent, you'll need to save the rules to a file that gets run whenever the network interface is set up. The general approach is to save the rules to a file, like /etc/iptables.up.rules, and then edit /etc/network/interfaces to use it:

```
auto eth0
iface eth0 inet dhcp
  pre-up iptables-restore < /etc/iptables.up.rules
  post-down iptables-restore < /etc/iptables.down.rules
```

This is highly dependent on your operating system; these rules are adapted from Debian and Ubuntu's documentation, but it may be different in other Linux distributions.

One downside of this technique is that it maps traffic to *any* process that's listening to that port. An alternative solution is to grant the Node binary extra capabilities. You can do this by installing `libcap2`.

In Debian and Ubuntu, you can use `sudo apt-get install libcap2-bin`. Then you just need to grant the Node binary the capabilities for accessing privileged ports:

```
sudo setcap cap_net_bind_service=+ep /usr/local/bin/node
```

You may need to change the path to Node—check the output of `which node` if you're not sure where it is. The downside of using capabilities for this is that now the `node` binary can bind to all ports from 1–1024, so it's not as specific as restricting it to port 80.

Once you've applied a capability to a binary, it will be fixed until the file changes. That means that you'll need to run this command again if you upgrade Node.

Now that your application is running on a server, you'll want to ensure that it runs forever. There are many different ways to do this; the next technique outlines runit and the `forever` module.

TECHNIQUE 99 Keeping Node processes running

Programs inevitably crash, and it's unfortunate when this happens. What matters is how well you handle failure—users should be informed, and programs should recover elegantly. This technique is all about keeping Node programs running, no matter what.

Problem

Your program crashed in the middle of the night, and customers were unable to use the service until you restarted it.

Solution

Use a process monitor to automatically restart the Node program.

Discussion

There are two main ways to keep a Node program running: service supervision or a Node program that manages other Node programs. The first method is a generic, operating system–specific technique. You've already seen runit in technique 97. Runit supports service supervision, which means it detects when a process stops running and tries to restart it.

Another daemon manager is Upstart (http://upstart.ubuntu.com/). You may have seen Upstart if you use Ubuntu. To use it, you'll need a configuration file that describes how the Node program is managed. Listing 12.4 contains an example that you can modify for your server—it should be saved in /etc/init/nodeapp.conf, where `nodeapp` is the name of your application.

Listing 12.4 Managing a Node program with Upstart

```
#!upstart
description "ExampleApp"
author      "alex"

env PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

respawn
start on runlevel [23]

script
    export NODE_ENV=production
    exec /usr/bin/node /apps/example/app.js 2>&1 >> /var/log/node.log
end script
```

1 You can change the PATH if it's required by your application.

2 This causes the application to be started on run levels 2 and 3.

3 The command that runs the application.

This configuration file tells Upstart to respawn the application (http://upstart.ubuntu.com/wiki/Stanzas#respawn) if it dies for any reason. It sets up a `PATH` 1

that's similar to the one you'll see in your terminal if you type `echo $PATH`. Then it states the program should be run on run levels 2 and 3 (2)—run level 2 is usually when networking daemons are started.

> **RUN LEVELS** Unix systems handle run levels differently depending on the vendor. The Linux Standard Base specification describes run level 2 as multi-user mode, and 3 as multi-user mode with networking. In Debian, 2–5 are grouped as multi-user mode with console logins and the display manager. However, Ubuntu treats run level 2 as graphical multi-user with networking, so you should check how your system implements run levels before using Upstart.

The Upstart `script` stanza allows you to include a short script, so this means you can do things like set `NODE_ENV` to `production`. The application itself is launched with the `exec` instruction. We've included some logging support by redirecting standard out and standard error to a log file (3).

Upstart can be more work to set up than runit, but we've used it in production for three years now without any issues. Both are easier to set up and maintain than traditional stop/start init scripts, but there's another technique you can use: Node programs that monitor other Node programs.

Node process managers work by using a small program that ensures another program runs continuously. This program is simple and therefore less likely to crash than a more complex web application. One of the most popular modules for this is `forever` (https://www.npmjs.org/package/forever), which can be used as a command-line program or programmatically.

Most people use it through the command-line interface. The basic usage is `forever start app.js`, where app.js is your web application. It has lots of options beyond this, though: it can manage log files and even wrap your program so it behaves like a daemon.

To start your program as a daemon, use the following options:

```
forever start -l forever.log -o out.log -e err.log app.js
```

This will start app.js, creating some additional files: one to store the current PID of the active process, a log file, and an error log file. Once the program is running, you can stop it gracefully like this:

```
forever stop app.js
```

Forever can be used with any Node program, but it's generally seen as a tool for keeping web applications running for a long time. The command-line interface makes it easy to use alongside other Unix programs.

Deploying applications that use WebSockets can bring a set of unique requirements. It can be more difficult with PaaS providers, because they can kill requests that last for more than a certain number of seconds. If you're using WebSockets, look through the next technique to make sure your setup will work in production.

TECHNIQUE 100 Using WebSockets in production

Node is great for WebSockets—the same process can serve both standard HTTP requests and the newer WebSocket protocol. But how exactly do you deploy programs that use WebSockets in production? Read on to find out how to do this with web servers and cloud providers.

Problem

You want to use WebSockets in production.

Solution

Make sure the service provider or proxy you're using supports HTTP `Upgrade` headers.

Discussion

WebSockets are amazing, but are still treated almost like second-class citizens by hosting providers. Nodejitsu was the first PaaS provider to support WebSockets, and it uses `node-http-proxy` (https://github.com/nodejitsu/node-http-proxy) to do this. Almost all solutions involve a proxy. To understand why, you need to look at how WebSockets work.

HTTP is essentially a stateless protocol, which means all interactions between a server and a client can be modeled with requests and responses that hold all of the required state. This level of encapsulation has led to the design of modern client/server web applications.

The downside of this is that the underlying protocol doesn't support long-running full-duplex connections. There's a wide class of applications that are built on TCP connections of this type; video streaming and conferencing, real-time messaging, and games are prominent examples. As web browsers have evolved to support richer, more sophisticated applications, we're naturally left trying to simulate these types of applications using HTTP.

The WebSocket protocol was developed to support long-lived TCP-like connections. It works by using a standard HTTP handshake where the client establishes whether the server supports WebSockets. The mechanism for this is a new header called `Upgrade`. As HTTP clients and servers are typically bombarded with a variety of nonstandard headers, servers that don't support `Upgrade` should be fine—the client will just have to fall back to old-fashioned HTTP polling.

Because servers have to handle WebSocket connections so differently, it makes sense to effectively run two servers. In a Node program, we typically have an `http.listen` for our standard HTTP requests, and another "internal" WebSocket server.

In technique 97, you saw how to use nginx with Node. The example used proxies to pass requests from nginx to your Node process, which meant the Node process could bind to a different port to 80. By using the same technique, you can make nginx support WebSockets. A typical `nginx.conf` would look like the next listing.

Listing 12.5 Adding WebSocket support to nginx

```
http {
  server {
    listen 80;
```

```
    server_name example.com;

    location / {
      proxy_pass http://localhost:3000;
      proxy_http_version 1.1;
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection 'upgrade';
      proxy_set_header Host $host;
      proxy_cache_bypass $http_upgrade;
    }
  }
}
```

1 Support Upgrade header

Adding `proxy_http_version 1.1` and `proxy_set_header Upgrade` 1 enables nginx to filter WebSocket requests through to your Node process. This example will also skip caching for WebSocket requests.

Since we mentioned Nodejitsu supports WebSockets, what about Heroku? Well, you currently need to enable it as an add-on, which means you need to run a `heroku` command:

```
heroku labs:enable websockets
```

Heroku's web servers usually kill requests that take longer than around 75 seconds, but enabling this add-on means requests that originate with an `Upgrade` header should keep running for as long as the network allows.

There are times when you might not be able to use WebSockets easily. One example is older versions of Apache, where the proxy module doesn't support them. In cases like this, it can be better to use a proxy server that runs *before* everything else.

HAProxy (http://haproxy.1wt.eu/) is a flexible proxy server. The usage is similar to nginx, and it's also event-based, so it has been widely adopted in the Node community. If you're using an old version of Apache, you can proxy web requests to Apache or Node, depending on various options like URL or headers.

If you want to install HAProxy in Debian or Ubuntu, you can do so with `sudo apt-get install haproxy`. Once it's set up, you'll need to edit /etc/default/haproxy and set `ENABLED=1`—this is just because it ships with a default configuration, so it's disabled by default. Listing 12.6 is a sample configuration that's capable of routing requests to a Node web application that runs on port 3000, but will be accessible using port 80 externally.

Listing 12.6 Using HAProxy with a Node application

```
frontend http-in
  mode http
  bind *:80
  timeout client 999s
  default_backend node_backend

backend node_backend
  mode http
```

1 Allow WebSocket connections a very long time to live.

All HTTP requests will be routed to your Node application.

```
  timeout server 86400000
  timeout connect 5000
  server io_test localhost:3000
```

This should work with WebSockets, and we've used a long timeout so HAProxy doesn't close WebSockets connections, which are typically long-lived ❶. If you run a Node program that listens on port 3000, then after restarting HAProxy with `sudo /etc/init.d/haproxy restart`, your application should be accessible on port 80.

You can use table 12.2 to find the web server that's right for your application.

Table 12.2 Comparing server options

Server	Features	Best for
Apache	■ Fast asset serving ■ Works well with established web platforms (PHP, Ruby) ■ Lots of modules for things like proxying, URL rewriting ■ Virtual hosts	May already be on servers
nginx	■ Event-based architecture, very fast ■ Easy to configure ■ Proxy module works well with Node and WebSockets ■ Virtual hosts	Hosting Node applications when you also want to host static websites, but don't yet have Apache or a legacy server set up
HAProxy	■ Event-based and fast ■ Can route to other web servers on the same machine ■ Works well with WebSockets.	Scaling up to a cluster for high-traffic sites, or complex heterogeneous setups
Native Node proxy	■ Reuse your Node programming knowledge ■ Flexible	Useful if you want to scale and have a team with excellent Node skills

Which server is right for me?

This chapter doesn't cover *every* server choice out there—we've mainly focused on Apache and nginx for Unix servers. Even so, it can be difficult to pick between these options. We've included table 12.2 so you can quickly compare each option.

Your HAProxy setup can be made aware of multiple "back ends" by naming them with the `backend` instruction. In listing 12.7 we only have one—`node_backend` . It would be possible to also run Apache, and route certain requests to it based on the domain name:

```
frontend http-in
  mode http
  bind *:80
  acl static_assets hdr_end(host) -i static.manning.com
```

```
backend static_assets
  mode http
  server www_static localhost:8080
```

This works well if you have an existing set of Apache virtual hosts—perhaps serving things like static assets, blogs, and websites—and you want to add Node to the same server. Apache can be set up to listen on a different port so HAProxy can sit in front of it, and then route requests to Express on port 3000 and the existing Apache sites on port 8080. Apache allows you to change the port by using the `Listen 8080` directive.

You can use the same `acl` option to route WebSockets based on URL. Let's say you've mounted your WebSocket server on `/chat` in your Node application. You could have a specific instance of your server that just handles WebSockets, and route conditionally using HAProxy by using `path_beg`. The following listing shows how this works.

Listing 12.7 Using HAProxy with WebSockets

```
frontend http-in
  mode http
  bind *:80
  acl is_websocket hdr(Upgrade) -i WebSocket

  acl is_websocket path_beg -i /chat

  use_backend ws if is_websocket
  default_backend node_backend

backend node_backend
  mode http
  server www_static localhost:3000

backend ws
  timeout server 600s
  server ws1 localhost:3001
```

1 Check WebSocket header

2 Check if path for WebSockets was used

HAProxy can match requests based on lots of parameters. Here we've used `hdr(Upgrade) -i WebSocket` to test if an `Upgrade` header has been used 1. As you've already seen, that denotes a WebSocket handshake.

By using `path_beg` and marking matching routes with `acl is_websocket` 2, you can now route requests based on the prefix expression `if is_websocket`.

All of these HAProxy options can be combined to route requests to your Node application, Apache server, and WebSocket-specific Node server. That means you can run your WebSockets off an entirely different process, or even another internal web server. HAProxy is a great choice for scaling up Node programs—you could run multiple instances of your application on multiple servers.

HAProxy provides a `weight` option that allows you to implement *round-robin* load balancing by adding `balance roundrobin` to a `backend`.

You can initially deploy your application without nginx or HAProxy in front of it, but when you're ready, you can scale up by using a proxy. If you don't have performance

issues right now, then it's worth just being aware that proxies can do things like route WebSockets to different servers and handle round-robin load balancing. If you already have a server using Apache 2.2.x that isn't compatible with proxying WebSockets, then you can drop HAProxy in front of Apache.

If you're using HAProxy, you'll still have to manage your Node processes with a monitoring daemon like runit or Upstart, but it has proven to be an incredibly flexible solution.

Another approach that we haven't discussed yet is to put your Node applications behind a lightweight Node program that acts as a proxy itself. This is actually used behind the scenes by PaaS providers like Nodejitsu.

Selecting the right server architecture is just the first step to successfully deploying a Node application. You should also consider performance and scalability. The next three techniques include advice on caching and running clusters of Node programs.

12.2 Caching and scaling

This section is mainly about running multiple copies of Node applications at once, but we've also included a technique to give you details on caching. If you can make the client do more work, then why not?

TECHNIQUE 101 HTTP caching

Even though Node is known for high-performance web applications, there are ways you can speed things up. Caching is the major technique, and you should consider caching before deploying your application. This technique introduces the concepts behind HTTP caching.

Problem

You want to reduce how long it takes to make requests to your application.

Solution

Check to ensure that you're using HTTP caching correctly.

Discussion

Modern web applications can be huge: image assets, fonts, CSS, JavaScript, and HTML all add up to a formidable payload that's spread across several HTTP requests. Even with the best minimizers and compression, downloads can still run into megabytes. To avoid requiring users to wait for every action they perform on your site, the best strategy can be to remove the need to download anything at all.

Browsers cache content locally, and can look at the cache to determine if a resource needs to be downloaded. This process is controlled by *HTTP cache headers* and conditional requests. In this technique we'll introduce cache headers and explain how they work, so when you watch your application serving responses in a debugging tool like WebKit Inspector, you'll know what caching headers to expect.

The main two headers are `Cache-Control` and `Expires`. The `Cache-Control` header allows the server to specify a directive that controls how a resource is cached. The basic directives are as follows:

- *public*—Allow caching in the browser and any intermediate proxies between the browser and server.
- *private*—Only allow the browser to cache the resource.
- *no-store*—Don't cache the resource (but some clients still cache under certain conditions).

For a full list of `Cache-Control` directives, refer to the Hypertext Transfer Protocol 1.1 specification (http://www.w3.org/Protocols/rfc2616/rfc2616.html).

The `Expires` header tells the browser when to replace the local resource. The date should be in the RFC 1123 format: `Fri, 03 Apr 2014 19:06 BST`. The HTTP/1.1 specification notes that dates over a year shouldn't be used, so don't set dates too far into the future because the behavior is undefined.

These two headers allow the server to tell clients *when* a resource should be cached. Most Node frameworks like Express will set these headers for you—the static serving middleware that's part of Connect, for example, will set `maxAge` to `0` to indicate cache revalidation should occur. If you watch the Network console in your browser's debugging tools, you should see Express serving static assets with `Cache-Control: public, max-age=0`, and a `Last-Modified` dates based on the file date.

Connect's `static` middleware, which is found in the `send` module, does this by using `stat.mtime.toUTCString` to get the date of the last file modification. The browser will make a standard HTTP `GET` request for the resource with two additional request headers: `If-Modified-Since` and `If-None-Match`. Connect will then check `If-Modified-Since` against the file modification date, and respond with an HTTP 304, depending on the modification date. A 304 response like this will have no body, so the browser can conditionally use local content instead of downloading the resource again.

Figure 12.7 shows a high-level overview of HTTP caching, from the browser's perspective.

Conditional caching is great for large assets that may change, like images, because it's much cheaper to make a `GET` request to find out if a resource should be downloaded again. This is known as a *time-based conditional request*. There are also *content-based conditional* requests, where a digest of the resource is used to see if a resource has changed.

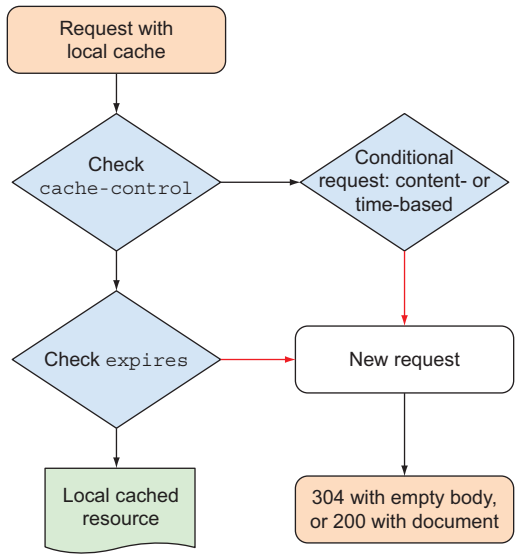


Figure 12.7 Browsers either use the local cache or make a conditional request, based on the previous request's headers.

Content-based conditional requests work using ETags. *ETag* is short for *entity tag*, and allows servers to validate resources in a cache based on their content. Connect's static middleware generates ETags like this:

```
exports.etag = function(stat) {
  return '"' + stat.size + '-' + Number(stat.mtime) + '"';
};
```

Now contrast this to how Express generates ETags for dynamic content—this is usually content sent with `res.send`, like a JavaScript object or a string:

```
exports.etag = function(body){
  return '"' + crc32.signed(body) + '"';
};
```

The first example uses the file modification time and size to create a hash. The second uses a hashing function based on the content. Both techniques send the browser tags that are based on the content, but they've been optimized for performance based on the resource type.

There's pressure on developers of static servers to make them as fast as possible. If you were to use Node's built-in `http` module, you'd have to take all of these caching headers into account, and then optimize things like ETag generation. That's why it's advisable to use a module like Express—it'll handle the details of the required headers based on sensible default behavior, so you can focus on developing your application.

Caching is an elegant way of improving performance because it effectively allows you to reduce traffic by making clients do a bit more work. Another option is to use a Node-based HTTP proxy to route between a cluster of processes or servers. Read on to learn how to do this, or skip to technique 103 to see how to use Node's cluster module to manage multiple Node processes.

TECHNIQUE 102 Using a Node proxy for routing and scaling

Local development is simple because you generally run one Node application at a time. But a production server can host multiple applications, and run the same application on multiple CPU cores to improve performance. So far we've talked about web and proxy servers, but this technique focuses on pure Node servers.

Problem

You want to use a pure Node solution to host multiple applications, or scale an application.

Solution

Use a proxy server module like Nodejitsu's `http-proxy`.

Discussion

This technique demonstrates how to use Node programs to route traffic. It's similar to the proxy server examples in technique 100, so you can reapply these ideas to HAProxy or nginx. But there are times when it might be easier to express routing logic in code rather than using settings files.

Also, as you've seen before in this book, Node programs run as a single process, which doesn't usually take advantage of a modern server that may have multiple CPUs and CPU cores. Therefore, you can use the techniques here to route traffic based on your production needs, but also to run multiple instances of your application so it can better take advantage of your server's resources, reducing response latency and hopefully keeping your customers happy.

Nodejitsu's `http-proxy` (https://www.npmjs.org/package/http-proxy) is a lightweight wrapper around Node's built-in `http` core module that makes it easier to define proxies with code. The basic usage should be familiar to you if you've followed our chapter on Node web development. The following listing is a simple proxy that redirects traffic to another port.

Listing 12.8 Redirecting traffic to another port with `http-proxy`

```
var httpProxy = require('http-proxy');
var proxy = httpProxy.createProxyServer({
  target: 'http://localhost:3000'
});

proxy.on('error', function(err) {
  console.error('Error:', err);
});

proxy.listen(9000);
```

1 Redirect traffic to port 3000

2 Catch errors and log them

3 Set this server to listen on port 9000

This example redirects traffic to port 3000 by using `http-proxy`'s `target` option 1. This module is event-based, so errors can be handled by setting up an error listener 2. The proxy server itself is set to listen on port 9000 3, but we've just used that so you can run it easily—port 80 would be used in production.

The options passed to `createProxyServer` can define other routing logic. If `ws: true` is set, then WebSockets will be routed separately. That means you can create a proxy server that routes WebSockets to one application, and standard requests elsewhere. Let's look at that in a more detailed example. The next listing shows you how to route WebSocket requests to a separate application.

Listing 12.9 Routing WebSocket connections separately

```
var http = require('http');
var httpProxy = require('http-proxy');

var proxy = new httpProxy.createProxyServer({
  target: 'http://localhost:3000'
});

var wsProxy = new httpProxy.createProxyServer({
  target: 'http://localhost:3001'
});

var proxyServer = http.createServer(function(req, res) {
```

1 Create another proxy server for WebSockets

```
  proxy.web(req, res);
});

proxyServer.on('upgrade', function(req, socket, head) {
  wsProxy.ws(req, socket, head);
});

proxyServer.listen(9000);
```

Listen for upgrade event; then use WebSocket proxy instead of the standard web request proxy (2)

This example creates two proxy servers: one for web requests and the other for WebSockets (1). The main web-facing server emits `upgrade` events when a WebSocket is initiated, and this is intercepted so requests can be routed elsewhere (2).

This technique can be extended to route traffic according to any rules you like—if you can infer something from a `request` object, you can route traffic accordingly. The same idea can also be used to map traffic to multiple machines. This allows you to create a cluster of servers, which can help you scale up an application. The following listing could be used to proxy to several servers.

Listing 12.10 Scaling using multiple instances of a server

```
var http = require('http');
var httpProxy = require('http-proxy');

var targets = [
  { target: 'http://localhost:3000' },
  { target: 'http://localhost:3001' },
  { target: 'http://localhost:3002' }
];

var proxies = targets.map(function(options, i) {
  var proxy = new httpProxy.createProxyServer(options);
  proxy.on('error', function(err) {
    console.error('Proxy error:', err);
    console.error('Server:', i);
  });
  return proxy;
});

var i = 0;
http.createServer(function(req, res) {
  proxies[i].web(req, res);
  i = (i + 1) % proxies.length;
}).listen(9000);
```

Create a proxy for each instance of application (1)

Proxy requests using round-robin (2)

This example uses an array that contains the options for each proxy server, and then creates an instance of proxy server for each one (1). Then all you need to do is create a standard HTTP server and map requests to each server (2). This example uses a basic round-robin implementation—after each request a counter is incremented, so the next request will be mapped to a different server. You could easily take this example and reconfigure it to map to any number of servers.

Mapping requests like this can be useful on a single server with multiple CPUs and CPU cores. If you run your application multiple times and set each instance to listen on a different port, then your operating system should run each Node process on a different CPU core. This example uses `localhost`, but you could use another server, thereby clustering the application across several servers.

In contrast to this technique's use of additional servers for scaling, the next technique uses Node's built-in features to manage multiple copies of the same Node program.

TECHNIQUE 103 Scaling and resiliency with cluster

JavaScript programs are considered *single-threaded*. Whether they actually use a single thread or not is dependent on the platform, but conceptually they execute as a single thread. That means you may have to do additional work to scale your application to take advantage of multiple CPUs and cores.

This technique demonstrates the core module `cluster`, and shows how it relates to scalability, resiliency, and your Node applications.

Problem

You want to improve your application's response time, or increase its resiliency.

Solution

Use the `cluster` module.

Discussion

In technique 102, we mentioned running multiple Node processes behind a proxy. In this technique we'll explain how this works purely on the Node side. You can use the ideas in this technique with or without a proxy server to load balance. Either way, the goal is the same: to make better use of available processor resources.

Figure 12.8 shows a system with two CPUs with four cores each. A Node program is running on the system, but only fully utilizing a single core.

There are reasons why figure 12.8 isn't entirely accurate. Depending on the operating system, the process might be moved around cores, and although it's accurate to say a Node program is a single process, it still uses several threads. Let's say you start up an Express application that uses a MySQL database, static file serving, user sessions, and so on. Even though it will run as a single process, it'll still have eight separate threads.

We're trained to think of Node programs as single-threaded because JavaScript platforms are conceptually single-threaded, but behind the scenes, Node's libraries like `libuv` will use threads to provide asynchronous APIs. That gives us the event-based programming style without having to worry about the complexity of threads.

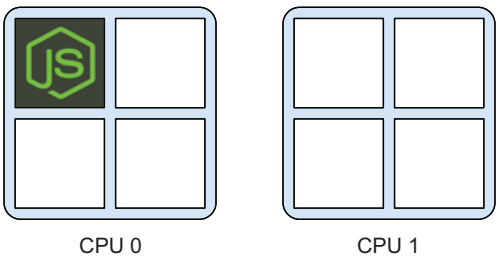


Figure 12.8 A Node process running on a single core

If you're deploying Node applications and want to get more performance out of your multicore, multi-CPU system, then you need to start thinking more about how Node works at this level. If you're running a single application on a multicore system, you want something like the illustration in figure 12.9.

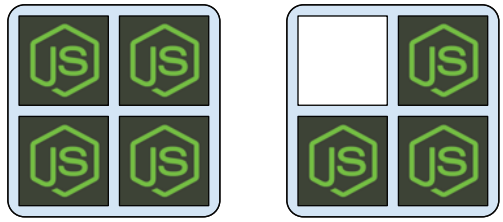


Figure 12.9 Take advantage of more cores by running multiple processes.

Here we're running a Node program on all but one core, the idea being that a core is left free for the system. You can get the number of cores for a system with the `os` core module. On our system, running `require('os').cpus().length` returns 4—that's the number of cores we have, rather than CPUs—Node's API `cpus` method returns an array of objects that represent each core:

```
[{ model: 'Intel(R) Core(TM) i7-4650U CPU @ 1.70GHz',
   speed: 1700,
   times:
   { user: 11299970, nice: 0, sys: 8459650, idle: 93736040, irq: 0 } },
 { model: 'Intel(R) Core(TM) i7-4650U CPU @ 1.70GHz',
   speed: 1700,
   times:
   { user: 5410120, nice: 0, sys: 2514770, idle: 105568320, irq: 0 } },
 { model: 'Intel(R) Core(TM) i7-4650U CPU @ 1.70GHz',
   speed: 1700,
   times:
   { user: 10825170, nice: 0, sys: 6760890, idle: 95907170, irq: 0 } },
 { model: 'Intel(R) Core(TM) i7-4650U CPU @ 1.70GHz',
   speed: 1700,
   times:
   { user: 5431950, nice: 0, sys: 2498340, idle: 105562910, irq: 0 } } ]
```

With this information, we can automatically tailor an application to scale to the target server. Next, we need a way of forking our application so it can run as multiple processes. Let's say you have an Express web application: how do you safely scale it up without completely rewriting it? The main issue is communication: once you start running multiple instances of an application, how does it safely access shared resources like databases? There are platform-agnostic solutions to this that would require a big project rewrite—pub/sub servers, object brokers, distributed systems—but we'll use Node's `cluster` module.

The `cluster` module provides a way of running multiple worker processes that share access to underlying file handles and sockets. That means you can wrap a Node application with a master process that works workers. Workers don't need access to shared state if you're doing things like accessing user sessions in a database; all the workers will have access to the database connection, so you shouldn't need to set up any communication between workers.

Listing 12.11 is a basic example of using clustering with an Express application. We've just included the server.js file that loads the main Express application in app.js. This is our preferred method of structuring Node web applications—the part that sets up the server using `.listen(port)` is in a different file than the application itself. In this case, separating the server and application has the additional benefit of making it easier to add clustering to the project.

Listing 12.11 Clustering a Node web application

```
var app = require('./app');
var cluster = require('cluster');

if (cluster.isMaster) {
  var totalWorkers = require('os').cpus().length - 1;

  console.log('Running %d total workers', totalWorkers);

  for (var i = 0; i < totalWorkers; i += 1) {
    cluster.fork();
  }
} else {
  console.log('Worker PID:', process.pid);
  app.listen(process.env.PORT || 3000);
}
```

1 Load the cluster core module.

2 Determine how many processes should be started.

3 Fork the process to create a worker.

4 Workers will hit this branch, and the PID is displayed.

The basic pattern is to load the `cluster` core module 1, and then determine how many cores should be used 2. The `cluster.isMaster` allows the code to branch if this is the first (or *master*) process, and then fork workers as needed with `cluster.fork` 3.

Each worker will rerun this code, so when a worker hits the `else` branch, the server can run the code particular to the worker 4. In this example workers start listening for HTTP connections, thereby starting the Express application.

There's a full example that includes this code in this book's code samples, which can be found in production/inky-cluster.

If you're a Unix hacker, this should all look suspiciously familiar. The semantics of `fork()` are well known to C programmers. The way it works is whenever the system call `fork()` is used, the current process is cloned. Child processes have access to open files, network connections, and data structures in memory. To avoid performance issues, a system called *copy on write* is used. This allows the same memory locations to be used until a write is attempted, at which point each forked process receives a copy of the original. After the processes are forked, they're isolated.

There's an additional step to properly dealing with clustered applications: worker exit recovery. If one of your workers encounters an error and the process ends, then you'll want to restart it. The cool thing about this is any other active workers can still serve requests, so clustering will not only improve request latency but also potentially uptime as well. The next listing is a modification of listing 12.11, to recover from workers exiting.

Listing 12.12 Recovering from untimely worker death

```
var app = require('./app');
var cluster = require('cluster');

if (cluster.isMaster) {
  var totalWorkers = require('os').cpus().length - 1;

  console.log('Running %d total workers', totalWorkers);

  for (var i = 0; i < totalWorkers; i += 1) {
    cluster.fork();
  }

  cluster.on('exit', function(worker) {
    console.log('Worker %d died', worker.id);
    cluster.fork();
  });
} else {
  console.log('Worker PID:', process.pid);
  app.listen(process.env.PORT || 3000);
}
```

1 cluster module is event-based

2 Fork again after worker dies

The `cluster` module is event-based, so the master can listen for events like `exit` 1, which denotes the worker died. The callback for this event gets a `worker` object, so you can get a limited amount of information about the worker. After that all you need to do is fork again 2, and you'll be back to the full complement of workers.

Recovering from a crash in the master process

You might be wondering what happens when the master process itself dies. Even though the master should be kept simple to make this unlikely, a crash is still of course possible. To minimize downtime, you should still manage your clustered applications with a process manager like the `forever` module or Upstart. Both of these solutions are explored in technique 99.

You can run this example with an Express application, and then use `kill` to force workers to quit. The transcript of such a session should look something like this:

```
Running 3 total workers
    Worker PID: 58733
    Worker PID: 58732
    Worker PID: 58734
    Worker 1 died
    Worker PID: 58737
```

Three workers were running until `kill 58734` was issued, and then a new worker was forked and `58737` started.

Once you've got clustering set up, there's one more thing to do: benchmark. We'll use `ab` (http://httpd.apache.org/docs/2.0/programs/ab.html), the Apache benchmarking tool. It's used like this:

```
ab -n 10000 -c 100 http://localhost:3000/
```

This makes 10,000 requests with 100 concurrent requests at any one time. Using three workers on our system gave 260 requests per second, whereas a single process version resulted in 171 requests per second. The cluster was definitely faster, but is this really working as well as our round-robin example with HAProxy or nginx?

The advantage of the `cluster` module is that you can script it with Node. That means your developers should be able to understand it rather than having to learn how HAProxy or nginx works for load balancing. Load balancing with an additional proxy server doesn't have the same kind of interprocess communication options that `cluster` has—you can use `process.send` and `cluster.workers[id].on('message', fn)` to communicate between workers.

But proxies with dedicated load-balancing features have a wider choice of load-balancing algorithms. Like all things, it would be wise to invest time in testing HAProxy, nginx, and Node's clustering module to see which works best for your application and your team.

Also, dedicated load-balancing servers can proxy requests to multiple servers—you could technically proxy from a central server to multiple Node application servers, each of which uses the `cluster` core module to take advantage of the server's multicore CPU.

With heterogeneous setups like this, you'll need to keep track of what instances of your application are doing. The next section is dedicated to maintaining production Node programs.

12.3 Maintenance

No matter how solid your server architecture is, you're still going to have to maintain your production system. The techniques in this section are all about maintaining your Node program; first, package optimization with npm.

TECHNIQUE 104 Package optimization

This technique is all about npm and how it can make deployments more efficient. If you feel like your module folder might be getting a bit large, then read on for some ideas on how to fix it.

Problem

Your application seems larger than expected when it's released to production.

Solution

Try out some of npm's maintenance features, like `npm prune` and `npm shrinkwrap`.

Discussion

Heroku makes your application's size clear when you deploy: each release displays a *slug size* in megabytes, and the maximum size on Heroku is 300 MB. Slug size is closely related to dependencies, so as your application grows and new dependencies are added, you'll notice that it can increase dramatically.

Even if you're not using Heroku, you should be aware of your application's size. It will impact how quickly you can release new code, and releasing new code should be as fast as possible. When deployment is fast, then releasing bug fixes and new features becomes less of a chore and less risky.

Once you've gone through your dependencies in package.json and weeded out any that aren't necessary, there are some other tricks you can use to reduce your application's size. The `npm prune` command removes packages that are no longer listed in your package.json, but it also applies to the dependencies themselves, so it can sometimes dramatically reduce your application's storage footprint.

You should also consider using `npm prune --production` to remove `devDependencies` from production releases. We've found test frameworks in our production releases that didn't need to be there. If you have `./node_modules` checked into git, then Heroku will run `npm prune` for you, but it doesn't currently run `npm prune --production`.

Why check in ./node_modules?

It might be tempting to add `./node_modules` to `.gitignore`, but don't! When you're working on an application that will be deployed, then you should keep `./node_modules` in your repository. This will help other people to run your application, and make it easier to reproduce your local setup that passes tests and everything else on a production environment.

Do not do this for modules you release through *npm*. Open source libraries should use *npm* to manage dependencies during installation.

Another command you can use to potentially improve deployment is `npm shrinkwrap`. This will create a file called npm-shrinkwrap.json that specifies the exact version of each of your dependencies, but it doesn't stop there—it continues recursively to capture the version of each submodule as well. The npm-shrinkwrap.json file can be checked into your repository, and npm will use it during deployment to get the exact version of each package.

`shrinkwrap` is also useful for collaboration, because it means people can duplicate the modules you've had living on your computer during development. This helps when someone joins a project after you've been working solo for a few months.

Some PaaS providers have features for excluding files from deployment as well. For example, Heroku can accept a .slugignore file, which works like .gitignore—you could create one like this to ignore tests and local seed data:

```
/test
/seed-data
/docs
```

By taking advantage of npm's built-in features, you can create solid and maintainable packages, reduce deployment time, and improve deployment reliability.

Even with a well-configured, scalable, and carefully deployed application, you'll still run into issues. When things go wrong, you need logs. Read on for techniques when dealing with log files and logging services.

TECHNIQUE 105 Logging and logging services

When things break—not if, but when—you'll need logs to uncover what happened. On a typical server, logs are text files. But what about PaaS providers, like Heroku and Nodejitsu? For these platforms you'll need logging services.

Problem

You want to log messages from a Node application on your own server, or on a PaaS provider.

Solution

Either redirect logs to files and use `logrotate`, or use a third-party logging service.

Discussion

In Unix, everything is a file, and that partly dictates the way systems administrators and DevOps experts think about log files. Logs are just files: programs stream data into them, and we stream data out. This kind of setup is convenient for those of us that live in the command line—piping files through commands like `grep`, `sed`, and `awk` makes light work of even gigabyte-sized logs.

Therefore, whatever you do, you'll want to correctly use `console.log` and `console.error`. It also doesn't hurt to be aware of `err.stack`—instances of `Error` in Node get a `stack` property when they're defined, which can be extremely helpful for debugging problems in production. For more on writing logs, take a look at technique 6 in chapter 2.

The benefit of using `console.error` and `console.log` is that you can pipe output to different locations. The following command will redirect data from standard out (`console.log`) to `application.log`, and standard error (`console.error`) to `errors.log`:

```
npm start 1> application.log 2> errors.log
```

All you need to remember is the greater-than symbol redirects output, and using a number specifies the output stream: 1 is standard out, and 2 is standard error.

After a while, your log files will get too large. Fortunately, modern Unix systems usually come with a log rotation package. This will split files up over time and optionally compress them. The `logrotate` package can be installed in Debian or Ubuntu with `apt-get install logrotate`. Once you've installed it, you'll need a configuration

file for each set of log files you want to rotate. The following listing shows an example configuration that you can tailor for your application.

Listing 12.13 `logrotate` configuration

```
"/var/www/nodeapp/logs/application.log"
"/var/www/nodeapp/logs/application.err" {
  daily
  rotate 20
  compress
  copytruncate
}
```

Run every day (1)
Keep 20 files (2)
Compress rotated files (3)
Truncate current log file (4)

After listing the log files you want to rotate, you can list the options you want to use. `logrotate` has many options, and they're documented in `man logrotate`. The first one here, `daily` (1), just states that we want to rotate files every day. The next line makes `logrotate` keep 20 files; after that files will be removed (2). The third option will make sure old log files are compressed so they don't use up too much space (3).

The fourth option, `copytruncate` (4), is more important for an application that uses simple standard I/O-based logging. It makes `logrotate` copy and then truncate the current log file. That means that your application doesn't need to close and reopen standard out—it should just work without any special configuration.

Using standard I/O and `logrotate` works well for a single server and a simple application, but if you're running an application in a cluster, you might find it difficult to manage logging. There are Node modules that are dedicated to logging and provide cluster-specific options. Some people even prefer to use these modules because they generate output in standard log file formats.

Using the `log4node` module (https://github.com/bpaquet/log4node) is similar to using `console.log`, but has features that make it easier for use in a cluster. It creates one log file for all workers, and listens for a USR2 signal to determine when to reopen files. It supports configuration options, including log level and message prefix, so you can keep logs quiet during tests or increase the verbosity for critical production systems.

`winston` (https://github.com/flatiron/winston) is a logging module that supports multiple transports, including Cassandra, which allows you to cluster your log writes. That means that if you have an application that writes millions of log entries an hour, then you can use multiple servers to capture the logs in a more reliable manner.

`winston` supports remote log services, including commercial ones like Papertrail. Papertrail and Loggly (see figure 12.10) are commercial services that you can pipe your logs to, typically using the syslogd protocol. They will also index logs, so searching gigabytes of logs is extremely fast, depending on the query.

A service like Loggly is absolutely critical for Heroku. Heroku only stores the last 5,000 log entries, which can be flooded off within minutes of running a typical application. If you've deployed a Node application to Heroku that uses `console.log`, `log4node`, or `winston`, then you'll be able to redirect your logs just by enabling the add-on.

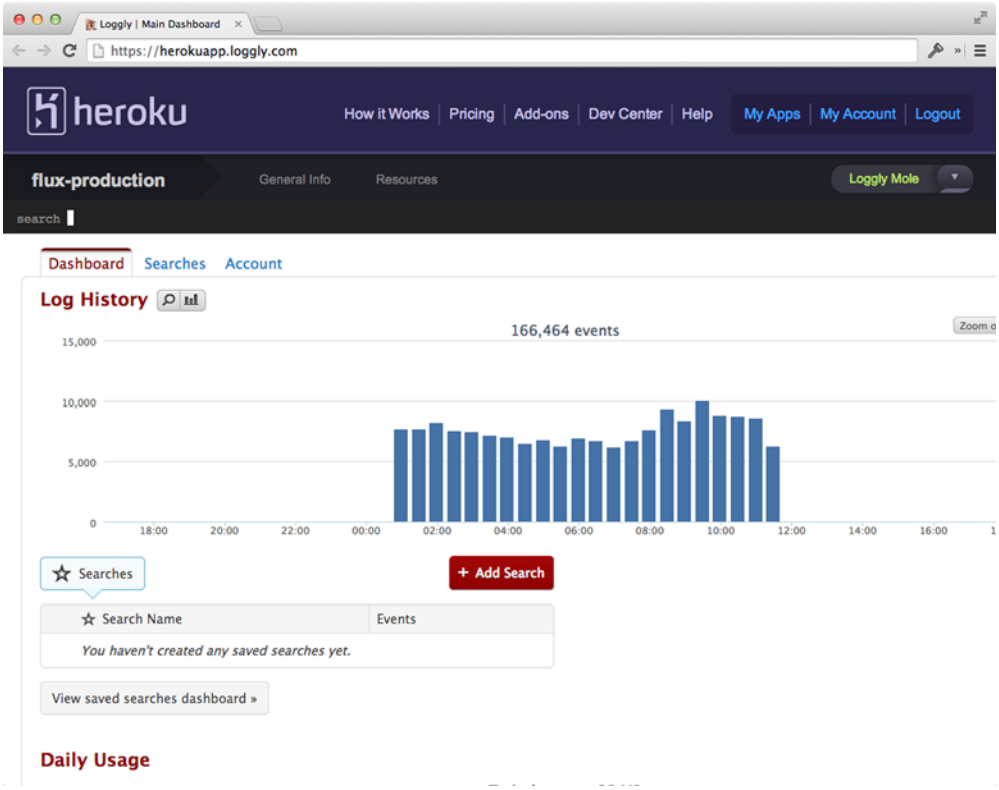


Figure 12.10 Loggly's dashboard

With Heroku, Loggly can be configured by selecting a plan name and running `heroku addons:add Loggly:PlanName` from your project's directory. Typing `heroku addons:open loggly` will open the Loggly web interface, but there's also a link in Heroku's administration panel under *Resources*. Any logging you've done with standard I/O should be sent straight to Loggly.

If you're using `winston`, then there are transports available for Loggly. One is `winston-loggly` (https://github.com/indexzero/winston-loggly), which can be used for easy access to Loggly with non-Heroku services, or your own private servers.

Because Winston transports can be changed by using `winston.add(winston.transports.Loggly, options)`, you don't need to do anything special to support Loggly if you're already using `winston`.

There's a standard for logging that you can use with your applications: The Syslog Protocol (RFC 5424). Syslog message packets have a standard format, so you won't usually generate them by hand. Modules like `winston` typically support syslog, so you can use it with your Node application, but there are two main benefits to using it. The first is that messages have standardized log levels, so filtering logs is easier. Some examples include level 0, known as *Emergency*, and level 4, which is *Warning*. The second is that the

protocol defines how messages are sent over the network, which means you can make your Node application talk to a syslog daemon that runs on a remote server.

Some log services like Loggly and Splunk can act as syslog servers; or, you could run your own daemon on dedicated hardware or a virtual machine. By using a standardized protocol like syslog, you can switch between log providers as your requirements change.

That's the last technique on Node-specific production concerns. The next section outlines some additional issues relating to scaling and resiliency.

12.4 Further notes on scaling and resiliency

In this chapter we've demonstrated how to use proxies and the `cluster` module to scale Node programs. One of the advantages we cited in `cluster`'s favor is easier interprocess communication. If you're running an application on *separate servers*, how can Node processes communicate?

One simple answer might be HTTP—you could build an internal REST API for communication. You could even use WebSockets if messages need faster responses. When we were faced with this problem, we used RabbitMQ (https://www.rabbitmq.com/). This allowed instances of our Node application to message each other using a shared message bus, thereby distributing work throughout a cluster.

The project was a search engine that used Node programs to download and scrape content. Work was classified into spidering, downloading, and scraping. Swarms of Node processes would take work from queues, and then push new jobs back to queues as well.

There are several implementations of RabbitMQ clients on npm—we used amqplib (https://www.npmjs.org/package/amqplib). There are also competitors to RabbitMQ—zeromq (http://zeromq.org/) is a highly focused and simple alternative.

Another option is to use a hosted publish/subscribe service. One example of this is Pusher (http://pusher.com/), which uses WebSockets to help scale applications. The advantage of this approach is that Pusher can message anything, including mobile clients. Rather than restricting messaging to your Node programs, you can create message channels that web, mobile, and even desktop clients can subscribe to.

Finally, if you're using private servers, you'll need to monitor resource usage. StrongLoop (http://strongloop.com/) offers monitoring and clustering tools for Node, and New Relic (New Relic) also now has Node-specific features. New Relic can help you break down where time is being spent in a live application, so you can use it to discover bottlenecks in database access, view rendering, and application logic.

With service providers like Heroku, Nodejitsu, and Microsoft, and the tools provided by StrongLoop and New Relic, running Node software in production has rapidly matured and become entirely feasible.

12.5 Summary

In this chapter you've seen how to run Node on PaaS providers, including Heroku, Nodejitsu, and Windows Azure. You've also learned about the issues of running Node on private servers: safely accessing port 80 (technique 98), and how WebSockets relate to production requirements (technique 100).

No matter how fast your code is, if your application is popular, then you may run into performance issues. In our section on scaling, you've learned all about caching (technique 101), proxies (technique 102), and scaling with `cluster` (technique 103).

To keep your application running solidly, we've included maintenance-related techniques on npm in production (technique 104) and logging (technique 105). Now if anything goes wrong, you should have enough information to solve the problem.

Now you should know how to build Node web applications and release them in a maintainable and scalable state.

Part 3

Writing modules

As we dove deep into Node's core libraries and looked into real-world recipes, we've been building a narrative that leads to the biggest part of the Node ecosystem: community-driven innovation through third-party module development. As the core provides the Legos with which we build, and the recipes provide the tooling and insight to build confidently, what we ultimately build is up to us!

We have one last chapter that will take you through the ins and outs of building a module and contributing it back to the community.

13 Writing modules: Mastering what Node is all about

This chapter covers

- Planning a module
- Setting up a package.json file
- Working with dependencies and semantic versioning
- Adding executable scripts
- Testing out a module
- Publishing modules

The Node package manager (npm) is arguably the *best* package manager any platform has seen to date. npm at its core is a set of tools for installing, managing, and creating Node modules. The barrier to entry is low and uncluttered with ceremony. Things "just work" and work well. If you aren't convinced yet, we hope this chapter will encourage you to take another look.

The subtitle for this chapter is "Mastering what Node is all about." We chose this because user-contributed modules *make up the vast majority of the Node ecosystem.* The

core team decided early on that Node would have a *small* standard library, containing just enough core functionality to build great modules upon. We knew understanding this core functionality was paramount to building modules, so we saved this chapter for the end. In Node, you may find 5 or 10 different implementations for a particular protocol or client, and we're *OK* with that because it allows experimentation to drive innovation in the space.

One thing we've learned through our experimentation is that *smaller modules matter*. Larger modules tend to be hard to maintain and test. Node enables smaller modules to be stuck together simply to solve more and more complex problems.

Node's require system (based on CommonJS; http://wiki.commonjs.org/wiki/Modules/1.1) manages those dependencies in a way that avoids dependency hell. It's perfectly fine for modules to depend on different versions of the same module, as shown in figure 13.1.

In addition to standard dependencies, you can specify development and peer dependencies (more on that later) and have npm keep those in check for you.

DEPENDENCY GRAPHS If you ever want to see a dependency graph for your project, just type `npm ls` at the project root to get a listing.

Another difference that was decided early on in the history of npm was to manage dependencies at a *local level by default* as popularized by the bundler Ruby gem. This bundles modules *inside* your project (sitting in the node_modules folder), making dependency hell a non-issue across multiple projects since there's no globally shared module state.

INSTALLING GLOBAL MODULES You can still install global modules if you want with `npm install -g module-name`, which can be useful when you need a system-wide executable, for instance.

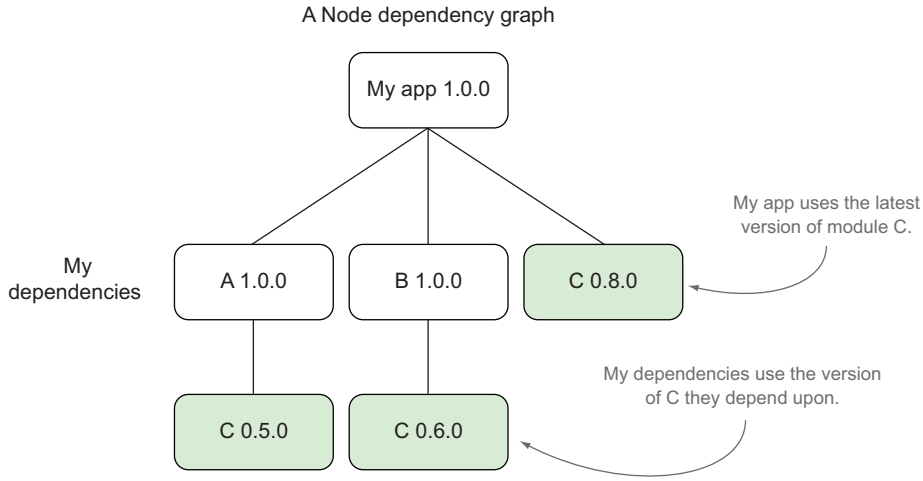


Figure 13.1 Node avoids dependency hell

Hopefully we've whetted your appetite for exploring a range of module-authoring techniques! In this chapter we'll focus on a variety of techniques that center around

- Effectively making the most of the package.json file
- Using npm for various module-authoring tasks
- Best practices for developing modules

Our techniques will follow a logical order from an empty project directory to a completed and published npm module. Although we tried to stuff as many concepts as possible into one module, you may find your module may only need a handful of these steps. When we can't fit a concept into the module, we'll focus on an isolated use case to illustrate the point.

13.1 Brainstorming

What kind of API do we want to build? How should someone consume it? Does it have a clear purpose? These are some of the questions we need to ask as we begin to write a module. In this section we'll walk through researching and proving out a module idea. But first, let's introduce a problem we want to solve, which will provide a context as we progress.

13.1.1 A faster Fibonacci module

One of the most famous Node critiques (although arguably misguided) early on in its history was "Node.js is Cancer" (http://pages.citebite.com/b2x0j8q1megb), where the author argued that a CPU-bound task on a running web server was miserably handled in Node's single-threaded system.

The implementation was a common recursive approach to calculating a Fibonacci sequence (http://en.wikipedia.org/wiki/Fibonacci_number), which could be implemented as follows:

```
function fibonacci (n) {
  if (n === 0) return 0;
  if (n === 1) return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

This line was added to the original implementation, since the original didn't return a proper sequence number for 0.

This implementation is slow in V8, and since proper tail calls don't yet exist in JavaScript, it wouldn't be able to calculate very high numbers due to a stack overflow.

Let's write a module to help rid the world of slow Fibonacci calculations in order to learn about module development from start to finish.

TECHNIQUE 106 Planning for our module

So we want to start writing a module. How should we approach it? Is there anything we can do before we even start writing a line of code? It turns out that planning ahead of time can be extremely helpful and save pain down the road. Let's take a peek on how to do that well.

■ **Problem**

You want to write a module. What steps should you take in planning?

■ **Solution**

Research what already exists, and ensure that your module does just one thing.

■ **Discussion**

It's important to clearly articulate the purpose of your module. If you can't boil it down to one sentence, it may be doing too much. Here's where an important aspect of the Unix philosophy comes in: *make each program do one thing well.*

Surveying the landscape

First, it's good to know what exists already. Has someone else implemented a solution to my problem? Can I contribute there? How did others approach this? A great way to do that is searching on npmjs.org or doing a search from the command line:

```
npm search fibonacci
```

If this is your first time running npm search, it will take a while to update the local cache before you get any results.

Let's look at some of the more interesting results:

```
fibonacci Calculates fibonacci numbers for one or endless iterations.…
  =franklin 2013-05-01 1.2.3 fibonacci math bignum endless

fibonacci-async So, you want to benchmark node.js with fibonacci once…
  =gottox 2012-10-29 0.0.2

fibonacci-native A C++ addon to compute the nth fibonacci number.
  =avianflu 2012-03-21 0.0.0
```

Here we can see the names and descriptions of three different implementations. We also see what version was released last and on what date. It looks like a couple are older and have a lower version number, which may mean the API is in flux or still in progress. But the top result looks pretty mature at version 1.2.3 and has been updated most recently. Let's get more information on that by running the following:

```
npm docs fibonacci
```

The `npm docs` command will load the module's homepage if specified, or the npmjs search result, which looks like figure 13.2.

The npmjs result page helps give you an overall picture for a module. We can see this module depends on the `bignum` module and was updated a year ago, and we can view its readme to get a sense of the API.

Although this module looks pretty good, let's create a module as an experiment to try out some other ideas for handling Fibonacci sequences. In our case, let's create a module where we'll experiment with different implementations and benchmark our results using straight JavaScript with no required `bignum` dependency.

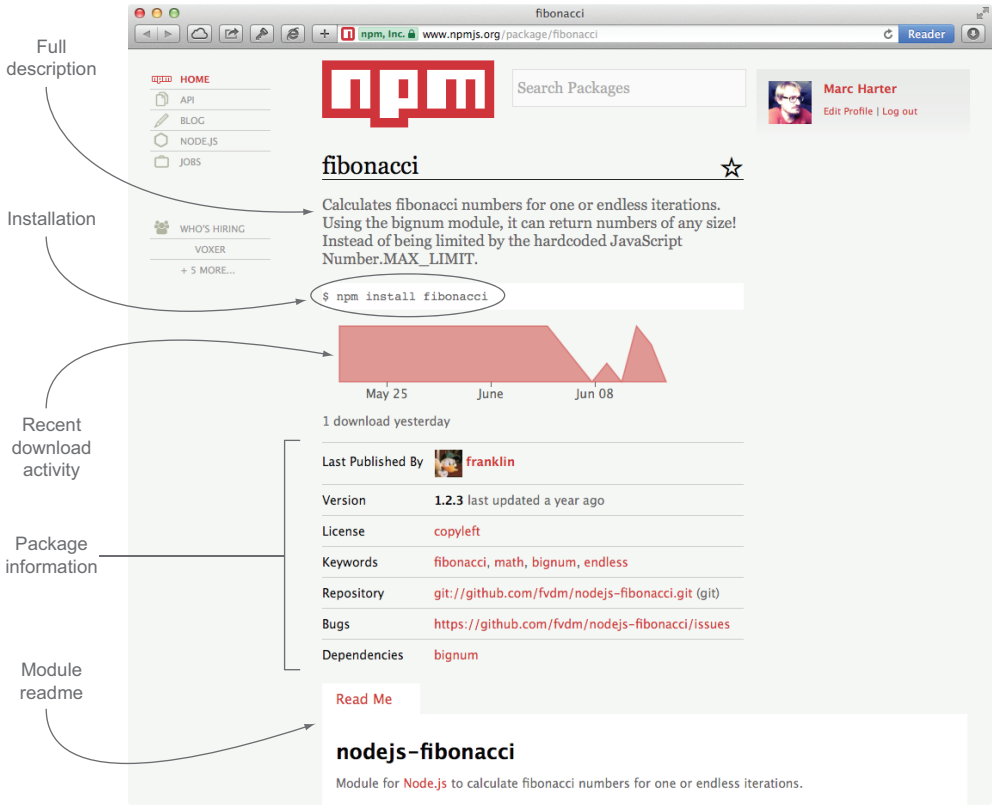


Figure 13.2 npmjs.com package details page

Embrace doing one thing well

A module should be simple and pluggable. In this case, let's try to define our module's purpose in one phrase:

Calculates a Fibonacci number as quickly as possible with only JavaScript

That's a pretty good start: it's clear and succinct. When that concept doesn't ring true anymore, we've blown our scope, and it may be time to write another module that extends this one rather than adding more to it. For this project, adding a web server endpoint that returns the result of this function may be better served in a new module that depends on this one.

Of course, this isn't a rigid requirement, but it helps us clarify the module's purpose and makes it clear for our end users. This statement will be great to add to our package.json (which we'll look at later) and to the top of our readme file.

We'll eventually need a module name, which isn't vital at the start, but in order to refer to it in future techniques, let's call ours `fastfib`. Go ahead and make a fastfib directory that will serve as our project directory:

```
mkdir fastfib && cd fastfib
```

Now that we've defined our "one thing" we want our module to do and have our bare project directory, let's prove out our module idea in the next technique to see if it will actually work.

TECHNIQUE 107 Proving our module idea

So we have a focus now; what next? Time to prove our idea. This is the step where we think about the API surface of our module. Is it usable? Does it accomplish its purpose? Let's look at this next.

Problem

What should you code first when proving out your module idea?

Solution

Look at the API surface through TDD.

Discussion

It's important to know how you want your module to function. In `fastfib`, we'll calculate a Fibonacci sequence synchronously. What would be the simplest and easiest-to-use API we can think of?

```
fastfib(3) // => 2
```

Right, just a simple function call that returns the result.

When building an asynchronous API, it's recommended to use the Node callback signature, as it will work well with pretty much any control flow library. If our module were asynchronous, it would look like this:

```
fastfib(3, function (err, result) {
  console.log(result); // => 2
});
```

We have our synchronous API. In the beginning of this chapter, we showed you an implementation that we wanted to improve on. Since we want a baseline to compare other implementations, let's bring that recursive implementation into our project by creating a lib folder with a file called recurse.js with the following content:

```
module.exports = recurse;

function recurse (n) {
  if (n === 0) return 0;
  if (n === 1) return 1;
  return recurse(n-1) + recurse(n-2);
}
```

Exporting a single function to match our API design

Defining an entry point

Every module has an *entry point*: the object/function/constructor that we get when it's required elsewhere using the `require` keyword. Since we know that we'll be trying different implementations inside our lib directory, we don't want lib/recurse.js to be the entry point, as it may change.

Usually index.js in the project root makes the most sense as an entry point. Many times it makes sense to have the entry point be minimal in nature and just tie together the parts needed to provide the API to the end user. Let's create that file now:

```
module.exports = require('./lib/recurse');
```

Now when a consumer of the module does a `require('fastfib')`, they will get this file and in turn get our recursive implementation. We can then just switch this file whenever we need to change the exposed implementation.

Testing our implementation

Now that we have our first implementation of `fastfib`, let's ensure that we actually have a legit Fibonacci implementation. For that, let's make a folder called test with a single index.js file inside:

```
var assert = require('assert');
var fastfib = require ('../');

assert.equal(fastfib(0), 0);
assert.equal(fastfib(1), 1);
assert.equal(fastfib(2), 1);
assert.equal(fastfib(3), 2);
assert.equal(fastfib(4), 3);
assert.equal(fastfib(5), 5);
assert.equal(fastfib(6), 8);
assert.equal(fastfib(7), 13);
assert.equal(fastfib(8), 21);
assert.equal(fastfib(9), 34);
assert.equal(fastfib(10), 55);
assert.equal(fastfib(11), 89);
assert.equal(fastfib(12), 144);

// if we get this far we can assume we are on the right track
```

Now we can run our test suite to see if we're on track:

```
node test
```

We didn't get any errors thrown, so it looks like we're at least accurate in our implementation.

Benchmarking our implementation

Now that we have a well-defined API and tests around our implementation of `fastfib`, how do we determine how fast it is? For this we'll use a reliable JavaScript benchmarking tool behind the jsperf.com project called Benchmark.js (http://benchmarkjs.com/). Let's include it in our project:

```
npm install benchmark
```

Let's create another folder called benchmark and add an index.js file inside of it with the following code:

Set up a new benchmark suite.

```
var assert = require('assert');
var recurse = require('../lib/recurse');
var suite = new (require('benchmark')).Suite;
```

Include our recursive implementation to test.

```
suite
  .add('recurse', function () { recurse(20); })
.on('complete', function () {
  console.log('results: ') ;
  this.forEach(function (result) {
    console.log(result.name, result.count, result.times.elapsed);
  });
  assert.equal(
    this.filter('fastest').pluck('name')[0],
    'recurse',
    'expect recurse to be the fastest'
  );
})
.run();
```

After the tests complete, aggregate the results.

Add a test for the recurse function, calculating the 20th number in the Fibonacci sequence.

Assert that recurse was the fastest implementation; given it's the only implementation so far, that should be easy!

Output the test name, with the amount of iterations it was able to do in the elapsed time.

Let's run our benchmark now from the root module directory:

```
$ node benchmark
  results:
  recurse 392 5.491
```

Looks like we were able to calculate `recurse(20)` 392 times in ~5.5 seconds. Let's see if we can improve on that. The original recursive implementation wasn't tail call optimized, so we should be able to get a boost there. Let's add another implementation to the lib folder called tail.js with the following content:

```
module.exports = tail;

function tail (n) { return fib(n, 0, 1); }
function fib (n, current, next) {
  if (n === 0) return current;
  return fib(n -?1, next, current + next);
}
```

This recursive fibonacci function takes the next index n, the current sequence number, and the next sequence number.

In order to expose the same API as recurse, we add this function to set up our default values.

If we've reached the end, return the current sequence number.

Calculate the next call. This is in tail position because the calculations happen before the recursive function call and therefore are able to be optimized by the compiler.

Now, add the test to our benchmark/index.js file and see if we did any better by adding the implementation to the top of the file:

```
var recurse = require('../lib/recurse');
var tail = require('../lib/tail');
 ---
  .add('recurse', function () { recurse(20); })
  .add('tail', function () { tail(20); })
```

Require tail implementation after the recurse require

Add tail test after the recurse test

Let's see how we did:

```
$ node benchmark
results:
recurse 391 5.501
tail 269702 5.469
```

Putting our recursive function into tail position led to a 689x speedup!

```
assert.js:92
  throw new assert.AssertionError({
  ^
AssertionError: expect recurse to be the fastest
```

Our assertion failed, as recurse is no longer the fastest implementation.

Wow! Tail position really helped speed up our Fibonacci calculation. So let's switch that to be our default implementation in our main index.js file:

```
module.exports = require('lib/tail');
```

And make sure our tests pass:

```
node test
```

No errors; it looks like we're still good. As noted earlier, a proper tail call implementation will still blow our stack when it gets too large, due to it not being supported yet in JavaScript. So let's try one more implementation and see if we can get any better. To avoid a stack overflow on larger sequences of numbers, let's make an iterative implementation and create it at lib/iter.js:

Set up current and next defaults

Iterate through index n, swapping next and current values and incrementing the next value

```
module.exports = iter;

function iter (n) {
  var current = 0, next;
  for (var i = 0; i < n; i++) {
    swap = current, current = next;
    next = swap + next;
  }
  return current;
}
```

Return final current value

Let's add this implementation to the benchmark/index.js file:

Require iterative implementation after the tail require

```
var tail = require('../lib/tail');
var iter = require('../lib/iter');
 ---
  .add('tail', function () { tail(20) })
  .add('iter', function () { iter(20) })
```

Add iterative test after the tail test

Let's see how we did:

```
$ node benchmark
  results:
  recurse 392 5.456
  tail 266836 5.455
  iter 1109532 5.474
```

An iterative approach turns out to be 4x faster than the tail version, and 2830x faster than the original function. Looks like we have a `fastfib` indeed, and have proven our implementation. Let's update our benchmark/index.js file to assert that `iter` should be the fastest now:

```
assert.equal(
  this.filter('fastest').pluck('name')[0],
  'iter',
  'expect iter to be the fastest'
);
```

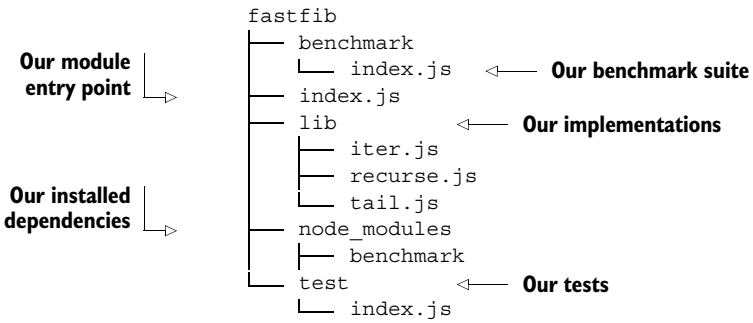
Then update our main index.js to point to our fastest version:

```
module.exports = require('./lib/iter');
```

And test that our implementation is still correct:

```
node test
```

No errors still, so we're good! If we later find that V8 optimizes tail call flows to be even faster than our iterative approach, our benchmark test will fail and we can switch implementations. Let's review our overall module structure at this point:



Looks like we've proved our idea. What's important to take away is to experiment! Try different implementations! You likely won't get it right initially, so take this time to experiment until you're satisfied. In this particular technique, we tried three different implementations until we landed one.

Time to look at the next step in module development: setting up a package.json file.

13.2 Building out the package.json file

Now we have an idea we like and we've proven that our idea does what we intend it to do, we'll turn to describing that module though a package.json file.

TECHNIQUE 108 Setting up a package.json file

A package.json is the central file for managing core data about your module, common scripts, and dependencies. Whether you ultimately publish your module or simply use it to manage your internal projects, setting up a package.json will help drive your

development. In this technique we'll talk about how to get a package.json set up and how to populate your package json using npm.

■ **Problem**

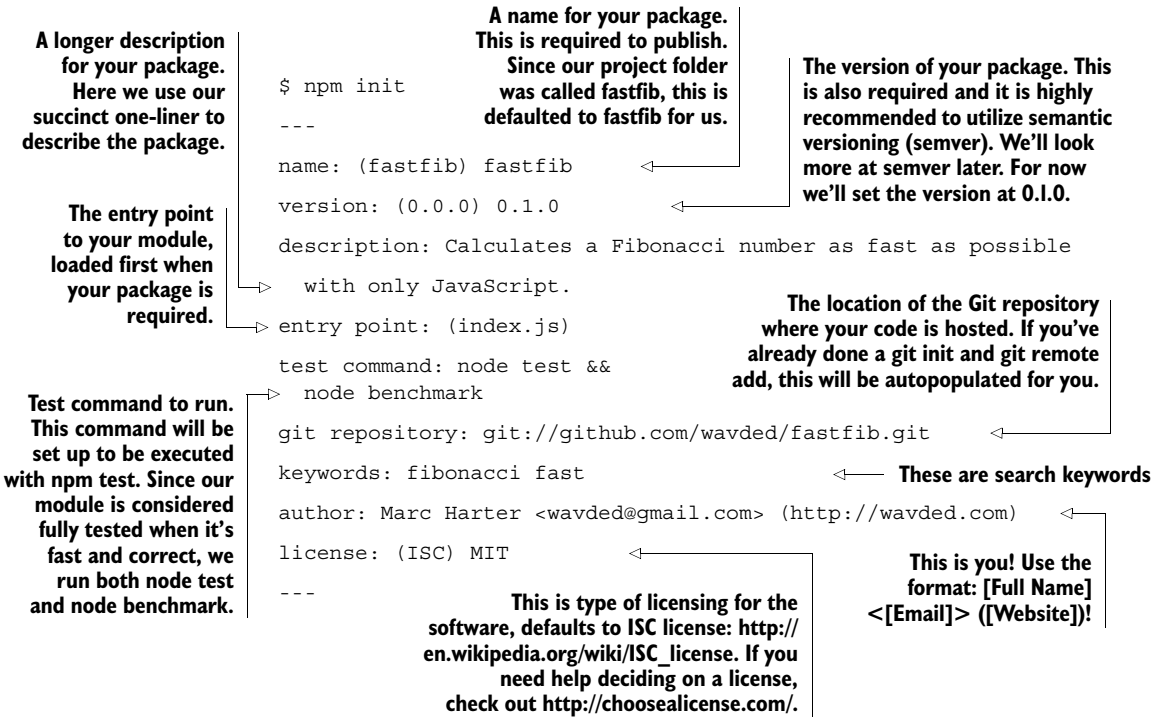
You need to create a package.json file.

■ **Solution**

Use the built-in npm tools.

■ **Discussion**

The `npm init` command provides a nice step-by-step interface for setting up a package.json. Let's run this command on our `fastfib` project directory:



PACKAGE OPTIONS For extensive detail on each package option, view the official documentation (https://www.npmjs.org/doc/json.html) by running `npm help json`.

Running `npm init` gets even simpler when you set up your user config (`$HOME/.npmrc`) to prepopulate the values for you. Here are all the options you can set:

```
npm config set init.author.name "Marc Harter"
npm config set init.author.email "wavded@gmail.com"
npm config set init.author.url "http://wavded.com"
npm config set init.license "MIT"
```

With these options, `npm init` won't ask you for an author, but instead autopopulate the values. It will also default the license to `MIT`.

A NOTE ABOUT EXISTING MODULES If you already have modules that you installed prior to setting up your package.json file, `npm init` is smart enough to add them to package.json with the correct versions!

Once you've finished initializing, you'll have a nice package.json file in your directory that looks something like this:

```
{
  "name": "fastfib",
  "version": "0.1.0",
  "description": "Calculates a Fibonacci number as fast
      as possible with only JavaScript.",
  "main": "index.js",
  "bin": {
    "fastfib": "index.js"
  },
  "directories": {
    "test": "test"
  },
  "dependencies": {
     "benchmark": "^1.0.0"
  },
  "devDependencies": {},
   "scripts": {
    "test": "node test && node benchmark"
  },
  "repository": {
    "type": "git",
    "url": "git://github.com/wavded/fastfib.git"
  },
  "keywords": [
    "fibonacci",
    "fast"
  ],
  "author": "Marc Harter <wavded@gmail.com> (http://wavded.com)",
  "license": "MIT",
  "bugs": {
     "url": "https://github.com/wavded/fastfib/issues"
  },
  "homepage": "https://github.com/wavded/fastfib"
}
```

Dependencies your module depends upon. Note how npm init discovered that we already were using the benchmark module and added it for us.

Additional dependencies only used for development. These are by default not included when someone installs your module.

Used by npm bugs to launch a browser at the location where issues can be reported. Since we're using a GitHub repository, npm init autopopulated this property for us.

The location of the project's homepage. Defaulted to GitHub since we entered a GitHub repository URL. This is used by npm docs to launch a browser at the project's homepage.

Now that we have a good start on a package.json file, we can add more properties by either directly modifying the JSON file or using other npm commands that modify different parts of the file for you. The `npm init` command just scratches the surface on what we can do with a package.json file. We'll look at more things we can add as we continue.

In order to look at more package.json configuration and other aspects of module development, let's head to the next technique.

TECHNIQUE 109 Working with dependencies

Node has over 80,000 published modules on npm. In our `fastfib` module, we've already tapped into one of those: the `benchmark` module. Having dependencies well defined in our package.json file helps maintain the integrity of our module when it's installed and worked on by ourselves and others. A package.json file tells npm what to fetch and *at what version* to fetch our dependencies when using `npm install`. Failing to include dependencies inside our package.json file will result in errors.

Problem

How do you effectively manage dependencies?

Solution

Keep the package.json file in sync with your module requirements using npm.

Discussion

The package.json file allows you to define four types of dependency objects, shown in figure 13.3.

The types of dependencies are as listed here:

- *`dependencies`*—Required for your module to function properly
- *`devDependencies`*—Required solely for development, like testing, benchmarking, and server reloading tools
- *`optionalDependencies`*—Not required for your module to work, but may enhance the functionality in some way
- *`peerDependencies`*—Requires another module to be installed in order to run properly

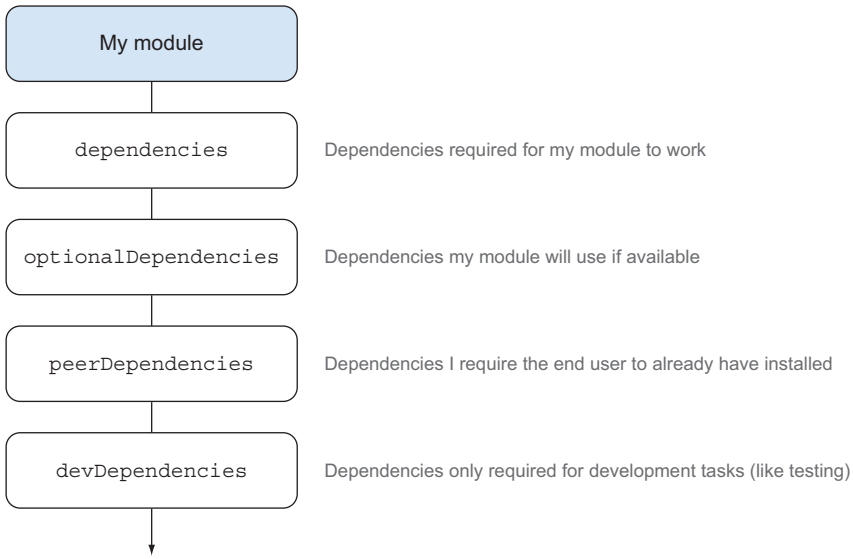


Figure 13.3 The different types of dependencies

Let's look at these in turn with our project and talk about adding and removing within your package.json file as we go.

Main and development dependencies

Currently the package.json file that was generated with `npm init` has `benchmark` listed in the `dependencies` object. If we look at our list, that doesn't hold true for a couple reasons. The first is because our main entry point (index.js) will never require `benchmark` in its require chain, so an end user has no need for it:

```
index.js requires ./lib/iter.js which requires nothing
```

The second reason is because benchmarking is typically a development-only thing for those who work on our module. To remove that out of our dependencies, we can use `npm remove` and have it removed from our package.json file using the `--save` flag:

```
$ npm remove benchmark --save
  unbuild benchmark@1.0.0
```

Then we can install it into our development dependencies using `npm install` with the `--save-dev` flag:

```
$ npm install benchmark --save-dev
  benchmark@1.0.0 node_modules/benchmark
```

Now if we look at our package.json file, we'll see that `benchmark` is now a part of the `devDependencies` object:

```
"devDependencies": {
    "benchmark": "^1.0.0"
},
```

This was somewhat of a brute force way to show you the commands to remove and install with npm. We could have also just moved `benchmark` inside the package.json file in our text editor, avoiding the uninstall and re-install.

Now we have `benchmark` in the right spot, so it won't be installed when others want to use our module.

Optional dependencies

Optional dependencies aren't required for a project to run, but they will be installed along with the regular dependencies. The only difference from normal dependencies is that if an optional dependency *fails* to install, it will be ignored and the module should continue to install properly.

This typically plays out for modules that can get a boost by including a native add-on. For example, `hiredis` is a native C add-on to boost performance for the `redis` module. But it can't be installed everywhere, so it *attempts* to install, but if it fails, the `redis` module falls back to a JavaScript implementation. A typical pattern to check for the dependency in the parent module is this:

```
try {
  var client = require('hiredis'); // super fast!
}
```

Attempt to load the optional dependency.

```
catch (e) {
  var client = require('./lib/redis'); // fast
}

module.exports = client;
```

If that fails, continue without the dependency, perhaps shimming with another implementation.

Expose a common interface entry point anyone can use regardless of whether they got the optional dependency.

Let's say we wanted to support a larger set of sequence numbers for our `fastfib`. We could add the `bignum` native add-on to enable that functionality by running

```
npm install bignum --save-optional
```

Then we could optionally use that iteration instead if we detect the `bignum` module was able to be installed in our index.js file:

```
try {
    var fastfib = require('./lib/bigiter');
}
catch (er) {
    var fastfib = require('./lib/iter');
}

module.exports = fastfib;
```

Try to include bignum implementation of fastfib.

If that fails, include iterative implementation.

Unfortunately, the `bignum` implementation would be much slower, as it can't be optimized by the V8 compiler. We'd be violating our goal of having the fastest Fibonacci if we included that optional dependency and implementation, so we'll scratch it out for now. But this illustrates how you may want to use optional dependencies (for example, if you wanted to support the highest possible Fibonacci numbers as your goal).

> **HOMEWORK** The code and tests were intentionally left out for the `bignum` implementation; try implementing a version that uses `bignum` and see what performance benchmarks you get from our test suite.

Peer dependencies

Peer dependencies (http://blog.nodejs.org/2013/02/07/peer-dependencies/) are the newest to the dependency scene. Peer dependencies say to someone installing your module: *I expect this module to exist in your project and to be at this version in order for my module to work.* The most common type of this dependency is a plugin.

Some popular modules that have plugins are

- Grunt
- Connect
- winston
- Mongoose

Let's say we *really* wanted to add a Connect middleware component that calculates a Fibonacci number on each request; who wouldn't, right? In order for that to work,

we need to make sure the API we write will work against the right version of Connect. For example, we may trust that for Connect 2 we can reliably say our module will work, but we can't speak for Connect 1 or 3. To do this we can add the following to our package.json file:

```
"peerDependencies": {
  "connect": "2.x"
 }
```

Only allow module to be installed if Connect 2.x is also installed.

In this technique we looked at the four types of dependencies you can define in your package.json file. If you're wondering what ^1.0.0 or 2.x means, we'll cover that in depth in the next technique, but let's first talk about updating existing dependencies.

Keeping dependencies up to date

Keeping a module healthy also means keeping your dependencies up to date. Thankfully there are tools to help with that. One built-in tool is `npm outdated`, which will strictly match your package.json file as well as all the package.json files in your dependencies, to see if any newer versions match.

Let's purposely change our package.json file to make the `benchmark` module out of date, since `npm install` gave us the latest version:

```
"devDependencies": {
  "benchmark": "^0.2.0"
 },
```

Roll benchmark back to earlier version.

Then let's run `npm outdated` and see what we get:

```
$ npm outdated
  Package  Current  Wanted  Latest  Location
  benchmark  1.0.0   0.2.2   1.0.0  benchmark
```

Looks like we have 1.0.0 currently installed, but according to our package.json we just changed, we want the latest package matching ^0.2.0, which will give us version 0.2.2. We also see the latest package available is 1.0.0. The location line will tell us where it found the outdated dependencies.

> **OUTDATED DEPENDENCIES THAT YOU DIRECTLY REQUIRE** Often it's nice to see just your outdated dependencies, not your subdependencies (which can get very large on bigger projects). You can do that by running `npm outdated --depth 0`.

If we want to update to the `wanted` version, we can run

```
npm update benchmark --save-dev
```

This will install 0.2.2 and update our package.json file to ^0.2.2.

Let's run `npm outdated` again:

```
$ npm outdated
  Package Current Wanted Latest Location
  benchmark 0.2.2  0.2.2  1.0.0 benchmark
```

Looks like our current and our desired versions match now. What if we wanted to update to the latest? That's easy: we can install just the latest and save it to our package.json by running

```
npm install benchmark@latest --save-dev
```

> **VERSION TAGS AND RANGES** Note the use of the `@latest` tag in order to get the latest published version of a module. npm also supports the ability to specify versions and version ranges, too! (https://www.npmjs.org/doc/cli/npm-install.html)

We've talked a little about version numbers so far, but they really need a technique unto their own, as it's important to understand what they mean and how to use them effectively. Understanding semantic versioning will help you define versions better for your module and for your dependencies.

TECHNIQUE 110 Semantic versioning

If you're not familiar with semantic versioning, you can read up on it at http://semver.org. Figure 13.4 captures the major points.

Here is how it's described in the official documentation:[1]

Given a version number MAJOR.MINOR.PATCH, increment the:

1. *MAJOR version when you make incompatible API changes,*
2. *MINOR version when you add functionality in a backwards-compatible manner, and*
3. *PATCH version when you make backwards-compatible bug fixes.*

In practice, these rules can be ignored or loosely followed, since, after all, *nobody is mandating* your version numbers. Also, many authors like to play around with their API in the early stages and would prefer not to be at version 24.0.0 right away! But semver can give you, as a module author and as a module consumer, clues within the version number itself as to what may have happened since the last release.

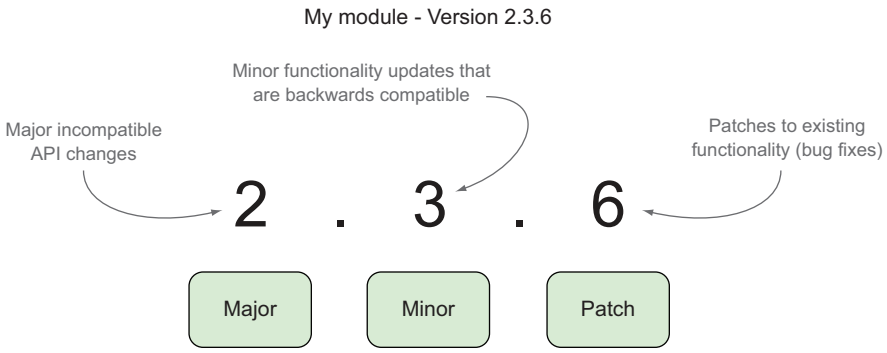


Figure 13.4 Semantic versioning

[1] From http://semver.org/.

In this technique we'll look at how to use semver effectively within our `fastfib` library.

Problem

You want to use semver effectively in your module and when including dependencies.

Solution

Understand your underlying projects in order to have a safe upgrade path, and clearly communicate the intent of your module version.

Discussion

We currently have one development dependency in our project, which looks like this in the package.json file:

```
"devDependencies": {
    "benchmark": "^1.0.0"
},
```

This is how npm, by default, will include the version inside the package.json file. This plays nice for how most modules authors behave:

- If the version is less than 1.0.0, like ^0.2.0, then allow any greater PATCH version to be installed. In the previous technique, we saw this ended up being 0.2.2 for the `benchmark` module.
- If the version is 1.0.0 or greater, like ^1.0.0, then allow any greater MINOR version to be installed. Typically 1.0.0 is considered stable and MINOR versions aren't breaking in nature.

This means that when another user installs your module dependencies, they'll get the latest version that's allowed in your version range. For example, if Benchmark.js released version 1.1.0 tomorrow, although you currently have 1.0.0 on your machine, they would get version 1.1.0, since it still matches the version range.

> **VERSION OPERATORS** Node supports a whole host of special operators to customize multiple versions or version ranges. You can view them in the semver documentation (https://www.npmjs.org/doc/misc/semver.html).

Versioning dependencies

When writing modules, it can increase the confidence in your dependencies to use a specific version number a user will install along with your module. This way, you know what you've tested will run the same down the dependency chain. Since we know our test suite works with benchmark 1.0.0, let's lock it in to be only that version by running the following:

```
npm install benchmark --save-dev --save-exact
```

Save exact same version installed to the package.json file.

We could've so updated our package.json manually. Let's take a look at what it looks like now:

```
"devDependencies": {
  "benchmark": "1.0.0"
},
```

Exact versions have no special identifiers.

Now that we've locked in our dependency, we can always use `npm outdated` to see if a new version exists and then `npm install` using the `--save-exact` flag to update our package.json!

Versioning the module

As already noted, many module authors use versions less than 1.0.0 to indicate that the API hasn't been fully implemented yet and may change in subsequent versions. Typically, when the version number hits 1.0.0, there's some good stability to the module, and although the API surface may grow, existing functionality shouldn't change that much. This matches how npm behaves when a module is saved to the package.json file.

Currently we have our `fastfib` module at version 0.1.0 in the package.json file. It's pretty stable, but there may be other changes we want to make before we give it the 1.0.0 status, so we'll leave it at 0.1.0.

The change log

It's also helpful for module authors to have a change log summarizing anything users should be aware of when new releases happen. Here's one such format:

```
Version 0.5.0?--?2014-04-03
---
added; feature x
removed; feature y [breaking change!]
updated; feature z
fixed; bug xx

Version 0.4.3?--?2014-03-25
---
```

Breaking changes, especially in a minor version, should be noted clearly in the change log so users know how to prepare for the update. Some authors like to keep a change log inside their main readme or have a separate change log file.

We've covered some understanding and tooling around versioning our dependencies and our module; let's look at what else we can expose to the consumers of our modules.

13.3 The end user experience

Before we push our module out for consumption, it would be nice to test that it actually works. Of course, we already have a test suite, so we know our logic is sound, but what is the experience of an end user installing the module? How do we expose executable scripts to a user in addition to an API? What versions of Node can we support? In this section we'll take a look at those questions, starting with adding executable scripts.

TECHNIQUE 111 Adding executable scripts

Want to expose an executable when your module is installed? Express, for example, includes an `express` executable you can run from the command line to help initialize new projects:

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

Installs express module globally making the executable accessible from anywhere

npm itself is an installable module with an npm executable, which we've been using all over in this chapter.

Executables can help end users use your module in different ways. In this technique we'll look at adding an executable script to `fastfib` and include it in our package.json to be installed along with our module.

Problem

How do you add an executable script?

Solution

How do you add command-line tools and scripts for a package and link it inside the package.json file?

Discussion

We have our `fastfib` module built, but what if we wanted to expose a `fastfib` executable to the end user where they could run a command like `fastfib 40` and get the 40th Fibonacci number printed out? This would allow our module to be used on the command line as well as programmatically.

In order to do this, let's create a bin directory with an index.js file inside containing the following:

Indicate operating system should look for a node executable to run following code

Require fastfib module

Get sequence number argument

If we didn't get valid number, exit early and give error message with instructions

```
#!/usr/bin/env node
var fastfib = require('../');
var seqNo = Number(process.argv[2]);

if (isNaN(seqNo)) {
   return console.error('\nInvalid sequence number provided,
    try:\n fastfib 30\n');
}

console.log(fastfib(seqNo));
```

Output result

Now that we have our application executable, how do we expose it as the `fastfib` command when someone installs our module? For that, we need to update our package.json file. Add the following lines underneath `main`:

```
"main": "index.js",
"bin": {
  "fastfib": "./bin/index.js"
},
```

Alias executable as fastfib and have it run ./bin/index.js

Testing executables with npm link

We can test our executable by using `npm link`. The `link` command will create a global symbolic link to our live module, simulating installing the package globally, as a user would if they installed the module globally.

Let's run `npm link` from our `fastfib` directory:

```
$ npm link
/usr/bin/fastfib
  -> /usr/lib/node_modules/fastfib/bin/index.js
/usr/lib/node_modules/fastfib
  -> /Users/wavded/Dev/fastfib
```

Link fastfib executable to the ./bin/index.js file

Link fastfib module to our working directory code

Now that we've globally linked up our executable, let's try it out:

```
$ fastfib 40
  102334155
```

Since these links are in place now, any edits will be reflected globally. Let's update the last line of our bin/index.js file to announce our result:

```
console.log('The result is', fastfib(seqNo));
```

If we run the `fastfib` executable again, we get our update immediately:

```
$ fastfib 40
  The result is 102334155
```

We've added a `fastfib` executable to our module. It's important to note that everything discussed in this technique is completely cross-platform compatible. Windows doesn't have symbolic links or #! statements, but npm wraps the executable with additional code to get the same behavior when you run `npm link` or `npm install`.

Linking is such a powerful tool, we've devoted the next technique to it!

TECHNIQUE 112 Trying out a module

Besides using `npm link` to test our executables globally, we can use `npm link` to try out our module elsewhere. Say we wanted to try out our shiny new module in another project and see if it'll work out. Instead of publishing our module and installing it, we can just link to it and play around with the module as we see it used in the context of another project.

Problem

You want to try out your module before publishing it *or* you want to make changes to your module and test them in another project without having to republish first.

Solution

Use `npm link`

Discussion

In the previous technique we showed how to use `npm link` to test an executable script's behavior. This showed that we can test our executables while we're developing, but now we want to simulate a local install of our module, not a global one.

Let's start by setting up another project. Since we started this chapter with our cancerous implementation of a Fibonacci web server, let's go full circle and make a little project that exposes `fastfib` as a web service.

Create a new project called `fastfibserver` and put a single server.js file inside with the following content:

```
var fastfib = require('fastfib');
var http = require('http');

http.createServer(function (req, res) {
  res.end(fastfib(40));
}).listen(3000);

console.log('fastfibber running on port 3000');
```

Require fastfib module

Respond with 40th Fibonacci number on every request

We have our server set up, but if we were to run `node server`, it wouldn't work yet because we haven't installed the `fastfib` module in this project yet. To do that we use `npm link`:

```
$ npm link ../fastfib
  /usr/bin/fastfib
    -> /usr/lib/node_modules/fastfib/bin/index.js
  /usr/lib/node_modules/fastfib
    -> /Users/wavded/Dev/fastfib
  /Users/wavded/Projects/Dev/fastfibserver/node_modules/fastfib
    -> /usr/lib/node_modules/fastfib
    -> /Users/wavded/Dev/fastfib
```

Pass in path to fastfib module

Links are created globally first

A final link is set up in fastwebserver project

Now if we run our web server, it will run successfully:

```
$ node server
  fastfibber running on port 3000
```

And a visit to our site will give us the 40th Fibonacci number, as shown in figure 13.5.

> **ANOTHER WAY TO LINK** Since we already linked our module globally in the previous technique running `npm link` inside the `fastfib` project, we could've also run `npm link fastfib` in our `fastfibserver` project to set up the link.

Using `npm link` also helps a lot in debugging your module in the context of another module. Edge cases come up that can best be debugged while running the project that's requiring your module. Once you `npm link` the module, any changes will take

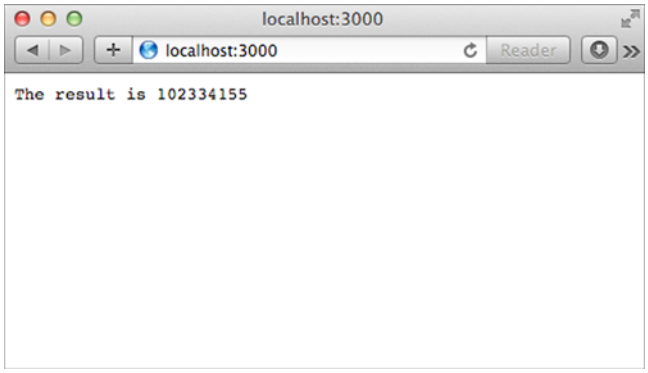


Figure 13.5 Sample output from fastfibserver

effect immediately without the need to republish and re-install. This allows you to fix the problem in your module's code base as you debug.

So far we've defined and implemented our module with tests, set up our dependencies, locked our versions down for our dependencies and our module, added a command-line executable, and practiced using our module. Next we'll look at another aspect of the package.json file—the engines section, and testing our module across multiple versions of Node.

TECHNIQUE 113 Testing across multiple Node versions

Unfortunately, not everybody is able to upgrade to the latest and greatest Node version when it comes on the scene. It takes time for companies to adapt all their code to newer versions, and some may never update. It's important that we know what versions of Node our module can run on so npm knows who can install and run it.

Problem

You want to test your module across multiple versions of Node, and you want your application to be installed only for those versions.

Solution

Keep the `engines` object accurate in the package.json file by running tests across multiple versions of Node.

Discussion

The `npm init` script we ran when first setting up our package.json doesn't include an `engines` section, which means that npm will install it on *any* version of Node. At first glance, we may think that's OK, since we're running pretty vanilla JavaScript code. But we don't really know that without actually testing it.

Typically patch version updates (Node 0.10.2 to 0.10.3, for instance) shouldn't break your modules. But it's a good idea at a minimum to test your modules across minor and major version updates, as V8 receives a decent upgrade and Node's APIs can change. Currently, we've been working off 0.10 branch of Node and things have been working well. So let's start with that. Let's add the following to the end of our package.json file:

```
"homepage": "https://github.com/wavded/fastfib",
  "engines": {
    "node": "0.10.x"
  }
}
```

Indicate that our module can run on any patch version of Node 0.10.

That's a start, but it really seems like we should be able to support earlier versions of Node. How do we test for that?

A variety of popular options are available:

- Install multiple versions of Node on your machine
- Use Travis CI's multi-Node version support (https://travis-ci.org/)
- Use a third-party multiversion test module that works for your environment (like `dnt`—https://github.com/rvagg/dnt)

ABOUT NODE VERSIONS In Node, all odd-numbered minor versions are considered unstable. So 0.11.0 is the unstable version for 0.12.0, and so on. You shouldn't need to test any existing unstable releases. Typically, module authors will only test the latest unstable release as it nears completion.

For our technique we'll focus on installing multiple versions of Node, as that can come in handy for testing new features in upcoming versions of Node, as well as for testing our module.

The tool we'll use is nvm (https://github.com/creationix/nvm; the Windows counterpart is nvmw: https://github.com/hakobera/nvmw). The following instructions will be for nvm, but the commands will be similar in nvmw once installed.

To install, run

```
curl https://raw.github.com/creationix/nvm/v0.5.0/install.sh | sh
source ~/.nvm/nvm.sh
```

We source it right away so we don't have to reload our session. This is done automatically in future sessions.

Now that we have it installed, let's go ahead and test Node version 0.8 of our `fastfib` module. First let's install Node 0.8:

```
$ nvm install 0.8
  ######################################################### 100.0%
  Now using node v0.8.26
```

nvm went out and grabbed the latest version of the 0.8 branch to test against. We could have specified a patch if we wanted, but this will work for now. Note how we're also using this version. We can validate that by running

```
$ node -v
  v0.8.26
```

Now, all Node and npm interaction happens within an isolated environment just for Node 0.8.26. If we were to install more versions, they would be in their own isolated environments. We use `nvm use` to switch between them. For example, if you wanted to go back to your system install of Node, you could do the following:

```
nvm use system
```

And to go back to Node version 0.8.26:

```
nvm use 0.8
```

Let's run our test suite in 0.8.26 and see how we do:

```
$ npm test

  > fastfib@0.1.0 test /Users/wavded/Dev/fastfib
  > node test && node benchmark

  results:
  recurse 432 5.48
  tail 300770 5.361
  iter 1109759 5.428
```

Looks good! Let's update our package.json to include 0.8 versions:

```
"engines": {
  "node": ">=0.8.0 <0.11.0"
}
```

Include any version from 0.8.0 up to but not including 0.11.0.

WHAT IF MY MODULE LOSES SUPPORT FOR A PARTICULAR NODE VERSION? That's totally fine. Users of an older version of Node will get the last-published package that's supported for their version.

We've tested Node version 0.10 and 0.8; try testing a few other versions on your own. When you're done, switch back to the system Node version.

Now that we've looked through a variety of steps to get our module into a usable state for others, let's publish it!

13.4 Publishing

As we wrap up this chapter, we'll turn our focus on module distribution by looking at publishing modules publicly on npm or privately for internal use.

TECHNIQUE 114 Publishing modules

Whew! We've gone through a lot of different techniques getting our module ready to publish. We know there will likely be changes, but we're ready to release our first version out in the wild to be required in other projects. This technique explores the various aspects of publishing.

- **Problem**

You want to get your module published publicly.

- **Solution**

Register with npm if you haven't and `npm publish`.

- **Discussion**

If it's your first time publishing a module, you'll need to register yourself with npm. Thankfully, it couldn't be any easier. Run the following command and follow the prompts:

```
npm adduser
```

Once finished, npm will save your credentials to the .npmrc file.

CHANGING EXISTING ACCOUNT DETAILS The `adduser` command can also be used to change account details (except username) and register a fresh install with an existing account.

Once registered, publishing a module is just as simple as adding a user. But before we get to that, let's cover some good practices when publishing modules.

Before you publish

One of the biggest things before publishing is to review technique 110 about semantic versioning:

- Does your version number accurately reflect the changes since the last push? If this is your first push, this doesn't matter as much.
- Do you do a changelog update with your release? Although not required, it can be extremely helpful to those who depend on your project to get a high-level view of what they can expect in this release.

Also, check whether your tests pass to avoid publishing broken code.

Publishing to npm

Once you're ready to publish, it's as simple as running the following command from the project root directory:

```
npm publish
```

npm will respond with the success or failure of the publish. If successful, it will indicate the version that was pushed to the public registry.

Can you tell that npm wants you to get your modules out there as painlessly as possible?

Undoing a publish

Although we want a publish to go well, sometimes we miss things we wanted to release, or have some things we realize are broken after the fact. It's recommended you don't unpublish modules (although the ability exists). The reason is that people who are depending on that module and/or version can no longer get it.

Typically, make the fix, increase the PATCH version, and `npm publish` again. A simple way to do that is by running the following commands:

```
// make fixes
$ npm version patch
 v0.1.1
$ npm publish
```

The npm version command (https://www.npmjs.org/doc/cli/npm-version.html) will update your package.json based on the arguments passed. Here we used patch to tell it to increment the patch version.

npm *does not* allow you to publish over an existing version, since that also would affect people who have already downloaded that particular version.

There are some cases where you really want to discourage users from using a particular version. For example, maybe a severe security flaw was fixed in versions 0.2.5 and above, yet you have users depending on versions earlier than that. npm can help you get the word out by using `npm deprecate`.

Let's say in the future, we find a critical bug in `fastfib` version 0.2.5 and below, and we wanted to warn users who are using those modules. We could run the following:

```
npm deprecate fastfib@"<= 0.2.5"
  "major security issue was fixed in v0.2.6"
```

Deprecate module name followed by version range affected and message NPM should display

Now if any user installs `fastfib` 0.2.5 or less, they'll receive the specified warning from npm.

TECHNIQUE 115 Keeping modules private

Although open source can be a fun and collaborative environment, there are times when you want your project to remain private. This is especially true for work done for clients. It can also be handy to bake a module first internally before deciding whether to publish. npm can safeguard your module and keep it private for you. In this technique we'll talk about configuring your module to stay private and including private modules in your projects.

Problem

You want to keep your module private and use it internally.

Solution

Configure `private` in your package.json file and share it internally.

Discussion

Let's say we want to let `fastfib` to only be used internally. To ensure it doesn't get accidentally published, we add the following to our package.json file:

```
"private": true
```

This tells npm to refuse to publish your package with `npm publish`.

This setting works well for client-specific projects. But what if you have a core set of internal modules you want to share across projects within your development team? For that there are a few different options.

Sharing private modules with Git

npm supports a couple of ways you can share your internal modules that are minimal to set up out of the box. If you're using a Git repository, npm makes this incredibly simple to do.

Let's use GitHub as an example (although it can be *any* Git remote). Let's say we had our private repo at

```
git@github.com:mycompany/fastfib.git
```

We can then include it in our package.json dependencies with `npm install` (or modify the package.json directly):

```
npm install git+ssh://git@github.com:mycompany/fastfib.git --save
```

Pretty sweet! This by default will pull the contents of the `master` branch. If we wanted to specify a particular commit-ish (tag, branch, or SHA-1—http://git-scm.com/book/en/Git-Internals-Git-Objects), we can do that too! Here are some examples within a package.json file:

```
"dependencies": {
  "a": "git+ssh://git@github.com:mycompany/a.git#0.1.0",
  "b": "git+ssh://git@github.com:mycompany/b.git#develop",
  "c": "git+ssh://git@github.com:mycompany/c.git#dacc525c"
}
```

Specifying by tag

Specifying by branch name

Specifying by commit SHA-1 (typically you won't need the whole SHA-1)

INCLUDING PUBLIC REPOSITORIES You may have guessed it, but you can also use `public` Git repositories as well. This can be helpful if you really need a feature or fix that hasn't been published on npm yet. For more examples, see the package.json documentation (https://www.npmjs.org/doc/json.html#Git-URLs-as-Dependencies).

Sharing private modules as a URL

If you aren't using Git or prefer to have your build system spit out packages, you can specify a URL endpoint where npm can find a tarball. To package up your module, you can use the `tar` command like the following:

```
tar -czf fastfib.tar.gz fastfib
```

We tell tar to make a new archive (c), compress the archive using gzip (z), and store it in the file (f) fastfib.tar.gz, giving it the contents of the fastfib directory.

From here, we can throw that file on a web server and install it using the following:

```
npm install http://internal-server.com/fastfib.tar.gz --save
```

A NOTE ABOUT PUBLIC ENDPOINTS Although typically not used often, tarballs of packages can be used with public endpoints too; it's usually better and easier to publish to npm instead.

Sharing modules with a private npm registry

Another option for private repositories is hosting your own private npm registry and having `npm publish` push to that repository. For the complete functionality of npm, this will require an installation of a recent version of CouchDB, which, in turn, requires Erlang.

Since this involves a variety of tricks/headaches depending on your operating system, we won't cover setting up an instance here. Hopefully, the process will get streamlined soon. If you want to experiment, check out the `npm-registry-couchapp` project (https://github.com/npm/npm-registry-couchapp).

13.5 Summary

Third-party modules are where innovation happens. npm makes this trivial and fun! With the rise of social coding sites like GitHub, collaboration on modules is also easy to do. In this chapter we looked at many different aspects of module development. Let's summarize what we learned.

When starting to work on a module, consider the following:

- Define your module idea. Can you summarize it in one sentence?
- Check your module idea. Is there another module out there doing what you want to do? Search it out with `npm search` or npmjs.org.

Once you've landed on an idea, prove it out. Start with a simple API you'd like to work with. Write an implementation and tests installing any dependencies you need along the way.

After you've proven your idea (or perhaps during), think about these things:

- Have you initialized your package.json file? Run `npm init` to get a skeleton representing the state of the current project.
- Work with your dependencies. Are some optional, development-only? Make sure that's indicated in your package.json file.
- Check your semver ranges in package.json. Do you trust the version ranges specified in your package.json file? Check for updates with `npm outdated`.
- What versions of Node will your code run on? Check it out by using nvm or a build system like Travis CI. Specify the version range in your package.json file.
- Try out your module using `npm link` in another project.

When you're ready to publish, it's as simple as `npm publish`. Consider keeping a changelog for your users and try to follow semantic versioning, so users have a reasonable idea of what to expect from version to version.

And that's a wrap for this book! We hope at this point you were able to grasp the core foundations of Node, understand how to apply those foundations in real-world scenarios, and how to go beyond standard development by writing your own Node modules (which we hope to see on npm!).

A growing Node community is available to help you continue to level up on your journey. Please check out the appendix to make the most of that community. If you have specific questions for us, please visit the #nodejsinpractice Google group (https://groups.google.com/forum/#!forum/nodejsinpractice), and thanks for reading!

appendix
Community

This section will help you to make the most of the growing Node community. Programming communities can help you get answers to problems that aren't directly answered by the documentation. You can learn more effectively just by hanging out with like-minded people—whether online or in person.

A.1 Asking questions

Sometimes you just want to know how to do something that seems like it should be easy, but isn't. Other times you think you might have found a serious bug in Node. Whatever the situation, when you need help that isn't satisfied by Node's API documentation, there are several official channels you can use.

The first is the Node mailing list, which is the nodejs Google Group (http://groups.google.com/group/nodejs). You can subscribe by email or use Google's web interface. The web interface allows posts to be searched, so you can see if someone has asked your question before.

The group has contributions from prominent community members, including Isaac Schlueter, Mikeal Rogers, and Tim Caswell, so it's a good place to get help and learn about Node in general.

There's also an official IRC chat room: `#node.js` on `irc.freenode.net`. It's extremely busy though, so be prepared for a lot of messages. Informative discussions do happen in `#node.js`, so some patience may be rewarded!

If you're a fan of the Stack Exchange network, you can post questions using the node.js tag (http://stackoverflow.com/questions/tagged/node.js).

If you prefer social networks, the Node users group (https://github.com/joyent/node/wiki/Node-Users) in the Node wiki lists hundreds of Twitter accounts alongside the developer's time zone, so you could look for people to talk to that way. Hint: the authors of this book are listed!

Finally, if your question is about a specific module, you should check that module's documentation for community information. For example, the Express web framework has its own express-js Google Group (https://groups.google.com/group/express-js).

A.2 Hanging out

Your city may have an active Node meet-up group. Examples include the London Node.js User Group (http://lnug.org/), the Melbourne Node.JS Meetup Group (http://www.meetup.com/MelbNodeJS/), and BayNode (http://meetup.com/BayNode/) in Mountain View, California.

There are also major Node conferences, including NodeConf (http://nodeconf.com/) and NodeConf EU (http://nodeconfeu.com/).

To help you find more meet-up groups and conferences, the Node.js Meatspace page at https://github.com/knode/node-meatspace is frequently updated. You can, of course, try searching at meetup.com as well.

A.3 Reading

If you're looking for something to read, you'll find some great community publications. Naturally reddit.com/r/node collects some great posts, but there are also collections on Medium, including medium.com/node-js-javascript.

Noted Node developers have blogs you can check as well. Isaac Z. Schlueter (http://blog.izs.me/), James Halliday (http://substack.net/; see figure A.1), and Tim Caswell all have personal blogs where they write about Node. Tim's howtonode.org has material suitable for beginners, but will also help you keep track of new developments.

There are also commercial blogs that have some contributions from talented Node developers. Joyent's blog at joyent.com/blog often has interesting posts relating to deploying Node, and StrongLoop's blog, "In the Loop," at strongloop.com/strong-blog, does as well.

Nodejitsu's blog at blog.nodejitsu.com has advice on deployment, and also features module authors talking about their work.

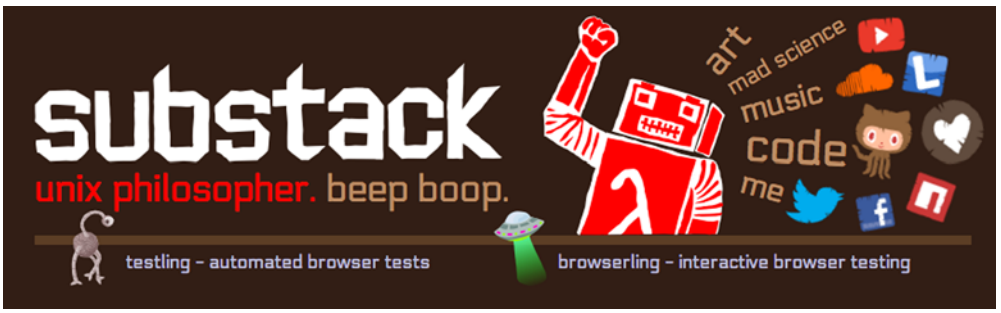


Figure A.1 **James Halliday's blog about Node and testing**

A.4 Training by the community, for the community

One interesting development in teaching Node is NodeSchool (http://nodeschool.io/; see figure A.2). You can install lessons yourself, but there are also community-run in-person training events. NodeSchool provides the materials to set up training events, so they're proliferating rapidly around the world. The site has more details on upcoming events.

Figure A.2
Learn Node with NodeSchool.

A.5 Marketing your open source projects

If you're going to take part in the Node community, one of the best ways to it is to share your work. But npm is now so popular that it's hard to get your module noticed.

To really make an impact, you should consider marketing your open source projects. If prominent Node bloggers have contact forms or Twitter accounts, it won't hurt to tell them about what you've made. As long as you're polite, and give your work some context so it's easy to understand, then it can really help you get feedback and improve your skills.

index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

WEB DEVELOPMENT/JAVASCRIPT

Node.js IN PRACTICE

Young • Harter

You've decided to use Node.js for your next project and you need the skills to implement Node in production. It would be great to have Node experts Alex Young and Marc Harter at your side to help you tackle those day-to-day challenges. With this book, you can!

Node.js in Practice is a collection of 115 thoroughly tested examples and instantly useful techniques guaranteed to make any Node application go more smoothly. Following a common-sense Problem/Solution format, these experience-fueled techniques cover important topics like event-based programming, streams, integrating external applications, and deployment. The abundantly annotated code makes the examples easy to follow, and techniques are organized into logical clusters, so it's a snap to find what you're looking for.

What's Inside

- Common usage examples, from basic to advanced
- Designing and writing modules
- Testing and debugging Node apps
- Integrating Node into existing systems

Written for readers who have a practical knowledge of JavaScript and the basics of Node.js.

Marc Harter works daily on large-scale projects including high-availability real-time applications, streaming interfaces, and other data-intensive systems. Alex Young is a seasoned JavaScript developer who blogs regularly at DailyJS.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/Node.jsinPractice

"An in-depth tour of Node.js."

—From the Foreword by Ben Noordhuis, Cofounder of StrongLoop, Inc.

"The missing manual for Node.js, packed with real-world examples!"

—Kevin Baister
1KB Software Solutions Ltd.

"Essential recipes for the server-side JavaScript developer."

—Gregor Zurowski, Sotheby's

"Useful techniques and resources that help with problem solving, debugging, and troubleshooting."

—Michael Piscatello
MBP Enterprises, LLC

ISBN 13: 978-1-617290-93-0
ISBN 10: 1-617290-93-9
54999
9 781617 290930

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]