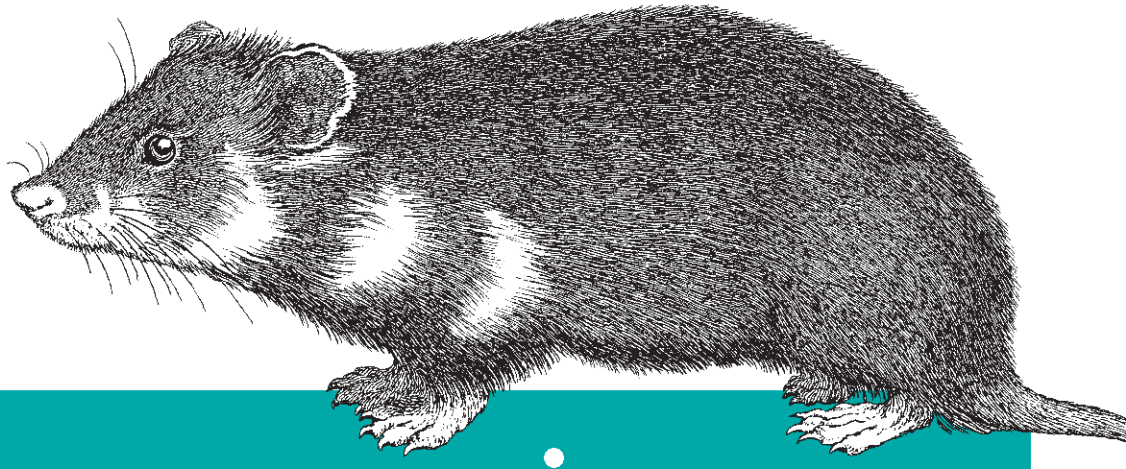
O'REILLY®

2nd Edition



Learning Node

MOVING TO THE SERVER-SIDE

Shelley Powers

www.allitebooks.com

O'REILLY®

Learning Node

Take your web development skills from browser to server with Node—and learn how to write fast, highly scalable network applications on this JavaScript-based platform. Updated for the latest Node Long Term Support (LTS) and Node Current (6.0) releases, this hands-on edition helps you master Node's core fundamentals and gain experience with several built-in and contributed modules.

Get up to speed on Node's event-driven, asynchronous I/O model for developing data-intensive applications that are frequently accessed but computationally simple. If you're comfortable working with JavaScript, this book provides many programming and deployment examples to help you take advantage of server-side development with Node.

- Explore the frameworks and functionality for full-stack Node development
- Dive into Node's module system and package management support
- Test your application or module code on the fly with Node's REPL console
- Use core Node modules to build web applications and an HTTP server
- Learn Node's support for networks, security, and sockets
- Access operating system functionality with child processes
- Learn tools and techniques for Node development and production
- Use Node in microcontrollers, microcomputers, and the Internet of Things.

"Shelley Powers' *Learning Node* not only provides an effective introduction to Node, it manages to convey the sense of excitement and fun that exemplifies this powerful and useful technology. Highly recommended."

—Ethan Brown
Author and Director of Engineering, Pop Art

Shelley Powers has been working with and writing about web technologies for more than 12 years, from the first release of JavaScript to the latest graphics and design tools. In recent O'Reilly books, she's covered the semantic Web, Ajax, JavaScript, and web graphics.

WEB PROGRAMMING

US $34.99 CAN $40.99
ISBN: 978-1-4919-4312-0

Twitter: @oreillymedia
facebook.com/oreilly

www.allitebooks.com

SECOND EDITION

Learning Node

Moving to the Server Side

Shelley Powers

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

www.allitebooks.com

Learning Node
by Shelley Powers

Copyright © 2016 Shelley Powers. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meg Foley
Production Editor: Shiny Kalapurakkel
Copyeditor: Gillian McGarvey
Proofreader: Rachel Monaghan
Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2016: Second Edition

Revision History for the Second Edition
2016-05-23: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491943120* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Learning Node*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-94312-0

[LSI]

www.allitebooks.com

Table of Contents

www.allitebooks.com

www.allitebooks.com

www.allitebooks.com

www.allitebooks.com

Preface

Node.js has been around long enough to have achieved adoption by some major players (LinkedIn, Yahoo!, and Netflix), but is still young enough to be cause for concern for your typical corporate middle manager. It's been a driving force behind creating a more sophisticated JavaScript, as well as the only safe place one can then use the newly improved scripting language. And, since turnabout is fair play, the cutting-edge JavaScript has now become the driver for a newly revamped Node.js organization and release paradigm.

Node.js has also redefined what we do with JavaScript. Nowadays, an employer is just as likely to demand that JavaScript developers work in a server environment as well as the more familiar browser environment. In addition, Node.js has created a new server language that's generating attention from Ruby, C++, Java, and PHP server-side developers—especially if they also know JavaScript.

To me, Node.js is fun. Compared to so many other environments, it takes little effort to get started, create and host an application, and try out new things. The scaffolding necessary for a Node project just isn't as complex or pedantic as what's required for other environments. Only PHP has as simple an environment, and even it requires tight integration with Apache to create outward-facing applications.

As simple as it is, though, Node.js has bits that can be hard to discover. It is true that learning Node.js requires obtaining a mastery of its environment and the core APIs, but it's also about finding and mastering these hard-to-discover bits.

Who This Book Is For

I see two audiences for this book.

The first audience is the developer who has been creating frontend applications using a variety of libraries and frameworks, and who now wants to take their JavaScript skills to the server.

www.allitebooks.com

The second audience is the server-side developer who wants to try something new or needs to make a shift to a newer technology. They've worked with Java or C++, Ruby or PHP, but now they want to take the JavaScript they've picked up over time, and their knowledge of the server environment, and merge the two.

These are two seemingly separate audiences with one shared knowledge: JavaScript, or ECMAScript if you want to be more precise. This book does require that you are very comfortable working with JavaScript. Another commonality is that both audiences need to learn the same Node basics, including the core Node API.

However, each audience brings a different perspective, and skills, to the learning experience. To enhance the usefulness, I'll work to incorporate both perspectives into the material. For instance, a C++ or Java developer might be interested in creating C++ Node add-ons, which might not be interesting to the frontend developer. At the same time, some concepts like *big-endian* may be very familiar to the server-side developer but unknown to the frontend person. I can't dive too deeply into either viewpoint, but I will work to ensure that all readers don't end up frustrated or bored.

One thing I'm not going to do is force rote memorization on you. We'll get into the core module APIs, but I'm not going to cover every object and function, because these are documented at the Node website. What I'll do instead is touch on important aspects of each core module or specific Node functionality I think is essential in order to, hopefully, give you a baseline so you can hold your own with other Node developers. Of course, practice makes mastery, and this book is a learning tool. Once you're finished with the book you'll need to continue on for more in-depth explorations of specific types of functionality, such as working the Mongo-Express-Angular-Node (MEAN) stack. This book will give you the grounding you need to branch out in any of the many Node directions.

Speaking of Node Documentation

At the time this was written, other Node developers and I were involved in discussions about issues associated with the Node.js website. Among them was defining the "current" version of Node.js, which is what should be documented when one accesses "the" documentation.

When last I joined the discussion, the plan was to list all current long-term support (LTS) versions of Node.js in the */docs* page, as well as the most current Stable release, and to provide an indicator for Node documentation versions on the top of each documentation page.

Eventually, the documentation people would like to generate *version diffs* of the API for each page, but that's going to be a challenging project.

www.allitebooks.com

At the time this book was published, Node released Node.js 6.0.0 as the Current release, and abandoned Stable as the designation for the active development branch. Node.js 6.0.x will eventually transition into the next LTS release.

Because of all these versions, when you do access documentation of APIs at the Node.js website, always check to make sure the documentation matches your version of Node.js. And it doesn't hurt to check newer versions to see what new thing is coming your way.

Node.js Is Node

The formal name is Node.js, but no one uses it. Everyone uses "Node." End of story. And that's what we'll be using, for the most part, in this book.

Book Structure

Learning Node is a back-to-basics book. Its focus is on Node, and the modules that make up the Node core. I do lightly touch on some third-party modules, and provide extensive coverage of npm, of course, but the primary goal of this book is to bring you, the reader, up to speed on basic Node functionality. From this solid platform, you can then move in other directions.

Chapter 1 includes an introduction to Node, including a brief section on how to install it. You'll also get a chance to take Node out for a spin, first by creating the web server listed in the Node documentation, and then by creating a bit more advanced server using the code I provide. In addition, I'll also touch on creating Node add-ons, for the C/C++ coders among you. And what would an introduction to Node be without an overview of the history of this environment, which issued its first release as 4.0 rather than 1.0.

Chapter 2 covers essential Node functionality including how events are handled, the global objects underlying the necessary functionality to create Node applications, and the asynchronous nature of Node. In addition, I also cover the *buffer* object, which is the data structure transmitted via most network services in Node.

Chapter 3 dives into the nature of Node's module system, as well as providing an in-depth look at npm, a Node third-party module management system. In this chapter, you'll discover how your application finds the Node modules you're using, and how you can create a Node module of your own to use. Lastly, I get into the more advanced aspects of Node functionality by exploring the support for sandboxing. Just for fun, I also cover three popular Node third-party modules: Async, Commander, and Underscore.

The interactive console that comes with Node, and which goes by the name of REPL, is an invaluable learning tool and environment, and it gets its own chapter in Chapter 4. I cover how to use the tool, in detail, as well as how to create your own custom REPL.

We explore developing Node applications for the Web in Chapter 5, including taking a much closer look at the Node modules that support web development. You'll get a chance to see what it takes to create a fully functional static web server, as well as learning how you can run a Node application alongside Apache, via the use of an Apache proxy.

Node works in a variety of environments, including both Windows and Unix-based systems such as OS X and Linux. It does so by providing a set of functionality that normalizes system differences, and this functionality is explored in Chapter 6. In addition, I also cover the fundamental nature of Node streams and pipes—essential elements of all input/output—as well as exploring Node's filesystem support.

Chapter 7 is all about networks, and you can't touch on networking without also touching on security. The two should go hand-in-hand, like peanut butter and jelly, or chocolate with anything. I'll introduce you to Node's TCP and UDP support, as well as cover how to implement an HTTPS server, in addition to the HTTP server you learned how to create in Chapter 5. I'll also cover the mechanics behind digital certificates and the fundamentals of Secure Sockets Layer (SSL), and its upgrade, Transport Layer Security (TLS). Lastly, we'll look at the Node crypto module and working with password hashes.

One of my favorite aspects of Node is its ability to work with operating system functionality via *child processes*. Some of my favorite Node applications are small utility programs for working with compressed files, using the popular graphics application, ImageMagick, and creating a screenshot application for grabbing screenshots from websites. They're not big-time applications with sophisticated cloud interfaces, but they are a fun way to learn to work with child processes. Child processes are covered in Chapter 8.

Most of the examples in the book use the same JavaScript you've been using for years. However, one of the main impetuses behind the Node.js/io.js split and the new, merged product is support for the newer ECMAScript versions, such as ES6 (or ECMAScript 2015, if you prefer). In Chapter 9, I cover what's currently supported in Node, the impact of the new functionality, and when and why to use new functionality over old. I also cover the gotchas related to using the newer JavaScript capabilities. The only time I won't focus on the native functionality is when I cover promises as implemented by that very popular module, Bluebird.

Chapter 10 takes a look at the frameworks and functionality that make up what is known as *full-stack Node development*. We'll take a look at Express, a now ubiquitous

component of most Node development, as well as try out both MongoDB and Redis. We'll also explore a couple of the frameworks that make up the "full stack" in full-stack Node Development: AngularJS and Backbone.js.

Once you've coded your Node application, you'll want to push it to production. Chapter 11 covers tools and techniques for Node development and production support, including unit, load, and benchmark testing, as well as essential debugging skills and tools. You'll also learn how to run your Node application forever, and restore it if it fails.

Chapter 12 is dessert. In this chapter, I'll introduce you to the ways of taking your mad Node skills to new worlds, including Node in microcontrollers/microcomputers, as a part of the Internet of Things, and a version of Node that does not run on V8.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/shelleyp/LearningNode2*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning Node* by Shelley Powers (O'Reilly). Copyright 2016 Shelley Powers, 978-1-4919-4312-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kauf-

mann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/learning-node-2e*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I would like to thank the folks who helped me pull this book together: editor, Meg Foley; tech reviewer, Ethan Brown; copyeditor, Gillian McGarvey; proofreader, Rachel Monaghan; indexer, Judy McConville; illustrator, Rebecca Panzer; and anyone else who has touched this book!

CHAPTER 1

The Node Environment

Forget what you've heard about Node being a server-side tool only. Node is primarily used in server applications, true. But Node can be installed on almost any machine and used for any purpose, including running applications on your PC or even your tablet, smartphone, or microcomputer.

I have Node installed on my Linux-based server, but I also have it installed on my Windows 10-based PCs, as well as a microcomputer (Raspberry Pi). I have an extra Android tablet that I'm thinking of trying Node on, can use Node to program my Arduino Uno microcontroller, and am currently playing around with incorporating Node into my smart home setup thanks to IFTTT's Maker Channel. On my PC, I use Node as a test environment for JavaScript, as well as an interface to ImageMagick for batch photo editing. Node is my go-to tool for any batch operation on my PCs or my server.

And yes, I use Node for server-side processing when I want a web interface that bypasses my Apache server, or provides a backend server process for a web application.

The point is, the Node environment is rich in functionality and reach. To start exploring this environment, we have to start at the beginning: installing Node.

IFTTT

IFTTT is a marvelous site that allows you to connect triggers and actions from a host of companies, services, and products using simple if-then logic. Each end point of the equation is a channel, including the aforementioned Maker Channel.

Installing Node

The best place to start when installing Node is the Node.js Downloads page. From here, you can download binaries (precompiled executables) for Windows, OS X, SunOS, Linux, and ARM architectures. The page also provides access to installers for a specific architecture that can greatly simplify the installation process—particularly with Windows. If your environment is set up for building, download the source code and build Node directly. That's my preference for my Ubuntu server.

You can also install Node using a package installer for your architecture. This is a helpful option not only for installing Node, but also for keeping it up-to-date (as we'll discuss further, in the section "The Node LTS and Upgrading Node" on page 13).

If you decide to compile Node directly on your machine, you'll need to set up the proper build environment, and install the proper build tools. For instance, on Ubuntu (Linux), you'll need to run the following command to install the tools needed for Node:

```
apt-get install make g++ libssl-dev git
```

There are some differences in behavior when you first install Node in the various architectures. For instance, when you're installing Node in Windows, the installer not only installs Node, but it also creates a Command window you'll use to access Node on your machine. Node is a command-line application, and doesn't have a graphical UI like the typical Windows application. If you want to use Node to program an Arduino Uno, you'll install Node and Johnny-Five, and use both to program the connected device.

Accept Defaults in Windows World

You'll want to accept the default location and feature installation when installing Node in Windows. The installer adds Node to the PATH variable, which means you can type **node** without having to provide the entire Node installation path.

If you're installing Node on Raspberry Pi, download the appropriate ARM version, such as ARMv6 for the original Raspberry Pi, and ARMv7 for the newer Raspberry Pi 2. Once downloaded, extract the binary from the compressed tarball, and then move the application to */usr/local*:

```
wget https://nodejs.org/dist/v4.0.0/node-v4.0.0-linux-armv7l.tar.gz
tar -xvf node-v4.0.0-linux-armv7l.tar.gz

cd node-v4.0.0-linux-armv7l

sudo cp -R * /usr/local/
```

You can also set up a build environment and build Node directly.

New Node Environments

Speaking of Node with Arduino and Raspberry Pi, I cover Node in nontraditional environments, such as the Internet of Things, in Chapter 12.

Saying Hello to the World with Node

You've just installed Node and you naturally want to take it for a spin. There is a tradition among programmers that the first application in a language is the well-known "Hello, World" application. The application typically writes out the words "Hello, World" to whatever is the output stream, demonstrating how an application is created, run, and can process input and output.

The same holds true for Node: it is the application that the Node.js website includes in the Synopsis in the application's documentation. And it is the first application we'll create in this book, but with a few tweaks.

A Basic Hello World Application

First, let's take a look at the "Hello, World" application included in the Node documentation. To re-create the application, create a text document with the following JavaScript, using your favorite text editing tool. I use Notepad++ in Windows, and Vim in Linux.

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

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

Save the file as *hello.js*. To run the application, open a terminal if you're using OS X or Linux, or the Node Command window with Windows. Change to the directory where your saved file is located and type the following to run the application:

```
node hello.js
```

The result is printed out to the command line, via the `console.log()` function call in the application:

```
Server running at http://127.0.0.1:8124/
```

Now open a browser and type either **`http://localhost:8124/`** or **`http://127.0.0.1:8124`** into the address bar (or your domain, if you're hosting Node on

your server). What appears is a simple unadorned web page with "Hello World" in text at the top, as shown in Figure 1-1.

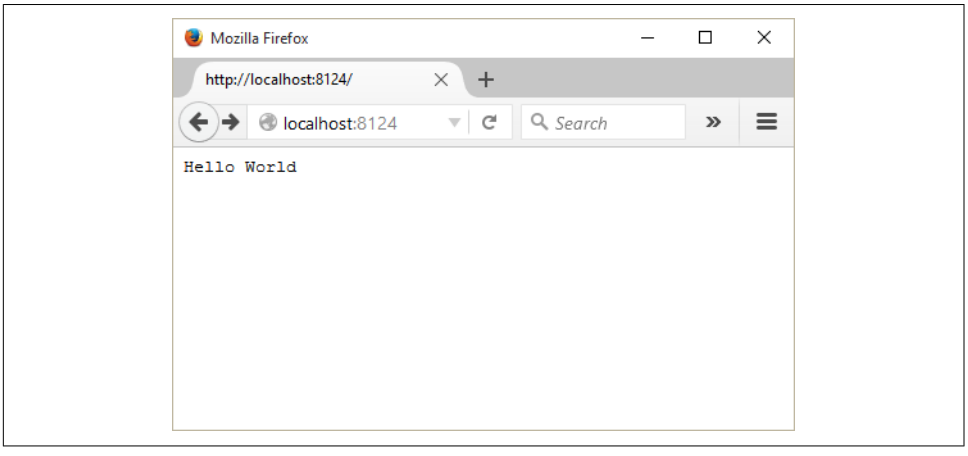


Figure 1-1. Your first Node application

If you're running your application in Windows, you'll most likely receive a Windows Firewall alert, as shown in Figure 1-2. Uncheck the Public Network option, check the Private network option, and then click the button to Allow access.

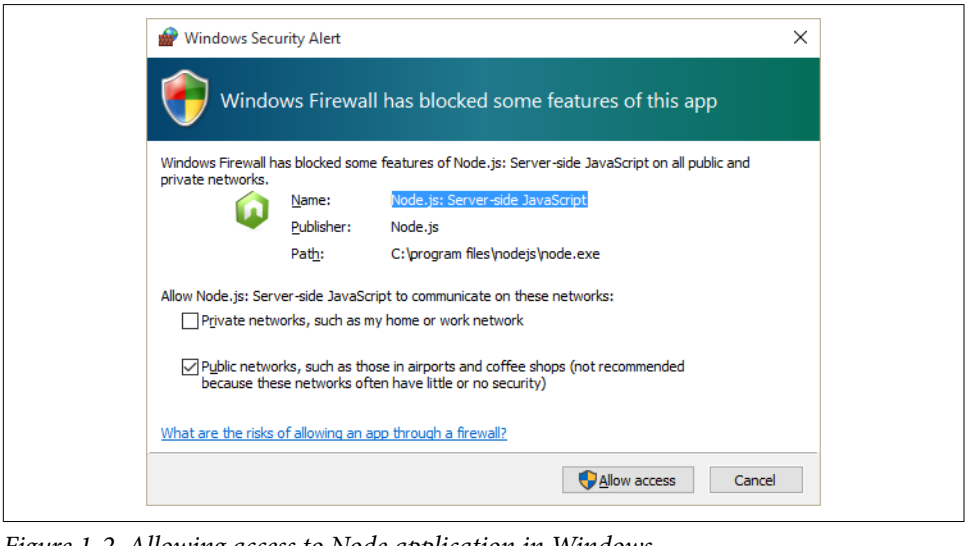


Figure 1-2. Allowing access to Node application in Windows

You won't have to repeat this process in Windows: the system remembers your choice.

To end the program, you can either close the terminal/Command window (just terminal from this point), or type Ctrl-C. When you ran the application, you did so in the foreground. This means you can't type any other command in the terminal. It also means when you closed the terminal, you stopped the Node process.

Running Node Forever

For now, running Node in the foreground is a good thing. You're learning how to use the tool, you don't yet want your applications to be externally available to everyone, and you want it to terminate when you're finished for the day. In Chapter 11, I'll cover how you can create a more robust Node runtime environment.

To return to the Hello World code, JavaScript creates a web server that displays a web page with the words "Hello World" when accessed via a browser. It demonstrates several key components of a Node application.

First, it includes the module necessary to run a simple HTTP server: the appropriately named HTTP module. External functionality for Node is incorporated via modules that export specific types of functionality that can then be used in an application (or another module). They're very similar to the libraries you've used in other languages.

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

Node Modules, Core Modules, and the http Module

The HTTP module is one of Node's core modules, which are the primary focus of this book. I'll cover Node modules and module management thoroughly in Chapter 3, and the HTTP module in Chapter 5.

The module is imported using the Node `require` statement, and the result assigned to a local variable. Once imported, the local variable can be used to instantiate the web server, via the `http.createServer()` function. In the function parameters, we see one of the fundamental constructs of Node: the *callback* (Example 1-1). It's the anonymous function that's passing the web request and response into the code to process the web request and provide a response.

Example 1-1. Hello, World callback function

```
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);
```

JavaScript is single-threaded, and the way Node emulates an asynchronous environment in a single-threaded environment is via an *event loop*, with associated *callback* functions that are called once a specific event has been triggered. In Example 1-1, when a web request is received, the callback function is called.

The `console.log()` message is output to the terminal as soon as the call to create the server is made. The program doesn't stop and block, waiting for a web request to be made.

```
console.log('Server running at http://127.0.0.1:8124/');
```

More on the Event Loop and Callback Function

I'll be covering the Node event loop, its support for asynchronous programming, and the callback function in more detail in Chapter 2.

Once the server is created and has received a request, the callback function writes a plain text header with server status of `200` back to the browser, writes out the Hello World message, and then ends the response.

Congratulations, you've created your first web server in Node in just a few lines of code. Of course, it's not particularly useful, unless your only interest is in greeting the world. Throughout the book you'll learn how to make more useful Node applications, but before we leave Hello World, let's make some modifications to the basic application to make it a little more interesting.

Hello World, Tweaked

Just printing out a static message does demonstrate, first of all, that the application is working and, second, how to create a simple web server. The basic example also demonstrated several key elements of a Node application. But it could be just a little richer, a little more fun to play with. So I tweaked it to provide you a second application you can try out that has a few more moving parts.

The tweaked code is in Example 1-2. In it, I modified the basic application to parse the incoming request to look for a query string. The name in the string is extracted and used to determine the type of content returned. Almost any name will return a personalized response, but if you use `name=burningbird` as the query, you'll get an image. If no query string is used, or no name passed, the name variable is set to '`world`'.

Example 1-2. Hello World, tweaked

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

http.createServer(function (req, res) {
   var name = require('url').parse(req.url, true).query.name;

   if (name === undefined) name = 'world';

   if (name == 'burningbird') {
      var file = 'phoenix5a.png';
      fs.stat(file, function (err, stat) {
         if (err) {
            console.error(err);
            res.writeHead(200, {'Content-Type': 'text/plain'});
            res.end("Sorry, Burningbird isn't around right now \n");
         } else {
            var img = fs.readFileSync(file);
            res.contentType = 'image/png';
            res.contentLength = stat.size;
            res.end(img, 'binary');
         }
      });
   } else {
      res.writeHead(200, {'Content-Type': 'text/plain'});
      res.end('Hello ' + name + '\n');
   }
}).listen(8124);

console.log('Server running at port 8124/');
```

The result of accessing the web-based application with a query string of `?name=burningbird` is shown in Figure 1-3.

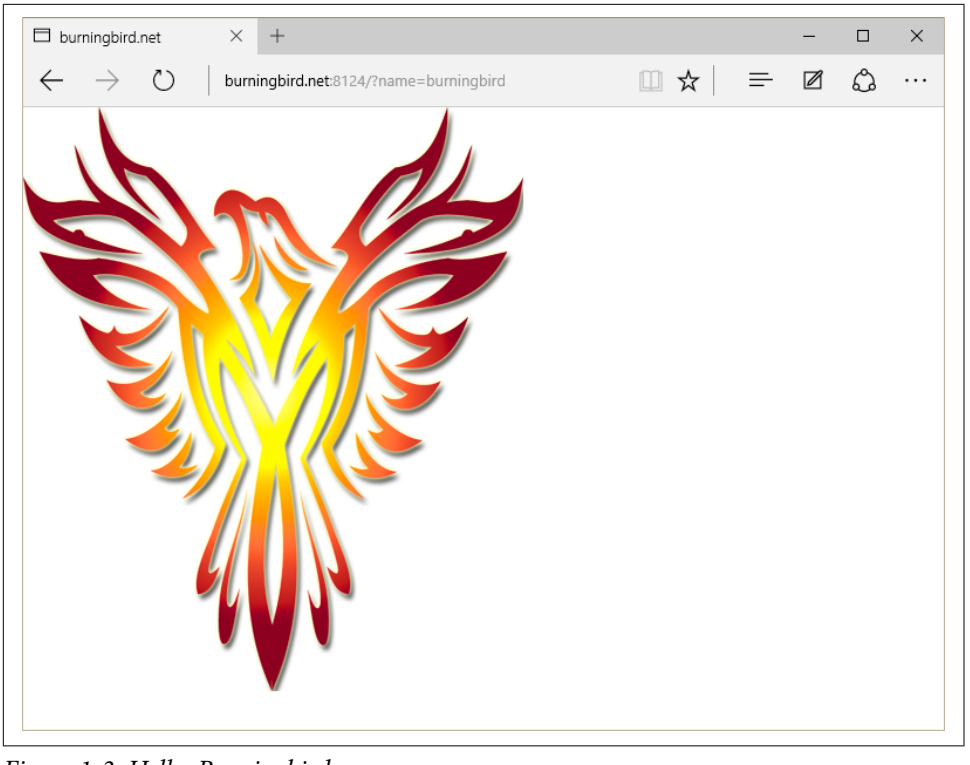


Figure 1-3. Hello, Burningbird

Not much extra code, but there are several differences between the basic Hello World application and the tweaked version. From the top, a new module is included in the application, named `fs`. This is the File System module, one you will become very familiar with in the next several chapters. But there's also another module imported, but not in the same way as the other two:

```
var name = require('url').parse(req.url, true).query.name;
```

Exported module properties can be chained, so we can both import the module and use its functions in the same line. This frequently happens with the URL module, whose only purpose is to provide a URL utility.

The response and request parameter variable names are shortened to `res` and `req` to make them easier to access. Once we parse out the request to get the `name` value, we first test to see if it's `undefined`. If not, the value is set to the fallback result, `world`. If `name` does exist, it's tested again to see if it's equal to `burningbird`. If it isn't, then the response is close to what we had in the basic application, except for inserting the supplied name into the return message.

If the name is `burningbird`, though, we're dealing with an image rather than text. The `fs.stat()` method not only verifies that the file exists but also returns an object with information about the file, including its size. This value is used in creating the content header.

If the file doesn't exist, the application handles the situation gracefully: it issues a friendly message that the bird has flown the coop, but also provides error information at the console, using the `console.error()` method this time:

```
{ [Error: ENOENT: no such file or directory, stat 'phoenix5a.png']
  errno: -2,
  code: 'ENOENT',
  syscall: 'stat',
  path: 'phoenix5a.png' }
```

If the file does exist, then we'll read the image into a variable and return it in the response, adjusting the header values accordingly.

The `fs.stats()` method uses the standard Node callback function pattern with the error value as the first parameter—frequently called an *errback*. However, the part about reading the image may have you scratching your head. It doesn't look right, not like other Node functions you've seen in this chapter (and most likely in other online examples). What's different is that I'm using a synchronous function, `readFileSync()`, rather than the asynchronous version, `readFile()`.

Node does support both synchronous and asynchronous versions of most File System functions. Normally, using a synchronous operation in a web request in Node is taboo, but the capability is there. An asynchronous version of the same bit of code is used in Example 1-3.

```
fs.readFile(file, function(err,data) {
                res.contentType = 'image/png';
                res.contentLength = stat.size;
                res.end(data, 'binary');
              });
```

Why use one over the other? In some circumstances, file I/O may not impact performance regardless of which type of function you use, and the synchronous version can be cleaner and easier to use. It can also lead to less nested code—a particular problem with Node's callback system, and one I'll cover in more detail in Chapter 2.

Additionally, though I don't use exception handling in the example, you can use `try...catch` with synchronous functions. You can't use this more traditional error handling with asynchronous functions (hence the error value as the first parameter to the anonymous callback function).

The important fact to take away from this second example, though, is that not all Node I/O is asynchronous.

Filesystem and URL Modules, Buffers, and Aynchronous I/O

I'll cover the URL module in more detail in Chapter 5, and File System is explored in Chapter 6. Note, though, that File System is used all throughout the book. Buffers and asynchronous processing are covered in Chapter 2.

Node Command-Line Options

In the last two sections, Node is invoked at the command line without the use of any command-line options. I wanted to briefly introduce some of these options before we move on. Others will be introduced when needed throughout the book.

To discover all the available options and arguments, use the help option, written as either `-h` or `--help`:

```
$ node --help
```

This option will list out all of the options, and provides the syntax to follow when running the Node application:

```
Usage: node [options] [ -e script | script.js ] [arguments]
       node debug script.js [arguments]
```

To find the version of Node, use the following command:

```
$ node -v or --version
```

To check the syntax of a Node application, use the `-c` option. This checks the syntax without running the application:

```
$ node -c or --check script.js
```

To discover the V8 options, type the following:

```
$ node --v8-options
```

This returns several different options, including the `--harmony` option, used to enable all completed Harmony JavaScript features. This includes all ES6 functionality that's been implemented but not yet staged into either the LTS or Current release.

A favorite Node option of mine is `-p` or `--print`, which can evaluate a line of Node script and print the results. This is especially helpful if you're checking out the Process environmental properties, discussed more fully in Chapter 2. An example is the following, which prints out all of the values for the `process.env` property:

```
$ node -p "process.env"
```

Node Hosting Environments

As you learn Node, you'll want to become familiar with it in your own local environment, be it Windows, OS X, or Linux. When you're ready to start providing greater access to your applications, you'll either need to find an environment where you can run a Node application, such as a virtual private network (VPN), which is what I use, or a host that specifically provides for Node application support. The former requires that you have significant experience running a Internet-faced server, while the latter may limit what you can or cannot do with your Node application.

Hosting Node on Your Server, VPS, or Managed Host

Hosting Node on the same server as your WordPress weblog is likely going to be a dead end, because of Node's requirements. You don't *have* to have root or administrative access to run Node, but you should. In addition, many hosting companies are not going to be happy with you hosting an application on various ports that may or may not play havoc with their systems.

Hosting Node on a virtual private server (VPS), like my VPN at Linode, is a simple matter. You do have root access for your VPS and can pretty much do whatever you want, as long as you don't endanger other users who may be on the same machine. Most companies that provide VPSs ensure that each individual account is isolated from the others, and that no one account can hog all the available resources either.

The issue, though, with a VPS is the same issue you'd have if you hosted your own server: you have to maintain the server. This includes setting up an email system and an alternative web server, most likely Apache or Nginx, to handle firewalls and other security, email, etc. It's not trivial.

Still, if you're comfortable with managing all aspects of a Internet-faced environment, a VPS can be an affordable option for hosting a Node application. At least until you're ready to push it into production, in which case you may want to look at hosting the application in the *cloud*.

Cloud Hosting

Nowadays, an application is just as likely to reside in a cloud server as it is on an individual's or group's own computers. Node applications are well-suited to cloud-based implementations.

When you host a Node application in the cloud, what you're really doing, typically, is creating the application on your own server or PC, testing it, making sure it's what you want, and then pushing the application out to the cloud server. A cloud server for Node allows you to create the Node application you want to create, using the resources of whatever database system or other system you wish, but without having to

manage the server directly. You can focus specifically on the Node application without having to worry about FTP or email servers, or general server maintenance.

Git and GitHub: Prerequisites for Node Development

If you've not used the Git source code control system, you'll need to install it into your environments and learn how to use it. Almost all transitioning of Node applications, including pushing the applications to a cloud server, happens through Git.

Git is open source, freely available, and easy to install. You can access the software at the Git website. There's also an interactive guide you can use to learn the basic Git commands, at GitHub.

Speaking of Git, where there's Git, there's typically GitHub. The Node.js source is maintained at GitHub, as are most, if not all, of the available Node modules. The source for the examples for this book is available at GitHub.

GitHub is probably the largest repository of open source code in the world. It's definitely the center of the universe for the Node world. It is a commercial enterprise, but free to most users. The GitHub organization provides excellent documentation for how to use the site, and there are books and other tutorials you can access to get up to speed with both Git and GitHub. Among them is a free, online book on Git, as well as Loeliger and McCullough's *Version Control with Git* (O'Reilly), and Bell and Beer's *Introducing GitHub* (O'Reilly).

The paradigm for hosting a Node application in the cloud is fairly similar across all of the hosts. First, create the Node application, either locally or on your own server. When you're ready to start testing a deployment environment, then it's time to look for a cloud server. For most I'm familiar with, you sign up for an account, create a new project, and specify that it is Node-based if the cloud is capable of hosting many environments. You may or may not need to specify which other resources are needed, such as database access.

Once ready to deploy, you'll push the application to the cloud. You'll either use Git to push the application or you may need to use a toolset supplied by the cloud provider. As an example, Microsoft's Azure cloud utilizes Git to push the application from your local environment to the cloud, while Google's Cloud Platform provides a toolset to perform the same process.

Finding a Host

Though not completely up-to-date, a good place to start looking for a Node host is a GitHub page devoted to the topic.

The Node LTS and Upgrading Node

In 2014, the Node world was stunned (or at least some of us were taken by surprise) when a group of Node maintainers split from the rest and formed their own fork of Node.js called io.js. The reason for the split is that the io.js folks felt that Joyent, the company that had been maintaining Node, wasn't moving fast enough to open governance of Node. The group also felt that Joyent was behind in its support of the latest V8 engine updates.

Thankfully, I can report that the two groups resolved the issues leading to the split, merging their efforts back to one product, still named Node.js. Node is now managed through a governing nonprofit, the Node Foundation, under the auspices of the Linux Foundation. As a result, the codebase for both groups was combined, and rather than the first official release of Node being Node 1.0, it became Node 4.0: representing Node's original slow pace to Node 1.0, and io.js's faster 3.0 version.

Node's New Semantic Versioning

One result of the merge is a strict timeline of Node releases, based on semantic versioning (Semver), familiar to Linux users. Semver releases features as three groups of numbers, each with a specific meaning. For instance, as I'm writing this, I'm currently using Node.js version 4.3.2 on my server. This translates to:

- The major release is 4. This number will only increase when a major, backward-incompatible change is made to Node.
- The minor release is 3. This number increases when new functionality is added, but the functionality is still backward compatible.
- The patch release is 2. This number changes when security or other bug fixes require a new release of the application. It is also backward compatible.

I'm using the Stable release of 5.7.1 on my Windows machine, and tested the code using the Current release of 6.0.0 on a Linux machine.

The Node Foundation also supports a much more cohesive, albeit somewhat problematic, release cycle than the hit-or-miss releases we have become familiar with. It started with the first LTS (Long-Term Support) release of Node.js v4, which will be supported until April 2018. The Node Foundation then released its first Stable release, Node.js v5, at the end of October 2015. Node 5.x.x will only be supported until April 2016, when it will be replaced by Node.js v6. The strategy is for a new Stable (now Current) release every six months, but only every other one goes LTS, like Node v4.

Release of 6.0.0 as Current

In April 2016, Node released 6.0.0, which replaces 5.x, and will transition into the new LTS in October of 2016. Node also renamed the Stable designation for the active development branch to Current.

After April 2018, Node v4 enters maintenance mode. In the meantime, there will be new backward-compatible updates (known as *semver-major bumps*), as well as bug and security patches.

What Version Is the Book Covering?

This book covers the long-term stable release of Node.js v4. Annotations mark differences between v4 and v5/v6, wherever appropriate.

Regardless of which LTS major release you decide to use, you'll need to upgrade to each new bug/security fix as soon as it releases. However you handle each new semver-major bump is up to you and/or your organization. The upgrade should be backward compatible, though, with only underlying engine improvements impacted. Still, you'll want to incorporate any new release into an upgrade and testing plan.

Which version should you use? In a business or corporate environment, you'll most likely want to stick with the LTS release, which is, at this time, Node.js v4. However, if your environment can more quickly adapt to breaking changes, you can get access to the latest v8 and other goodies with the latest Node Current release.

The Joys of Testing and Production

I cover Node debugging and testing, as well as other development process and production procedures, in Chapter 11.

Upgrading Node

With the increased schedule of releases, keeping Node up-to-date is even more critical. Thankfully, the upgrade process is painless, and you have alternatives.

You can check your version with the following:

```
node -v
```

If you're using a package installer, then running the package update procedure updates Node, as well as any other software on your server (`sudo` is not required in Windows):

www.allitebooks.com

```
sudo apt-get update
sudo apt-get upgrade --show-upgraded
```

If you are using a package installer, follow the instructions associated with it that are provided at the Node website. Otherwise, you'll end up out-of-sync with releases.

You can also use npm to upgrade Node, using the following sequence of commands:

```
sudo npm cache clean -f
sudo npm install -g
sudo n stable
```

To install the latest version of Node on Windows, OS X, or your Raspberry Pi, grab the installer from the Node.js downloads and run it. It installs the new version over the old.

Node Version Manager

In a Linux or OS X environment, you can also use the Node Version Manager (nvm) tool to keep Node up-to-date.

The Node package manager (npm) updates more frequently than Node. To upgrade just it, run the following command:

```
sudo npm install npm -g n
```

This command installs the latest version of the necessary application. You can check the version using:

```
npm -v
```

Be aware, though, that this can cause issues, especially in a team environment. If your team members are using the version of npm that's installed with Node, and you've manually upgraded npm to the newer version, you can have inconsistent build results that may not be easy to discover.

I'll cover npm in more detail in Chapter 3, but for now, note that you can keep all Node modules up-to-date with the following command:

```
sudo npm update -g
```

Node, V8, and ES6

Behind Node is a JavaScript engine. For most implementations, the engine is V8. Originally created by Google for Chrome, the V8 source code was open-sourced in 2008. The V8 JavaScript engine was created to improve the speed of JavaScript by incorporating a just-in-time (JIT) compiler that compiles JavaScript to machine code rather than interpreting it, which had been the norm for JavaScript for years. The V8 engine is written in C++.

Microsoft's Node.js Fork

Microsoft forked Node to create a version that uses its JavaScript engine, Chakra, specifically to power its vision for the Internet of Things (IoT). I'll cover this fork in more detail in Chapter 12.

When Node v4.0 released, it did so with support for V8 4.5, the same version of the engine being used by Chrome. The Node maintainers are also committed to supporting upcoming versions of V8 as they're released. This means that Node now incorporates support for many of the new ECMA-262 (ECMAScript 2015 or ES6) features.

Node v6 V8 Support

Node v6 supports V8 version 5.0, and new releases of Node will support newer versions of V8 accordingly.

In prior versions of Node, to access the new ES6 features, you would have to use the harmony flag (`--harmony`) when running the application:

```
node --harmony app.js
```

Now, ES6 feature support is based on the following criteria (directly from the Node.js documentation):

- All *shipping* features, which V8 considers stable, are turned on by default on Node.js and do *not* require any kind of runtime flag.
- *Staged features*, which are almost-completed features that are not considered stable by the V8 team, require a runtime flag: `--es_staging` (or its synonym, `--harmony`).
- *In-progress* features can be activated individually by their respective harmony flag (e.g., `--harmony_destructuring`), although this is highly discouraged unless for testing purposes.

I'll cover the ES6 support in Node and how to effectively use the different features in Chapter 9. For now, know that the following are *some* of the ES6 features supported in Node, straight out of the can:

- Classes
- Promises
- Symbols
- Arrow functions
- Generators
- Collections
- `let`
- The spread operator

Advanced: Node C/C++ Add-ons

Now that Node is installed and you've had a chance to play around with it a bit, you might be wondering exactly what it is you installed.

Though the language used to create Node applications is based in JavaScript, much of Node is actually written in C++. Normally this information is behind the scenes in most applications we use, but if you're familiar with C or C++, you can choose to extend Node functionality using C/C++ to create an *add-on*.

Writing a Node add-on is not the same as writing a more traditional C/C++ application. For one, there are libraries, such as the V8 library, that you'll typically access. For another, the Node add-on is not compiled using the tools you would normally use.

The Node documentation for add-ons provides a Hello World example of an add-on. You can check out the code for the short example, which should be familiar if you have programmed with C/C++. Once you've written the code, though, you'll need to use a tool, `node-gyp`, to actually compile the add-on into a *.node* file.

First, a configuration file named *binding.gyp* is created. It uses a JSON-like format to provide information about the add-on:

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "hello.cc" ]
    }
  ]
}
```

The Node add-on configuration step is performed using the following command:

```
node-gyp configure
```

It creates the appropriate configuration file (a Makefile for Unix, a vcxproj file in Windows) and places it in the *build/* directory. To build the Node add-on, run the following command:

```
node-gyp build
```

The compiled add-on is installed in the *build/release* directory, and is now ready to use. You can import it into your application like you would many of the others installed with Node (covered in Chapter 3).

Maintaining Native Mode Modules

Though outside the scope of this book, if you're interested in creating native mode modules (the add-ons), you need to be aware of platform differences. For instance, Microsoft provides special instructions for native modules in Azure, and the maintainer for the popular *node-serialport* native module has detailed the challenges he's faced maintaining the module.

Of course, if you're not familiar with C/C++, you'll most likely want to create modules using JavaScript, and I'll cover that in Chapter 3 also. But if you do know these languages, an add-on can be an effective extension, especially for system-specific needs.

One thing to be aware of is the rather dramatic changes that have occurred within Node as it has progressed from v0.8 through the new v6.x. To counter the problems that can occur, you'll need to install NAN, or Native Abstractions for Node.js. This header file helps to smooth out the differences between versions of Node.js.

CHAPTER 2

Node Building Blocks: Global Objects, Events, and Node's Asynchronous Nature

Though both are built on JavaScript, the environments between browser-based applications and Node.js applications are very different. One fundamental difference between Node and its browser-based JavaScript cousin is the *buffer* for binary data. True, Node does now have access to the ES6 ArrayBuffer and *typed arrays*. However, most binary data functionality in Node is implemented with the `Buffer` class.

The `buffer` is one of Node's global objects. Another global object is `global` itself, though the global object in Node is fundamentally different than the global object we're used to in the browser. Node developers also have access to another global object, `process`, which provides a bridge between the Node application and its environment.

Thankfully, one aspect of Node should be familiar to frontend developers, and that's its event-driven asynchronous nature. The difference in Node is that we're waiting for files to open rather than for users to click a button.

Event-driven also means those old friends, the timer functions, are available in Node.

Modules and Console

I'll cover several other global components—`require`, `exports`, `module`, and `console`— later in the book. I cover the `require`, `exports`, and `module` globals in Chapter 3, and the `console` in Chapter 4.

The global and process Objects

Two fundamental objects in Node are the `global` and `process` objects. The `global` object is somewhat similar to the global object in the browser, with some very major differences. The `process` object, however, is pure Node all the way.

The global Object

In the browser, when you declare a variable at the top level, it's declared globally. It doesn't work that way in Node. When you declare a variable in a module or application in Node, the variable isn't globally available; it's restricted to the module or application. So you can declare a "global" variable named `str` in a module and also in the application that uses the module, and there won't be any conflict.

To demonstrate, we'll create a simple function that adds a number to a base and returns the result. We'll create it as a JavaScript library to use in a web page, and as a module to use in a Node application.

The code for the JavaScript library, added to a file named *add2.js*, declares a `base` variable, sets it to the value of 2, and then adds it to whatever number is passed to it:

```
var base = 2;

function addtwo(input) {
   return parseInt(input) + base;
}
```

Next, we'll create a very simple module that does the same thing, except using Node module syntax. I'll cover the specifics of the `module` in Chapter 3, but for now, copy the following code into a file named *addtwo.js*:

```
var base = 2;

exports.addtwo = function(input) {
  return parseInt(input) + base;
};
```

Now to demonstrate the differences in `global` in both environments. The *add2.js* library is used in a web page, which also declares a `base` variable:

```
<!DOCTYPE html>
<html>
   <head>
      <script src="add2.js"></script>
      <script>

         var base = 10;
         console.log(addtwo(10));
      </script>
   </head>
```

```
<body>

</body>
</html>
```

Accessing the web page with a browser displays a value of 20, rather than the expected 12, in the browser console. The reason is that all variables declared outside a function in JavaScript in the browser are added to the same global object. When we declared a new variable named `base` in the web page, we overrode the value in the included script file.

Now, we'll use the `addtwo` module in a Node application:

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

var base = 10;

console.log(addtwo(base));
```

The result of the Node application is 12. Declaring the new `base` variable in the Node application had no impact on the value of `base` in the module, because they both exist in different global namespaces.

Eliminating a shared namespace is a distinct improvement, but it isn't universal. What `global` does share in all environments is access to all the globally available Node objects and functions, including the `process` object, covered next. You can check it out for yourself by adding the following to a file and running the application. It prints out all the globally available objects and functions:

```
console.log(global);
```

The process Object

The `process` object is an essential component of the Node environment, as it provides information about the runtime environment. In addition, standard input/output (I/O) occurs through `process`, you can gracefully terminate a Node application, and you can even signal when the Node *event loop* (covered in the section "The Event Queue (Loop)" on page 33) has finished a cycle.

The `process` object is featured in many applications throughout the book, so check the index for all appearances. For now, we'll take a closer look at the `process` object's environmental reporting, as well as the all-important standard I/O.

The `process` object provides access to information about the Node environment, as well as the runtime environment. To explore, we'll use the `-p` command-line option with `node`, which executes a script and returns an immediate result. For instance, to check out the `process.versions` property, type the following:

```
$ node -p "process.versions"
{ http_parser: '2.5.0',
  node: '4.2.1',
  v8: '4.5.103.35',
  uv: '1.7.5',
  zlib: '1.2.8',
  ares: '1.10.1-DEV',
  icu: '56.1',
  modules: '46',
  openssl: '1.0.2d' }
```

Single or Double Quotes for Command Line

Note the use of double quotes, which are required in the Windows command window. Since double quotes work for all environments, use double quotes for all scripts.

Versions for various Node components and dependencies are listed, including the version of V8, OpenSSL (the library used for secure communications), Node itself, and so on.

The `process.env` property provides a great deal of information about what Node sees in your development/production environment:

```
$ node -p "process.env"
```

It's particularly interesting to see the differences between architectures, such as Linux and Windows.

To explore the `process.release` values, use the following:

```
$ node -p "process.release"
```

What you'll get depends on what you have installed. In both LTS and Current environments, you'll get the name of the application, as well as URLs for the source code. But in LTS, you'll have an additional property:

```
$ node -p "process.release.lts"
'Argon'
```

However, if you access this same value in a Current release, such as v6, you'll get a different result:

```
$ node -p "process.release.lts"
undefined
```

The environmental information is a way for you, as the developer, to understand what Node sees before and during development. However, don't include a dependency on most of the data directly into your application because, as you've seen, it may not be consistent between Node releases. However, do take time to explore the data.

What should be consistent between Node releases are several objects and functions essential for many applications. Among them is access to the standard I/O, and the ability to gracefully end a Node application.

Standard streams are pre-established communication channels between an application and the environment. They consist of a standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`). In a Node application, these channels provide communication between the Node application and the terminal. They're a way for you to communicate directly with the application.

Node supports the channels with three process functions:

- `process.stdin`: a readable stream for `stdin`
- `process.stdout`: a writable stream for `stdout`
- `process.stderr`: a writable stream for `stderr`

You can't close these streams, or end them within the application, but you can receive input from the `stdin` channel, and write to the `stdout` and `stderr` channels.

The `process` I/O functions inherit from `EventEmitter`, explored in the section "EventEmitter" on page 39, which means they can emit events, and you can capture these events and process any data. To process incoming data using `process.stdin`, you first of all need to set the encoding for the stream. If you don't, you'll get the results as a buffer rather than a string:

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

Next we'll listen for the `readable` event, which lets us know there's a chunk of data, ready to be read. We'll then use the `process.stdin.read()` function to read this data in, and if the data isn't `null`, echo it back out to `process.stdout`, using the `process.stdout.write()` function:

```
process.stdin.on('readable', function() {
   var input = process.stdin.read();

   if (input !== null) {
      // echo the text
      process.stdout.write(input);
   }
});
```

Now, we could forgo setting the encoding, and get the same results—we'll read a buffer in and write a buffer out—but to the user of the application, it looks like we're working with text (strings). We're not, though. And the next `process` function we'll explore demonstrates this difference.

In Chapter 1, we created a very basic web server that listened for a request and printed out a message. To end the program, you either had to kill the process via sig-

nal, or using Ctrl-C. You can, instead, terminate an application as part of the application using `process.exit()`. You can even signal whether the application terminated successfully or if a failure occurred.

We'll modify the simple I/O test application to "listen" for an exit string, and then exit the program when it occurs. Example 2-1 has the complete application.

Example 2-1. Demonstrating standard I/O in Node, and exiting application

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

process.stdin.on('readable', function() {
   var input = process.stdin.read();

   if (input !== null) {
      // echo the text
      process.stdout.write(input);

      var command = input.trim();
      if (command == 'exit')
         process.exit(0);

   }
});
```

When we run the application, any string we type in is immediately echoed out. And if we type **`exit`**, the application exits without having to use Ctrl-C.

If we remove the `process.stdin.setEncoding()` function call at the beginning, the application will fail. The reason is there is no `trim()` function on a buffer. We could convert the buffer to a string, and then run `trim`:

```
var command = input.toString().trim();
```

But the better approach is to add the encoding and remove any unexpected side effects.

The Stream Interface

The process I/O objects are implementations of the Stream interface, covered with the other system modules in Chapter 6.

The `process.stderr` object is just like it sounds: you can write to it when an error occurs. Why would you use it instead of `process.stdout`? For the same reason the `stderr` channel was created: to differentiate between outputs that are expected versus outputs that note a problem has occurred. In some systems, you can even process

`stderr` outputs differently than `stdout` outputs (such as redirecting a `stdout` message to a logfile, whereas `stderr` goes to the console).

There are several other objects and useful functions associated with `process`, and as I noted earlier, we'll be seeing many of them throughout the book.

Buffers, Typed Arrays, and Strings

In browser-based JavaScript, in the early days, there never was a need to handle binary data (an *octet stream*). Originally, JavaScript was meant to deal with string values accessed or output to alert windows or forms. Even when Ajax changed the game, the data between client and server was string (Unicode) based.

Things changed, though, when the demands on JavaScript became more sophisticated. Not only do we still have Ajax, but we also have WebSockets. In addition, what we can do in the browser has expanded—rather than simple form access, we have new technologies such as WebGL and Canvas.

The solution in JavaScript and in the browser is the `ArrayBuffer`, manipulated through *typed arrays*. In Node, the solution is the `Buffer`.

Originally, the two were not the same. However, when io.js and Node.js merged in Node v4.0.0, Node also received support for typed arrays via V8 v4.5. The Node buffer is now backed by `Uint8Array`, one of the typed arrays representing an array of 8-bit unsigned integers. That doesn't mean, though, that you can use one in place of the other. In Node, the `Buffer` class is the primary data structure used with most I/O, and you can't swap in a typed array without your application failing. In addition, converting a Node buffer to a typed array may be doable, but it's not without issues. According to the `Buffer` API documentation, when you "convert" a buffer to a typed array:

- The buffer's memory is copied, not shared.
- The buffer's memory is interpreted as an array, not a byte array. That is, new `Uint32Array(new Buffer([1,2,3,4]))` creates a four-element `Uint32Array` with elements `[1,2,3,4]`, not a `Uint32Array` with a single element `[0x1020304]` or `[0x4030201]`.

So you can use both types of octet stream handling in Node, but for the most part, you're primarily using buffer. So, what is a Node buffer?

What Is an Octet Stream?

Why is a binary or raw data file referred to as an octet stream? An octet is a unit of measurement in computing. It's 8 bits long, hence the "octet." In a system that supports 8-bit bytes, an octet and a byte are the same thing. A stream is just a sequence of data. Therefore, a binary file is a sequence of octets.

A Node buffer is raw binary data that's been allocated outside the V8 heap. It's managed via a class, the `Buffer`. Once allocated, the buffer can't be resized.

The buffer is the default data type for file access: unless a specific encoding is provided when reading and writing to a file, the data is read into, or out of, a buffer.

In Node v4, you can create a new buffer directly using the `new` keyword:

```
let buf = new Buffer(24);
```

Just be aware that, unlike `ArrayBuffer`, creating a new Node buffer doesn't initialize the contents. If you want to ensure you don't get pesky, unexpected consequences with working with a buffer that may or may not contain all sorts of peculiar and possibly sensitive data, you'll also want to fill the buffer as soon as you create it:

```
let buf = new Buffer(24);
buf.fill(0); // fills buffer with zeros
```

You can also partially fill a buffer, specifying a start and end value.

Specify Encoding for Buffer Fill

Beginning with Node v5.7.0, you can also specify an encoding for `buf.fill()`, using the syntax: `buf.fill(string[, start[, end]] [, encoding])`.

You can also directly create a new buffer by passing the constructor function an array of octets, another buffer, or a string. The buffer is created with the copied contents of all three. For the string, if it isn't UTF-8, you'll need to specify the encoding; strings in Node are encoded as UTF-8 (`utf8` or `utf-8`) by default.

```
let str = 'New String';
let buf = new Buffer(str);
```

I don't want to cover every method for the `Buffer` class, since Node provides comprehensive documentation. But I did want to look more closely at some of the functionality.

Node v4 to Node v5/v6 Differences

Both the `raw` and `raws` encoding types have been removed in Node v5 and later.

In Node v6, however, the constructors have been deprecated in favor of new Buffer methods to create new buffers: Buffer.from(), Buffer.alloc(), and Buffer.allocUnsafe().

With the Buffer.from() function, passing in an array returns a buffer with a copy of the contents. However, if you pass in an ArrayBuffer, with optional byte offset and length, the buffer shares the same memory as the ArrayBuffer. Passing in a buffer copies the contents of the buffer, and passing in a string copies the string.

The `Buffer.alloc()` function creates a filled buffer of a certain size, while `Buffer.allocUnsafe()`, creates a buffer of a certain size, but may contain old or sensitive data, and will then need to be filled, with `buf.fill()` to ensure otherwise.

The following Node code:

```
'use strict';

let a = [1,2,3];

let b = Buffer.from(a);

console.log(b);

let a2 = new Uint8Array([1,2,3]);

let b2 = Buffer.from(a2);

console.log(b2);

let b3 = Buffer.alloc(10);

console.log(b3);

let b4 = Buffer.allocUnsafe(10);

console.log(b4);
```

Results in the following, on my system:

```
<Buffer 01 02 03>
<Buffer 01 02 03>
<Buffer 00 00 00 00 00 00 00 00 00 00>
<Buffer a0 64 a3 03 00 00 00 00 01 00>
```

Note the data artifacts associated using Buffer.allocUnsafe(), compared to Buffer.alloc();

Buffer, JSON, StringDecoder, and UTF-8 Strings

Buffers can convert to JSON, as well as strings. To demonstrate, type the following into a Node file and run it at the command line:

```
"use strict";

let buf = new Buffer('This is my pretty example');
let json = JSON.stringify(buf);

console.log(json);
```

The result is:

```
{"type":"Buffer",
"data":[84,104,105,115,32,105,115,32,109,121,32,112,114,101,116,
116,121,32,101,120,97,109,112,108,101]}
```

The JSON specifies that the type of object being transformed is a `Buffer`, and its data follows. Of course, what we're seeing is the data after it's been stored in a buffer as a sequence of octets, which aren't human-readable.

ES6

Most of the code examples use very familiar JavaScript that's been around several years. However, I do sneak in ES6 from time to time, and I'll cover Node and ES6 (EcmaScript 2015) in more detail in Chapter 9.

To go full circle, we can parse the buffer data back out of the JSON object, and then use the `Buffer.toString()` method to convert to a string, as shown in Example 2-2.

Example 2-2. A string to buffer to JSON, and back to buffer and back to string

```
"use strict";

let buf = new Buffer('This is my pretty example');
let json = JSON.stringify(buf);

let buf2 = new Buffer(JSON.parse(json).data);

console.log(buf2.toString()); // this is my pretty example
```

The `console.log()` function prints out the original string after it's been converted from buffer data. The `toString()` function converts the string to UTF-8 by default, but if we wanted other string types, we'd pass in the encoding:

```
console.log(buf2.toString('ascii')); // this is my pretty example
```

We can also specify a starting and ending place in the string conversion:

```
console.log(buf2.toString('utf8', 11,17)); // pretty
```

Using `Buffer.toString()` isn't the only way we can convert a buffer to a string. We can also use a helper class, `StringDecoder`. This object's sole purpose is to decode buffer values to UTF-8 strings, but it does so with a little more flexibility and recoverability. If the `buffer.toString()` method gets an incomplete UTF-8 character sequence, it returns gibberish. The `StringDecoder`, on the other hand, buffers the incomplete sequence until it's complete and then returns the result. If you're receiving a UTF-8 result as chunks in a stream, you should use `StringDecoder`.

An example of the differences between the string conversion routines is in the following Node application. The euro symbol (€) is coded as three octets, but the first buffer only contains the first two octets. The second buffer contains the third.

```
"use strict";

let StringDecoder = require('string_decoder').StringDecoder;
let decoder = new StringDecoder('utf8');

let euro = new Buffer([0xE2, 0x82]);
let euro2 = new Buffer([0xAC]);

console.log(decoder.write(euro));
console.log(decoder.write(euro2));

console.log(euro.toString());
console.log(euro2.toString());
```

The result to the console is a blank line and a second line with the euro symbol (€) when you use `StringDecoder`, but two lines of gibberish when you use `buffer.toString()`.

You can also convert a string to an existing buffer using `buffer.write()`. It's important, though, that the buffer be correctly sized to hold the number of octets necessary for the characters. Again, the euro symbol requires three octets to represent it (0xE2, 0x82, 0xAC):

```
let buf = new Buffer(3);
buf.write('€','utf-8');
```

This is also a good demonstration that the number of UTF-8 characters is not equivalent to the number of octets in the buffer. If in doubt, you can easily check the buffer size with `buffer.length`:

```
console.log(buf.length); // 3
```

Buffer Manipulation

You can read and write buffer contents at a given offset with a variety of typed functions. Examples of using these functions are in the following code snippet, which writes out four unsigned 8-bit integers to a buffer, then reads them in again and prints them out:

```
var buf = new Buffer(4);

// write values to buffer
buf.writeUInt8(0x63,0);
buf.writeUInt8(0x61,1);
buf.writeUInt8(0x74,2);
buf.writeUInt8(0x73,3);

// now print out buffer as string
console.log(buf.toString());
```

Try this yourself by copying the code into a file and running it. You can also read each individual 8-bit integer, using `buffer.readUInt8()`.

Node supports reading in and writing out signed and unsigned 8-, 16-, and 32-bit integers, as well as floats and doubles. For all types other than the 8-bit integers, you can also pick whether you want *little-endian* or *big-endian* format. Examples of some of the functions supported are the following:

- `buffer.readUIntLE()`: read value at buffer offset using little-endian format
- `buffer.writeUInt16BE()`: write unsigned 16-bit integer at offset using big-endian format
- `buffer.readFloatLE()`: read float at offset with little-endian format
- `buffer.writeDoubleBE()`: write 64-bit double value at offset using big-endian format

Endianness

If you're unfamiliar with endianness, or byte order, the format determines how the value is stored: whether the most significant byte is stored at the lowest memory address (big-endian) or the least significant byte is stored at the lowest memory address (little-endian).

Figure 2-1, from the Wikipedia entry for endianness, provides a good visual demonstration of the differences between the two.

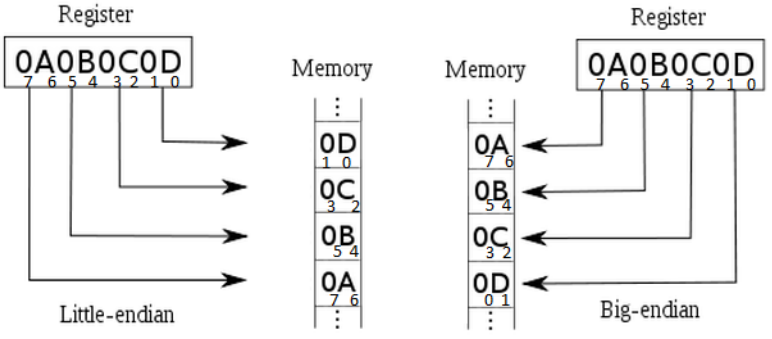


Figure 2-1. Demonstration of little-endian and big-endian formats. Image courtesy of Wikipedia.

You can also write 8-bit integers directly using an array-like format:

```
var buf = new Buffer(4);

buf[0] = 0x63;
buf[1] = 0x61;
buf[2] = 0x74;
buf[3] = 0x73;
```

In addition to reading and writing to a specific buffer offset, you can also create a new buffer consisting of a section of the old, using `buffer.slice()`. What's particularly interesting about this capability is that modifying the contents of the new buffer also modifies the contents of the old buffer. Example 2-3 demonstrates this by creating a buffer from a string, grabbing a slice of the existing buffer to create a new one, and then modifying the contents in the new buffer. Both buffers are then printed out to the console so you can see this *in-place* modification.

Example 2-3. Demonstrating in-place modification of old buffer when changing new buffer

```
var buf1 = new Buffer('this is the way we build our buffer');
var lnth = buf1.length;

// create new buffer as slice of old
var buf2 = buf1.slice(19,lnth);
console.log(buf2.toString()); // build our buffer

//modify second buffer
buf2.fill('*',0,5);
console.log(buf2.toString()); // ***** our buffer
```

```
// show impact on first buffer
console.log(buf1.toString()); // this is the way we ***** our buffer
```

If you ever need to test whether buffers are equivalent, you can use the `buffer.equals()` function:

```
if (buf1.equals(buf2)) console.log('buffers are equal');
```

You can also copy the bytes from one buffer to another using `buffer.copy()`. You can copy all or part of the bytes using optional parameters. Note, though, that if the second buffer isn't large enough to hold all of the contents, you'll only get the portion of the bytes that fit:

```
var buf1 = new Buffer('this is a new buffer with a string');

// copy buffer
var buf2 = new Buffer(10);
buf1.copy(buf2);

console.log(buf2.toString()); // this is a
```

If you need to compare buffers, you can use `buffer.compare()`, which returns a value indicating whether the compared buffer lexically comes before or after. If the compared buffer comes before, a value of -1 is returned; after, a value of 1. If the two buffers have equivalent bytes, a value of 0 is returned:

```
var buf1 = new Buffer('1 is number one');
var buf2 = new Buffer('2 is number two');

var buf3 = new Buffer(buf1.length);
buf1.copy(buf3);

console.log(buf1.compare(buf2)); // -1
console.log(buf2.compare(buf1)); // 1
console.log(buf1.compare(buf3)); // 0
```

There is another buffer class, `SlowBuffer`, that can be used if ever you need to retain the buffer contents for a small buffer for a long period of time. Normally, Node creates buffers from a preallocated chunk of memory if the buffers are small (less than 4 KB in size). This way, garbage collection doesn't have to deal with tracking and cleaning up many small chunks of memory.

The `SlowBuffer` class allows you to create small buffers from outside this preallocated (pooled) memory chunk and have them persist for longer periods of time. As you can imagine, though, using the class can have significant impact on performance. It should be used only if *nothing* else will work.

Node's Callback and Asynchronous Event Handling

JavaScript is single-threaded, which makes it inherently synchronous. This means that JavaScript is executed, line by line, until the application is finished. Since Node is based in JavaScript, it inherits this single-threaded synchronous behavior.

However, if you have functionality that needs to wait on something, such as opening a file, a web response, or other activity of this nature, then blocking the application until the operation is finished would be a major point of failure in a server-based application.

The solution to prevent blocking is the event loop.

The Event Queue (Loop)

To enable asynchronous functionality, applications can take one of two approaches. One approach would be to assign a thread to each time-consuming process. The rest of the code could then go on its way, in parallel. The problem with this approach is that threads are expensive. They're expensive in resources, and they're expensive in added application complexity.

The second approach is to adopt an event-driven architecture. What happens is that when a time-consuming process is invoked, the application doesn't wait for it to finish. Instead, the process signals when it's finished by emitting an event. This event gets added into a queue, or *event loop*. Any dependent functionality registers an interest in this event with the application, and when the event is pulled from the event loop and processed, the dependent functionality is invoked, with any event-related data passed to it.

JavaScript in the browser and Node both take the latter approach. In the browser, when you add a `click` handler to an element, what you've done is register (subscribe to) an event and provide a callback function to invoke when the event happens, freeing the rest of the application to continue:

```
<div id="someid"> </div>
<script>
   document.getElementById("someid").addEventListener("click",
                                        function(event) {
       event.target.innerHTML = "I been clicked!";
   }, false);
</script>
```

Node has its own event loop, but rather than wait for a UI event, such as clicking on an element, its loop is used to help facilitate server-based functionality, primarily input/output (I/O). This includes events associated with opening a file and signaling when the file is opened, reading its contents into a buffer and notifying the client the

process is finished, or waiting for a web-based request from a user. Not only are these types of processes potentially time-consuming, but there can also be a lot of contention for resources, and each access of the resource typically locks the resource from other access until the original process is finished. In addition, web-based applications are dependent on user actions and, sometimes, actions of other applications.

Node processes all of the events in the queue, in order. When it gets to the event you're interested in, it invokes the callback function you've provided, passing in any information associated with the event.

In the basic web server created as the first example of Node in Chapter 1, we saw the event loop in action. I'll repeat the code here, so you can have it handy for review:

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

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

In Example 2-4, I modified the code to break out the individual actions, and also to listen to more of the events that occur during the server creation, client connection, and listening process.

Example 2-4. Basic web server with additional event highlighting

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

var server = http.createServer();

server.on('request', function (request, response) {

   console.log('request event');

   response.writeHead(200, {'Content-Type': 'text/plain'});
   response.end('Hello World\n');
});

server.on('connection', function() {
   console.log('connection event');
});

server.listen(8124, function() {

   console.log('listening event');
});

console.log('Server running on port 8124');
```


www.allitebooks.com

Note that the `requestListener()` function, the server request callback, is no longer called in the `http.createServer()` function. Instead, the application assigned the newly created HTTP server to a variable, which it then uses to capture two events:

- The `request` event, emitted each time a web request is made from a client
- The `connection` event, emitted each time a new client connects to the web application

In both cases, the events are subscribed to using the `on()` function, which the HTTP server class inherits from the `EventEmitter` class. I'll get into the object this functionality is inherited from in the next section, but for now, let's focus on the events themselves. There's one more event subscribed to in the example, and that's the listening event, which is accessed using a callback function on the HTTP `server.listen()` function.

One object, the HTTP server, and three events: request, connection, and listening. So, what happens when the application is created and web requests are made?

Starting the application immediately prints out the message that the "Server is running on port 8124." That's because the application doesn't block when the server is created, a client connects, or when we start listening for requests. So, the first time a `console.log()` message is actually completed is when we're past all of the nonblocking asynchronous functions.

The very next message is "listening event." As soon as we create the server, we want to listen for new connections and requests. We do so by calling the `server.listen()` function. We don't need to wait for any "server-created" event, as the `http.createServer()` function returns immediately. You can test this for yourself by inserting a `console.log()` message directly after the `http.createServer()` function call. If you do add this line, it will be the first printed out to the console when you start the application.

In the previous version of the application, the `server.listen()` function is chained to the `http.createServer()` function, but it doesn't need to be. Doing so was a matter of convenience and programming elegance, not event-driven necessity. The `server.listen()` function is, however, an asynchronous function with a callback, invoked when the listening event is emitted. Therefore, the console message is displayed *after* the message about the server running at port 8124.

No other message is printed out until a client connects with the web application. Then we'll receive the connection event message, because the connection is the first event invoked with a new client. Following it are one or two request event messages. The reason for the difference is how each browser makes a request to a new website. Chrome wants the resource but also wants the *favicon.ico*, so the application gets two

requests. Firefox and IE don't, so the application only gets one request message for these browsers.

If you refresh the page request in the same browser, you'll only get the request event message(s). The connection has already been established and is maintained until the user closes the browser or some timeout occurs. Accessing the same resource with different browsers establishes a separate connection event for each.

Accessing the web application using Chrome results in the following messages to the console:

- Server running on port 8124
- Listening event
- Connection event
- Request event
- Request event

If you have a function in a module or directly in an application that you want to make asynchronous, then you need to define it using a specific criteria, covered next.

Creating an Asynchronous Callback Function

To demonstrate the fundamental structure of the callback functionality, Example 2-5 is a complete Node application that creates an object with one function, `doSomething()`. The function takes three arguments: the first is returned as data if no error occurs, the second must be a string, and the third is the callback function. In `doSomething()`, if the second argument is missing or is not a string, the object creates a new `Error` object, which gets returned in the callback function. If no error occurs, the callback function is called, the error is set to `null`, and the function data gets returned (in this case, the first argument).

The key elements of the callback functionality are in boldface in Example 2-5.

Example 2-5. The fundamental structure of the last callback functionality

```
var fib = function (n) {
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);
};

var Obj = function() { };

Obj.prototype.doSomething = function(arg1_) {
   var callback_ = arguments[arguments.length - 1];
   callback = (typeof(callback_) == 'function' ? callback_ : null);
   var arg1 = typeof arg1_ === 'number' ? arg1_ : null;
```

```
   if (!arg1)
      return callback(new Error('first arg missing or not a number'));

      process.nextTick(function() {

         // block on CPU
         var data = fib(arg1);
         callback(null, data);
    });
}

var test = new Obj();
var number = 10;

test.doSomething(number, function(err,value) {
      if (err)
         console.error(err);
      else
         console.log('fibonaci value for %d is %d', number, value);
});

console.log('called doSomething');
```

The first key functionality is to ensure the last argument is a callback function, and that the first argument in the callback function is an error. As noted in Chapter 1, this error-first pattern is frequently referred to as an *errback*. We can't determine the user's intent, but we can make sure the last argument is a function, and that will have to do.

The second key functionality is to create the new Node `Error` object if an error occurs, and return it as the result to the callback function. We can't rely on `throw...catch` in an asynchronous world, so the error handling must be handled in the `Error` object in the callback.

The last critical functionality is to invoke the callback function, passing in the function's data if no error occurs. However, to ensure this callback is asynchronous, we call it within a `process.nextTick()` function. The reason is that `process.nextTick()` ensures the event loop is cleared before the function is called. This means all of the synchronous functionality is processed before the blocking functionality (if any) is invoked. In the example, the blocking doesn't occur because of I/O but because the operation is CPU-bound. Calling a Fibonacci sequence function with a value of 10 may not take time, but calling the same function with a value of 50 or more will, depending on your system's resources. The Fibonacci function is called within `process.nextTick()`, therefore ensuring that the CPU-bound functionality is handled ascynchronously.

In short, everything else is changeable, as long as these four key functionalities are present:

- Ensure the last argument is a callback function.
- Create a Node `Error` and return it as the first argument in the callback function if an error occurs.
- If no error occurs, invoke the callback function, set the error argument to `null`, and pass in any relevant data.
- The callback function must be called within `process.nextTick()` to ensure the process doesn't block.

If you change the value of `number` to 10, the application prints the following to the console:

```
called doSomething
[Error: first argument missing or not a number]
```

If you look through the code in the *lib* directory of the Node installation, you'll see the last callback pattern repeated throughout. Though the functionality may change, this pattern remains the same.

This approach is quite simple and ensures consistent results from asynchronous methods.

Callback Nesting

The use of a callback function is simple, but generates its own challenges, including deep-nesting of callbacks. I cover deep-nesting and solutions in Chapter 3, in the section "Better Callback Management with Async" on page 77.

Earlier I mentioned that the `http.Server` object inherits from another object, and that's where we get the event-emitting capability. This object is named, appropriately enough, `EventEmitter`, and we'll cover it next.

Node Is Single-Threaded…Mostly

Node's event loop is single-threaded. However, this doesn't mean there aren't multiple threads busily working away in the background.

Node invokes functionality, such as the File System (fs), that is implemented in C++ rather than JavaScript. The fs functionality makes use of worker threads in order to accomplish its functionality. In addition, Node makes use of the libuv library, which makes use of a pool of worker threads for some functionality. How many threads is operating-system-dependent.

If you stick with JavaScript and creating JavaScript modules, you need never be concerned about worker threads and libuv. Point of fact: we have been patted on the head

and told not to worry our pretty heads about worker threads. And that's OK with me, having worked with multithreaded environments in the past.

However, if you're interested in developing Addon extensions for Node, you'll need to become very familiar with libuv. A good place to start is *An Introduction to libuv*.

For more on the interesting, hidden world of multithreaded Node, I suggest the answers to the Stack Overflow question: When is thread pool used?

EventEmitter

Scratch underneath the surface of many of the Node core objects, and you'll find `EventEmitter`. Anytime you see an object `emit` an event, and an event handled with the function `on`, you're seeing `EventEmitter` in action. Understanding how `EventEmitter` works and how to use it are two of the more important components of Node development.

The `EventEmitter` enables asynchronous event handling in Node. To demonstrate its core functionality, we'll try a quick test application.

First, include the Events module:

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

Next, create an instance of `EventEmitter`:

```
var em = new events.EventEmitter();
```

Use the newly created `EventEmitter` to do two essential tasks: attach an event handler to an event, and emit the actual event. The `EventEmitter.on()` event handler is invoked when a specific event is emitted. The first parameter to the method is the name of the event; the second, the callback function to perform some functionality:

```
em.on('someevent', function(data) { ... });
```

The event is emitted on the object via the `EventEmitter.emit()` method when some condition is met:

```
if (somecriteria) {
   en.emit('data');
}
```

In Example 2-6, we create an `EventEmitter` instance that emits a timed event every three seconds. In the event handler function for this event, a message with a counter is output to the console. Note the correlation between the `counter` argument in the `EventEmitter.emit()` function and the corresponding data in the `EventEmitter.on()` function that processes the event.

Example 2-6. Very basic test of the EventEmitter functionality

```
var eventEmitter = require('events').EventEmitter;
var counter = 0;

var em = new eventEmitter();

setInterval(function() { em.emit('timed', counter++); }, 3000);

em.on('timed', function(data) {
  console.log('timed ' + data);
});
```

Running the application outputs timed event messages to the console until the application is terminated. The key takeaway from this simple application is that an event is triggered via the `EventEmitter.emit()` function, and the `EventEmitter.on()` function can be used to trap that event and process it.

This is an interesting example, but not particularly helpful. What we need is the ability to add `EventEmitter` functionality to our existing objects—not use instances of `EventEmitter` throughout our applications. This is what `http.Server`, and most other event-enabled classes in Node, do.

`EventEmitter` functionality is inherited, so we have to use another Node object, `Util`, to enable this inheritance. The `Util` module is imported into an application using:

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

The `Util` module is a helpful beast. I'll cover most of its functionality in Chapter 11, when I get into debugging Node applications. But one of the functions, `util.inherits()`, is essential right now.

The `util.inherits()` function enables one constructor to inherit the prototype methods of another, a *superconstructor*. To make `util.inherits()` even more special, you can also access the superconstructor directly in the functions of the constructor.

The `util.inherits()` function allows us to inherit Node event queue functionality with any class, and that includes `EventEmitter`:

```
util.inherits(Someobj, EventEmitter);
```

By using `util.inherits()` with the object, you can call the `emit` method within the object's methods, and code event handlers on the object instances:

```
Someobj.prototype.someMethod = function() { this.emit('event'); };
...
Someobjinstance.on('event', function() { });
```

Rather than attempt to decipher how `EventEmitter` works in the abstract sense, let's move on to Example 2-7, which shows a working example of a class inheriting `EventE`

`mitter`'s functionality. In the application, a new class, `inputChecker`, is created. The constructor takes two values, a person's name and a filename. It assigns the person's name to a property, and also creates a reference to a writable stream using the File System module's `createWriteStream` method.

The object also has a method, `check`, that checks incoming data for specific commands. One command (`wr:`) emits a write event, another (`en:`) an end event. If no command is present, then an `echo` event is emitted. The object instance provides event handlers for all three events. It writes to the output file for the write event, it echoes the input for the commandless input, and it terminates the application with an end event, using the `process.exit` method.

All input comes from standard input (`process.stdin`). Written output uses a writable stream, which is a way of creating a new output source in the background and to which future writes are queued. It's a more efficient file output method if you're expecting frequent activity, as we are in this application. The input that's echoed is just output to `process.stdout`.

Example 2-7. Creating an event-based object that inherits from EventEmitter

```
"use strict";

var util = require('util');
var eventEmitter = require('events').EventEmitter;
var fs = require('fs');

function InputChecker (name, file) {
   this.name = name;
   this.writeStream = fs.createWriteStream('./' + file + '.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0o666});
};

util.inherits(InputChecker,eventEmitter);

InputChecker.prototype.check = function check(input) {

  // trim extraneous white space
  let command = input.trim().substr(0,3);

  // process command
  // if wr, write input to file
  if (command == 'wr:') {
     this.emit('write',input.substr(3,input.length));

  // if en, end process
  } else if (command == 'en:') {
     this.emit('end');
```

```
  // just echo back to standard output
  } else {
     this.emit('echo',input);
  }
};

// testing new object and event handling
let ic = new InputChecker('Shelley','output');

ic.on('write', function(data) {
   this.writeStream.write(data, 'utf8');
});

ic.on('echo', function( data) {
   process.stdout.write(ic.name + ' wrote ' + data);
});

ic.on('end', function() {
   process.exit();
});

// capture input after setting encoding
process.stdin.setEncoding('utf8');
process.stdin.on('readable', function() {
   let input = process.stdin.read();
   if (input !== null)
      ic.check(input);
});
```

Note that the functionality also includes the `process.stdin.on` event handler method, since `process.stdin` is one of the many Node objects that inherit from `EventEmitter`.

No Octal Literals in Strict Mode

In Example 2-7, I used *strict* mode because I'm using the `let` ES6 statement. Because I'm using strict mode, though, I can't use octal literals (such as 0666) in the write stream file descriptor flags. Instead, I use the notation `0o666`, which is an ES6-style literal.

The `on()` function is really a shortcut for the `EventEmitter.addListener` function, which takes the same parameters. So this:

```
ic.addListener('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});
```

is exactly equivalent to:

```
ic.on('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});
```

You can listen to the next event with `EventEmitter.once()`:

```
ic.once(event, function);
```

When you exceed 10 listeners for an event, you'll get a warning by default. Use `setMaxListeners`, passing in a number, to change the number of listeners. Use a value of zero (`0`) for an unlimited amount of listeners.

You can also remove listeners with `EventEmitter.removeListener()`:

```
ic.on('echo', callback);

ic.removeListener('echo',callback);
```

This removes one listener from the array of event listeners, maintaining the order. However, if you've copied the array of event listeners, for whatever reason, using `EventEmitter.listeners()`, you'll need to re-create it once you've removed listeners.

The Node Event Loop and Timers

In the browser we have `setTimeout()` and `setInterval()` to use for timers, and we have these same functions in Node, too. They're not exactly the same, since the browser uses the event loop maintained by whatever is the browser engine, and Node's event loop is handled by a C++ library, libuv—but any differences are likely to be negligible.

The Node `setTimeout()` takes a callback function as first parameter, the delay time (in milliseconds) as second parameter, and an optional list of arguments:

```
setTimeout(function(name) {
             console.log('Hello ' + name);
           }, 3000, 'Shelley');

console.log("waiting on timer...");
```

The name in the argument list is passed as an argument to the callback function in `setTimeout()`. The timer value is set to 3,000 milliseconds. The `console.log()` message of "waiting on timer..." is printed out almost immediately, as the `setTimeout()` function is asynchronous.

You can cancel a timer if your application assigns it to a variable when creating it. I modified the previous Node application to incorporate a faster-acting cancellation and message:

```
var timer1 = setTimeout(function(name) {
              console.log('Hello ' + name);
            }, 30000, 'Shelley');

console.log("waiting on timer...");

setTimeout(function(timer) {
             clearTimeout(timer);
             console.log('cleared timer');
            }, 3000, timer1);
```

The timer is set for a very long period of time, plenty of time for the new timer to invoke a callback that actually cancels the timer.

The `setInterval()` function operates in a similar manner to `setTimeout()`, except that the timer continues to re-fire until the application is cancelled, or the timer is cleared with `clearInterval()`. Modifying the `setTimeout()` example for one demonstrating `setInterval()`, the message repeats nine times before it gets cancelled.

```
var interval = setInterval(function(name) {
               console.log('Hello ' + name);
             }, 3000, 'Shelley');

setTimeout(function(interval) {
             clearInterval(interval);
             console.log('cleared timer');
            }, 30000, interval);

console.log('waiting on first interval...');
```

As the Node documentation carefully notes, there's no guarantee that the callback function will be invoked in exactly *n* milliseconds (whatever *n* is). This is no different than the use of `setTimeout()` in a browser—we don't have absolute control over the environment, and factors could slightly delay the timer. For the most part, we can't sense any time discrepancy when it comes to the timer functions. However, if we're creating animations, we can actually see the impact.

There are two Node-specific functions you can use with the timer/interval which are returned when you call `setTimeout()` or `setInterval()`: `ref()` and `unref()`. If you call `unref()` on a timer, and it's the only event in the event queue, the timer is cancelled and the program is allowed to terminate. If you call `ref()` on the same timer object, this keeps the program going until the timer has processed.

Returning to the first example, we'll create a longish timer, and then call `unref()` on it to see what happens:

```
var timer = setTimeout(function(name) {
              console.log('Hello ' + name);
            }, 30000, 'Shelley');

timer.unref();

console.log("waiting on timer...");
```

Running the application prints out the console message, and then quits. The reason is that the timer set with `setTimeout()` is the only event in the application's event queue. But what if we added another event? Modifying the code, we'll add an interval, as well as the timeout, and call `unref()` on the timeout:

```
var interval = setInterval(function(name) {
              console.log('Hello ' + name);
            }, 3000, 'Shelley');

var timer = setTimeout(function(interval) {
             clearInterval(interval);
             console.log('cleared timer');
            }, 30000, interval);

timer.unref();

console.log('waiting on first interval...');
```

The timer is allowed to continue, which means it terminates the interval. And yet it's the interval events that have kept the timer alive long enough to allow the timer to clear the interval.

The last set of Node timer-like functions are unique to Node: `setImmediate()` and `clearImmediate()`. The `setImmediate()` creates an event, but the event has precedence over those created by `setTimeout()` and `setInterval()`. However, it doesn't have precedence over I/O events. And it has no timer associated with it. The `setImmediate()` event is emitted after all I/O events, before any timer events, and in the current event queue. If you call it from within a callback function, then it's placed into the next event loop after the one in which it was invoked is finished. It is a way of adding an event to the current or next event loop without adding any arbitrary timers. It's more efficient than `setTimeout(callback, 0)`, since it takes precedence over other timer events.

It's similar to another function, `process.nextTick()`, except that the `process.nextTick()` callback function is invoked once the current event loop is finished but before any new I/O events are added. As demonstrated earlier in the section "Creating an Asynchronous Callback Function" on page 36, it's used extensively to implement Node's asynchronous functionality.

Nested Callbacks and Exception Handling

It's not unusual to find the following in a client-side JavaScript application:

```
val1 = callFunctionA();
val2 = callFunctionB(val1);
val3 = callFunctionC(val2);
```

The functions are called in turn, passing the results from the earlier function to each subsequent function. Since all the functions are synchronous, we don't have to worry about the function calls getting out of sequence—there are no unexpected results.

Example 2-8 shows a relatively common case of this type of sequential programming. The application uses synchronous versions of Node's File System methods to open a file and get its data, modify the data by replacing all references to "apple" with "orange," and output the resulting string to a new file.

Example 2-8. A sequential synchronous application

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

try {
   var data = fs.readFileSync('./apples.txt','utf8');
   console.log(data);
   var adjData = data.replace(/[A|a]pple/g,'orange');

   fs.writeFileSync('./oranges.txt', adjData);
} catch(err) {
  console.error(err);
}
```

Since problems can occur and we can't be sure errors are handled internally in any module function, we wrap all of the function calls in a `try` block to allow for graceful —or at least, more informative—exception handling. The following is an example of what the error looks like when the application can't find the file to read:

```
{ [Error: ENOENT: no such file or directory, open './apples.txt']
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: './apples.txt' }
```

While perhaps not very user-friendly, at least it's a lot better than the alternative:

```
$ node nested2
fs.js:549
  return binding.open(pathModule._makeLong(path), stringToFlags(flags),
mode);
                 ^
```

```
Error: ENOENT: no such file or directory, open './apples.txt'
    at Error (native)
    at Object.fs.openSync (fs.js:549:18)
    at Object.fs.readFileSync (fs.js:397:15)
    at Object.<anonymous>
              (/home/examples/public_html/learnnode2/nested2.js:3:18)
    at Module._compile (module.js:435:26)
    at Object.Module._extensions..js (module.js:442:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:311:12)
    at Function.Module.runMain (module.js:467:10)
    at startup (node.js:136:18)
```

Converting this synchronous sequential application pattern to an asynchronous implementation requires a couple of modifications. First, we have to replace all functions with their asynchronous counterparts. However, we also have to account for the fact that each function doesn't block when called, which means we can't guarantee the proper sequence if the functions are called independently of each other. The only way to ensure that each function is called in its proper sequence is to use *nested callbacks*.

Example 2-9 is an asynchronous version of the application from Example 2-8. All of the File System function calls have been replaced by their asynchronous versions, and the functions are called in the proper sequence via a nested callback. In addition, the use of the `try...catch` block is removed.

We can't use `try...catch`, because the use of the asynchronous functions means that the `try...catch` block is actually processed before the asynchronous function has been called. So trying to throw an error in the callback function is attempting to throw an error outside of the process to catch it. Instead, we just process the error directly: if an error exists, handle it and return; if no error exists, continue the callback function's process.

Example 2-9. Application from Example 2-8 converted into asynchronous nested callbacks

```
var fs = require('fs');
fs.readFile('./apples.txt','utf8', function(err,data) {
   if (err) {
      console.error(err);
   } else {

     var adjData = data.replace(/apple/g,'orange');

     fs.writeFile('./oranges.txt', adjData, function(err) {

        if (err) console.error(err);
     });
   }
```

```
});
```

In Example 2-9, the input file is opened and read, and only when both actions are finished does the callback function get called. In this function, the error is checked to see if it has a value. If it does, the error object is printed out to the console. If no error occurs, the data is processed and the asynchronous `writeFile()` method is called. Its callback function has only one argument, the error object. If it's not `null`, it's also printed out to the console.

If an error occurred, it would look similar to the following:

```
{ [Error: ENOENT: no such file or directory, open './apples.txt']
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: './apples.txt' }
```

If you want the stack trace of the error, you can print out the `stack` property of the Node error object:

```
if (err) {
   console.error(err.stack);
}
```

The result would look like the following:

```
Error: ENOENT: no such file or directory, open './apples.txt'
    at Error (native)
```

Including a sequential asynchronous function call adds another level of callback nesting and, potentially, new challenges in error handling. In Example 2-10, we access a listing of files for a directory. In each of the files, we replace a generic domain name with a specific domain name using the string `replace` method, and the result is written *back* to the original file. A log of each changed file is maintained using an open write stream.

Example 2-10. Retrieving directory listing for files to modify

```
var fs = require('fs');
var writeStream = fs.createWriteStream('./log.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});

writeStream.on('open', function() {
   // get list of files
   fs.readdir('./data/', function(err, files) {

      // for each file
```

```
      if (err) {
         console.log(err.message);
      } else {
         files.forEach(function(name) {

            // modify contents
            fs.readFile('./data/' + name,'utf8', function(err,data) {

               if (err){
                  console.error(err.message);
               } else {
                  var adjData = data.replace(/somecompany\.com/g,
                              'burningbird.net');

                  // write to file
                  fs.writeFile('./data/' + name, adjData, function(err)
                    {

                     if (err) {
                        console.error(err.message);
                     } else {

                        // log write
                        writeStream.write('changed ' + name + '\n',
                        'utf8', function(err) {

                           if(err) console.error(err.message);
                        });
                     }
                  });
               }
            });
         });
      }
   });
});

writeStream.on('error', function(err) {
  console.error("ERROR:" + err);
});
```

First off, we see something new: the use of event handling to handle errors when we make the `fs.createWriteStream` function call. The reason we're using the event handling is that `createWriteStream` is asynchronous, so we can't use the traditional `try...catch` error handling. At the same time, it also doesn't provide a callback function where we can capture errors. Instead, we look for an `error` event and handle it by writing out the error message. And then we look for an `open` event (a successful operation) and do the file processing.

The application prints out the error message directly.

Though the application looks like it's processing each file individually before moving on to the next, remember that each of the methods used in this application is asynchronous. If you run the application several times and check the *log.txt* file, you'll see that the files are processed in a different, seemingly random order. In my *data* subdirectory, I had five files. Running the application three times in a row resulted in the following output to *log.txt* (blank lines inserted for clarity):

```
changed data1.txt
changed data2.txt
changed data3.txt
changed data4.txt
changed data5.txt

changed data2.txt
changed data4.txt
changed data3.txt
changed data1.txt
changed data5.txt

changed data1.txt
changed data2.txt
changed data5.txt
changed data3.txt
changed data4.txt
```

Another issue arises if you want to check when all of the files have been modified in order to do something. The `forEach` method invokes the iterator callback functions asynchronously, so it doesn't block. Adding a statement following the use of `forEach`, like the following:

```
console.log('all done');
```

doesn't really mean the application is finished, just that `forEach` didn't block. If you add a `console.log` statement at the same time, you log the changed file:

```
// log write
writeStream.write('changed ' + name + '\n',
'utf8', function(err) {

    if(err) {
      console.log(err.message);
    } else {
      console.log('finished ' + name);
    }
 });
```

and add the following after the `forEach` method call:

```
console.log('all finished');
```

and you'll actually get the following console output:

```
all finished
finished data3.txt
finished data1.txt
finished data5.txt
finished data2.txt
finished data4.txt
```

To solve this challenge, add a counter that is incremented with each log message and then checked against the file array's length to print out the "all done" message:

```
// before accessing directory
var counter = 0;
...
  // log write
  writeStream.write('changed ' + name + '\n',
  'utf8', function(err) {

     if(err) {
        console.log(err.message);
     } else {
        console.log ('finished ' + name);
        counter++;
        if (counter >= files.length) {
           console.log('all done');
        }
     }
  });
```

You'll then get the expected result: an "all done" message displays after all the files have been updated.

The application works quite well—except if the directory we're accessing has subdirectories as well as files. If the application encounters a subdirectory, it spits out the following error, though it keeps on processing the other contents:

```
EISDIR: illegal operation on a directory, read
```

Example 2-11 prevents this type of error by using the `fs.stats` method to return an object representing the data from a Unix `stat` command. This object contains information about the object, including whether it's a file or not. The `fs.stats` method is, of course, another asynchronous method, requiring yet more callback nesting.

Example 2-11. Adding in a stats check of each directory object to make sure it's a file

```
var fs = require('fs');
var writeStream = fs.createWriteStream('./log.txt',
      {flags : 'a',
       encoding : 'utf8',
       mode : 0666});

writeStream.on('open', function() {
```

```
var counter = 0;

// get list of files
fs.readdir('./data/', function(err, files) {

   // for each file
   if (err) {
      console.error(err.message);
   } else {
      files.forEach(function(name) {

         fs.stat('./data/' + name, function (err, stats) {

            if (err) return err;
            if (!stats.isFile()) {
               counter++;
               return;
            }

            // modify contents
            fs.readFile('./data/' + name,'utf8', function(err,data) {

               if (err){
                  console.error(err.message);
               } else {
                  var adjData = data.replace(/somecompany\.com/g,
                          'burningbird.net');

                  // write to file
                  fs.writeFile('./data/' + name, adjData,
                                              function(err) {

                     if (err) {
                        console.error(err.message);
                     } else {

                        // log write
                        writeStream.write('changed ' + name + '\n',
                         function(err) {

                           if(err) {
                              console.error(err.message);
                           } else {
                              console.log('finished ' + name);
                              counter++;
                              if (counter >= files.length) {
                                 console.log('all done');
                              }
                           }
                        });
                     }
                  });
```

```
                }
            });
        });
    });
  }
});
});

writeStream.on('error', function(err) {
  console.error("ERROR:" + err);
});
```

Again, the application performs its purpose, and performs it well—but how difficult it is to read and maintain! I used a `return` for some of the error handling, eliminating one conditional nesting, but the program is still almost impossible to maintain. I've heard this type of nested callback called *callback spaghetti* and the even more colorful *pyramid of doom*, both of which are apt terms.

The nested callbacks continue to push against the right side of the document, making it more difficult to ensure we have the right code in the right callback. However, we can't break the callback nesting apart because it's essential that the methods be called in turn:

1. Start the directory lookup.
2. Filter out subdirectories.
3. Read each file's contents.
4. Modify the contents.
5. Write back to the original file.

What we'd like to do is find a way of implementing this series of method calls without having to depend on nested callbacks. For this, we need to look at third-party modules and other approaches. In Chapter 3, I'll use the Async module to tackle this pyramid of doom. And in Chapter 9, we'll look to see if ES6 promises can help us.

Another approach is to provide a named function as a callback function for each method. This way, you can flatten the pyramid, and it can simplify debugging. However, this approach doesn't solve some of the other problems, such as determining when all processes have finished. For this, you still need an asynchronous control handling module.

CHAPTER 3

Basics of Node Modules and Node Package Manager (npm)

Throughout the book you'll have a chance to work with several Node modules, most of them *core modules*. These modules are included with the Node installation, and incorporated into the application using the global `require` statement.

In this chapter, we'll explore the concept of the Node module more closely, look at the `require` statement in detail, and introduce you to npm, the Node package manager. We'll also go shopping for some non-core modules that many people believe are essential for any Node application development.

An Overview of the Node Module System

Node's basic implementation is kept as streamlined as possible. Rather than incorporate every possible component of use directly into Node, developers offer additional functionality via modules.

Node's module system is patterned after the *CommonJS module system*, a way of creating modules so that they're interoperable. The core of the system is a contract that developers adhere to in order to ensure that their modules play well with others.

Among the CommonJS module system requirements implemented with Node are:

- Support is included for a `require` function that takes the module identifier and returns the exported API.
- The module name is a string of characters, and may include forward slashes (for identification of paths).

- The module must explicitly export that which is to be exposed outside the module.
- Variables are private to the module.

Some Node functionality is global, which means you don't have to do anything to include it. However, most Node functionality is incorporated using the module system.

How Node Finds and Loads a Module

When you want to include access to a Node module, whether a core module or one you've installed outside of the Node application, use the `require` statement:

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

You can also access one specific property of an exported object. For instance, people frequently access only the `parse()` function when using the URL module:

```
var name = require('url').parse(req.url, true).query.name;
```

Or you can access a specific module object to use throughout your application:

```
var spawn = require('child_process').spawn;
```

When your application requests a module, several things happen. First, Node checks to see if the module has been *cached*. Rather than reload the module each time, Node caches the module the first time it's accessed. This eliminates the drag associated with the system having to perform a system lookup for the file.

One-to-One Correspondence: File to Module

Node only supports one module per file.

If the module isn't cached, Node then checks to see if it's a native module. Native modules are those that are precompiled binaries, such as the C++ add-ons discussed in Chapter 1. If the module is a native module, a function is used specifically for native modules, which returns the exported functionality.

If the module isn't cached or isn't a native module, a new Module object is created for it, and the module's `exports` property is returned. We cover the module exports in more detail in "Creating and Publishing Your Own Node Module" on page 69, but this basically returns public-facing functionality to the application.

The module is also cached. If, for some reason, you want to delete the module from cache, you can:

```
delete require('./circle.js');
```

The module is reloaded the next time the application requires it.

As part of loading the module, Node has to resolve the location for it. It goes through a hierarchy of checks when looking for the module file.

First, core modules have priority. You can name your module `http` if you wish, but when you go to load `http`, Node is going to grab the version from core. The only way you can use `http` as the module name is if you also provide a path, to differentiate it from the core module:

```
var http = require ('/home/mylogin/public/modules/http.js');
```

As this example demonstrates, if you provide an absolute or relative path with the filename, Node uses the path. The following looks for the module in the local subdirectory:

```
var someModule = require('./somemodule.js');
```

I give the module extension, but it's not necessary. When you give a module name without the extension, Node first looks for the module in the current subdirectory with a *.js* extension. If found, the module is loaded. If not found, Node looks for a file with a *.json* extension. If a file with the proper name is found with a *.json* extension, the module contents are loaded as JSON. Finally, Node looks for a module with a *.node* extension. It assumes this module is a precompiled Node add-on, and handles it accordingly.

JSON files don't require an explicit *exports* statement. They only need to be proper JSON.

You can also use a more complex relative path:

```
var someModule = require('./somedir/someotherdir/somemodule.js');
```

Or you can use an absolute path if you're sure the application will never be moved. This path is filesystem specific, not a URL:

```
var someModule = require('/home/myname/public/modules/somemodule.js');
```

If the module is installed using npm, you don't need to provide a path; you just list the module name:

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

Node looks for the module in a *node_modules* subdirectory, using a search hierarchy that includes searches in:

1. A *node_modules* subdirectory local to the application (*/home/myname/projects/node_modules*)
2. A *node_modules* subdirectory in the parent subdirectory to the current application (*/home/myname/node_modules*)
3. Continuing up the parent subdirectories, looking for *node_modules*, until top-level (root) is reached (*/node_modules*)
4. Finally, looking for the module among those installed globally (discussed next)

The reason Node uses this hierarchy is so that localized versions of a module are accessed before more global versions. So if you're testing a new version of a module and you've installed it locally relative to your application:

```
npm install somemodule
```

it will be loaded first, rather than the globally installed module:

```
npm install -g somemodule
```

You can see which module is loaded using the `require.resolve()` function:

```
console.log(require.resolve('async'));
```

The resolved location of the module is returned.

If you provide a folder name as a module, Node looks for a *package.json* file that contains a `main` property flagging the module file to load:

```
{ "name" : "somemodule",
  "main" : "./lib/somemodule.js" }
```

If Node can't find a *package.json* file, it looks for an *index.js* or *index.node* file to load.

If all these searches fail, you'll get an error.

Caching Is Filename-Specific

Be aware that caching is based on the filename and path used to load the module. If you've cached a global version of a module and then load a local version, the local version also gets cached.

Since the Module object is JavaScript-based, we can peek into the Node source code and get a closer look at what happens behind the scenes with all of this.

Each module wrapped with the Module object has a `require` function, and the global `require` we use invokes the Module-specific function. The `Module.require()` function, in turn, calls another internal function, `Module._load()`, which performs all of the functionality I just covered. The only exception is if the request is for REPL, covered in the next chapter, which has its own unique handling.

If the module is the main module, the one that's actually invoked at the command line (that I refer to as application), it's actually assigned to a property, `require.main`, of the global `require` object. Type the following into a file named *test.js* and run it with Node:

```
console.log(require);
```

You'll see the `main` object, which is the Module object wrapping the application code, and you can see that the filename is the filename and path for the application. You can also see the paths Node uses to look for modules, as well as the application cache, containing only the application in this instance.

Locating the Node Source

You can look at all of Node's functionality if you download the source. You don't have to use the source to build Node on your system; just use it as a learning exercise. The JavaScript functionality is located in the */lib* subdirectory, and the actual C++ code for Node can be found in */src*.

This leads me to revisit the concept of Node's global object once more. In Chapter 2, I covered the Node `global` object, and described how it differs from the browser global object. I noted that unlike the browser, top-level variables are constrained to their immediate context, which means that variables declared in a module are not going to conflict with variables declared in the application or any other module included in the application. That's because Node wraps all scripts in the following:

```
function (module, exports, __filename, ...) {}
```

In other words, Node wraps the modules (main or otherwise) in anonymous functions, only exposing what the module developer wants to expose. And since these module properties are prefaced by the module name when you use the module, they can't conflict with your locally declared variables.

Speaking of context, I'll get more into that in the next section.

Sandboxing and the VM Module

One of the first things you learn as a JavaScript developer is to avoid `eval()` at all costs. The reason is that `eval()` executes your JavaScript in the same context as the rest of your application. You're taking an unknown or arbitrary block of JavaScript and executing it with the same level of trust as the code you've carefully written. It's the same as taking text from an input field and tacking it into a SQL request without ensuring something nasty hasn't been inserted.

If, for some reason, you do need to execute an arbitrary chunk of JavaScript in your Node application, you can do so using the VM module to sandbox the script. As the

Node developers note, however, this isn't a completely trustworthy approach. The only safe way to execute an arbitrary chunk of JavaScript is in a separate process. However, if you're comfortable with the source of the JavaScript but are interested in avoiding unintended and accidental consequences, then you can isolate that script from your local environment with VM.

Scripts can be precompiled first, using the `vm.Script` object, or passed in as part of a function called directly on `vm`. There are also three types of functions. The first, either `script.runInNewContext()` or `vm.runInNewContext()`, runs the script in the new context, and the script doesn't have access to either local variables or the global object. Instead, a new *contextified* sandbox is passed in the function. The following code demonstrates this concept. The sandbox contains two global values—the same names as two Node global objects—but redefined:

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

var sandbox = {
   process: 'this baby',
   require: 'that'
};

vm.runInNewContext('console.log(process);console.log(require)',sandbox);
```

An error results, because the `console` object isn't part of the runtime context for the script. You can make it so:

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

var sandbox = {
   process: 'this baby',
   require: 'that',
   console: console
};

vm.runInNewContext('console.log(process);console.log(require)',sandbox);
```

But that defeats the purpose of creating an entirely new context for the script. If you want the script to have access to the global console (or other) object, use `runInThisContext()` instead. In Example 3-1, I'm using the Script object to demonstrate how the context includes the global object, but not the local objects.

Example 3-1. Running a script for access to the global console

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

global.count1 = 100;
var count2 = 100;

var txt = 'if (count1 === undefined) var count1 = 0; count1++;' +
```

```
          'if (count2 === undefined) var count2 = 0; count2++;' +
          'console.log(count1); console.log(count2);';

var script = new vm.Script(txt);
script.runInThisContext({filename: 'count.vm'});

console.log(count1);
console.log(count2);
```

The result of running this application is:

```
101
1
101
100
```

The variable `count1` is declared on the global object, and is available in the context in which the script is run. The variable `count2` is a local variable, and must be defined within the context. Changes to the local variable in the sandboxed script do not impact the local variable of the same name in the containing application.

I'll get an error if I don't declare `count2` in the script that's running in a separate context. The error is displayed because one of the sandbox content function's options is `displayErrors`, set to `true` by default. The other options for `runInThisContext()` are `filename`, shown in the example, and `timeout`, which is the number of milliseconds the script is allowed to run before being terminated (and throwing an error). The `filename` option is used to specify a filename that shows up in stack traces when the script is run. If you want to specify a filename for the Script object, though, you need to pass it in when you create the Script object, not in the context function call:

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

global.count1 = 100;
var count2 = 100;

var txt = 'count1++;' +
          'count2++;' +
          'console.log(count1); console.log(count2);';

var script = new vm.Script(txt, {filename: 'count.vm'});

try {
  script.runInThisContext();
} catch(err) {
   console.log(err.stack);
}
```

Other than the `filename` difference, Script supports the other two global options in the context function calls: `displayErrors` and `timeout`.

Running the code results in a displayed error because the sandboxed script does not have access to the local variable (`count2`) in the application, though it does have access to the global `count1`. And when the error is printed out, the *stack trace* is printed, displaying the filename passed as an option.

Instead of writing the code out directly in the application, we can load it from a file. Given the following script to run:

```
if (count1 === undefined) var count1 = 0; count1++;
if (count2 === undefined) var count2 = 0; count2++;
console.log(count1); console.log(count2);
```

We can precompile it and run it in a sandbox with the following:

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

global.count1 = 100;
var count2 = 100;

var script = new vm.Script(fs.readFileSync('script.js','utf8'));
script.runInThisContext({filename: 'count.vm'});

console.log(count1);
console.log(count2);
```

What's to stop us from using the File System module directly in the script? It's assigned to a local variable, and not accessible. Why can't we just import it into the script? Because `require` is not available. None of the global objects or functions, such as `require`, are accessible in the script.

The last sandbox function is `runInContext()`. It also takes a sandbox, but the sandbox must be *contextualized* (context explicitly created) before the function call. We'll call it on the VM object, directly, in the following code. Note that we're adding a new variable to the contextualized sandbox in the code, which then shows in the application:

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

var sandbox = {
     count1 : 1
    };

vm.createContext(sandbox);
if (vm.isContext(sandbox)) console.log('contextualized');

vm.runInContext('count1++; counter=true;',sandbox,
                {filename: 'context.vm'});

console.log(util.inspect(sandbox));
```

The result from this application is:

```
contextualized
{ count1: 2, counter: true }
```

The `runInContext()` function supports the three options that `runInThisContext()` and `runInNewContext()` supports. Again, the difference between running the functions in Script and directly in VM is that the Script object precompiles the code, and you pass the filename in when you create the object rather than as one of the options to the function calls.

Running Script in a Separate Process

If you're interested in further isolating the script by running it in a separate process, check out the third-party sandboxing modules that provide this additional protection. The next section describes how to search for these.

An In-Depth Exploration of NPM

Much of the rich functionality associated with Node comes in via third-party modules. There are router modules, modules for working with relational or document database systems, template modules, testing modules, and even modules for payment gateways.

GitHub

Though not a requirement, developers are encouraged to upload their modules to GitHub.

To use a module, you can download the source and then install it manually into your application environment. Most modules provide basic installation instructions, or, at a minimum, you can deduce the installation requirements by examining the files and directories included in the module. However, there is a far easier way to install a Node module: using npm.

The npm site is at *http://npmjs.org/*. Documentation is available at the npm Documentation site.

Node comes with npm installed, but it doesn't always come with the most recent version of npm. If you want to use a different version of npm, use the following to update the version you have (using `sudo`, if appropriate to your environment):

```
npm install npm -g
```

Be careful with updating npm, though: if you're using a different version than your teammates, you can have very unexpected results.

To get a detailed overview of npm commands, use the following:

```
$ npm help npm
```

Modules can be installed globally or locally. The local installation is the best approach if you're working on an isolated project and everyone sharing the same system doesn't need access to the module. A local installation, which is the default, installs the module in the current location in the *node_modules* directory.

```
$ npm install modulename
```

As an example, to install Request, use the following:

```
$ npm install request
```

npm not only installs Request, but it also discovers its module dependencies and installs them, too. And the more sophisticated the module, the more dependencies are installed. Figure 3-1 shows a partial listing from installing Request in my Windows machine (using Node 5.0.0 at the time).

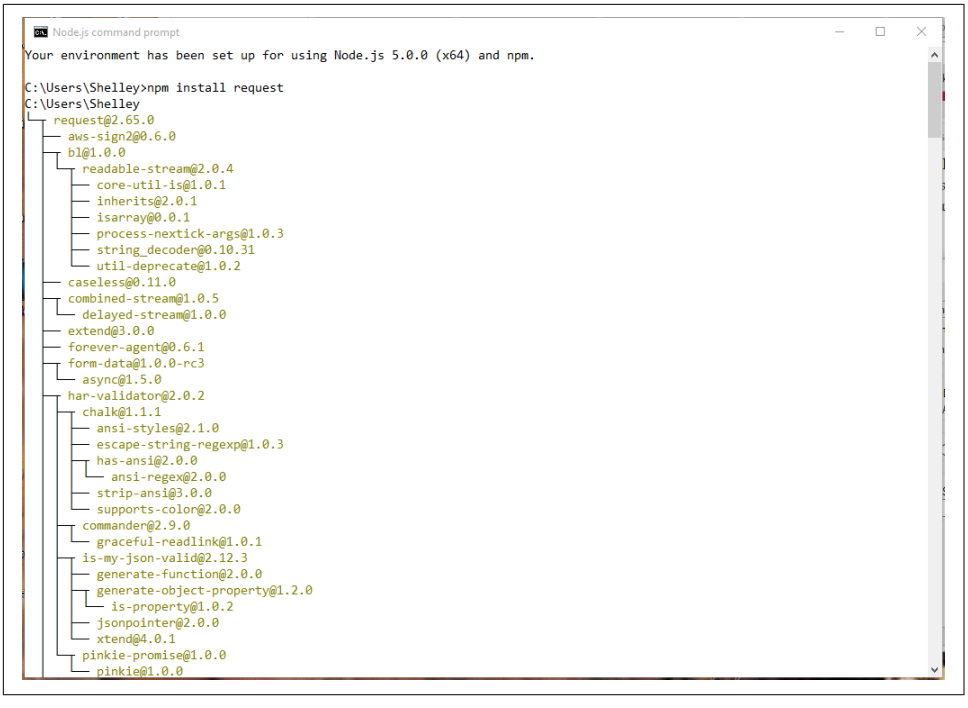


Figure 3-1. A partial listing of loaded dependencies for Request

Once the module is installed, you can find the module in your local directory's *node_modules* directory. Any dependencies are also installed in that module's *node_modules* directory. A glance shows you get a lot of bang for your buck when you install a module like Request.

If you want to install the package globally, use the `-g` or `--global` option:

```
$ npm install request -g
```

If you're working in Linux, don't forget to use `sudo` if you want to install a module globally:

```
$ sudo npm install request -g
```

Some modules, including command-line applications, may require global installation. These examples install packages that are registered at the npm site. You can also install a module that's in a folder on the filesystem, or a tarball that's either local or fetched via a URL:

```
npm install http://somecompany.com/somemodule.tgz
```

If the package has versions, you can install a specific version:

```
npm install modulename@0.1
```

You can even install an old friend, jQuery:

```
npm install jquery
```

If you're no longer using a module, you can uninstall it:

```
npm uninstall modulename
```

The following command tells npm to check for new modules, and perform an update if any are found:

```
npm update
```

Or you can update a single module:

```
npm update modulename
```

To update npm itself, use:

```
npm install npm -g
```

If you just want to check to see if any packages are outdated, use the following:

```
npm outdated
```

Again, you can run this command against a single module.

List installed packages and dependencies with `list`, `ls`, `la`, or `ll`:

```
npm ls
```

The `la` and `ll` options provide extended descriptions. For instance, one of the dependencies for Request is tunnel-agent@0.4.1 (version 0.4.1 of tunnel-agent). So what the heck is tunnel-agent? Running `npm la request` at the command line lists out the dependencies, including tunnel-agent, with additional detail:

```
tunnel-agent@0.4.1
HTTP proxy tunneling agent. Formerly part of mikeal/request,
now a standalone module
git+https://github.com/mikeal/tunnel-agent.git
https://github.com/mikeal/tunnel-agent#readme
```

Sometimes the output will display a warning, such as an unmet dependency, or a requirement for an older module. To correct, install the module, or the necessary version of the module:

```
npm install jsdom@0.2.0
```

You can also directly install all dependencies with the `-d` flag. For instance, in the directory for the module, type the following:

```
npm install -d
```

If you want to install a version of the module that hasn't yet been uploaded to the npm registry, you can install directly from the Git repository:

```
npm install https://github.com/visionmedia/express/tarball/master
```

Use caution, though, as I've found that when you install a not-yet-released version of a module and you do an npm update, the npm registry version can overwrite the version you're using.

Using npm in Linux via PuTTY

If you're working with Node/npm in Linux using the PuTTY application in Windows, you might notice that you don't get a clean printout when using the npm commands. Instead of nicely defined lines showing dependencies, you get strange characters.

To fix this problem, you need to instruct PuTTY to translate characters using UTF-8 encoding. In PuTTY, click the Window->Translation option, and select UTF-8 from the drop-down window.

To see which modules are installed globally, use:

```
npm ls -g
```

You can learn more about your npm installation using the `config` command. The following lists the npm configuration settings:

```
npm config list
```

You can get a more in-depth look at all configuration settings with:

```
npm config ls -l
```

You can modify or remove configuration settings either by using a command line:

```
npm config delete keyname
npm config set keyname value
```

or by directly editing the configuration file:

```
$ npm config edit
```

I would strongly recommend you leave your npm configuration settings alone, unless you're very sure of a change's effect.

You can also search for a module using whatever terms you think might return the best selection:

```
npm search html5 parser
```

The first time you do a search, npm builds an index, which can take a few minutes. When it's finished, though, you'll get a list of possible modules that match the term or terms you provided.

Getting "registry error parsing json" Results

If you're using npm and get a "registry error parsing json" error, you can use one of the npm mirrors to complete your task. For instance, to use the European mirror, use the following:

```
npm --registry http://registry.npmjs.eu/ search html5 parser
```

The npm website provides a registry of modules you can browse through, and an up-to-date listing of modules most depended on—that is, modules most used by other modules or by Node applications. In the next section, I'll cover a sampling of these modules.

One last note on npm before we move on. When you first start playing around with npm, you might notice a set of warning messages at the end of the output. The first line makes a note about not being able to find a *package.json* file, and all the rest of the warnings are associated with the missing *package.json* file.

The npm documentation recommends you create a *package.json* file to maintain your local dependencies. It's not a bothersome request, though the warnings are a bit of an irritant.

To create a default *package.json* file in the project directory, run the following command:

```
npm init --yes
```

This creates a default *package.json* file in the directory, asking you a set of basic project questions such as your name and the project name, each of which has a default value. From this point on, when you install a module, you'll no longer get most of the annoying messages. To eliminate the rest, you'll need to update the JSON in the file to include a `description` and a `repository`. In addition, if you want to update the file to reflect the newly installed module, use the following syntax:

```
npm install request --save-dev
```

This saves the module name and version to the `devDependencies` field in the *package.json* file. You can also save the module to production dependencies, but I'll get into more detail on the *package.json* file (in "Creating and Publishing Your Own Node Module" on page 69), where you create a Node module of your own.

To automatically save the dependencies, you can add and/or edit a *npmrc* file. You can add it per user (*~/.npmrc*), per project (*/path/project/.npmrc*), globally (*$PREFIX/etc/*

npmrc), and using the built-in configuration file (*/path/to/npm/npmrc*). Use the following to edit your personal settings to automatically save the dependencies:

```
npm config set save=true
npm config set save-exact=true
```

This automatically adds a `--save flag` (to save package in dependencies) and a `--save-exact` flag (saved with exact version, not npm's semver range default) when you install new packages.

There are also many different configuration settings you can adjust. A good place to start exploring these is in the npm documentation.

Creating and Publishing Your Own Node Module

Just as you do for your client-side JavaScript, you'll want to split off reusable JavaScript into its own libraries. The only difference is that you need to take a couple of extra steps to convert your JavaScript library into a module for use with Node.

Creating a Module

Let's say you have a JavaScript library function, `concatArray`, that takes a string and an array of strings, and concatenates the first string to each string in the array, returning a new array:

```
function concatArray(str, array) {
  return array.map(function(element) {
       return str + ' ' + element;
  });
}
```

You want to use this function, as well as others, in your Node applications.

To convert your JavaScript library for use in Node, you'll need to export all of your exposed functions using the `exports` object, as shown in the following code:

```
exports.concatArray = function(str, array) {
  return array.map(function(element) {
       return str + ' ' + element;
  });
};
```

To use `concatArray` in a Node application, import the library. The exported function can now be used in your application:

```
var newArray = require ('./arrayfunctions.js');

console.log(newArray.concatArray('hello', ['test1','test2']));
```

You can also create a module consisting of an object constructor or function, and export it using `module.exports`.

As an example, the heavily depended-on Mime module creates a function, `Mime()`:

```
function Mime() { ... }
```

adds functionality using the `prototype` property:

```
Mime.prototype.define = function(map) {...}
```

creates a default instance:

```
var mime = new Mime();
```

assigns the `Mime` function to its own same-named property:

```
mime.Mime = Mime;
```

and then exports the instance:

```
module.exports=mime;
```

You can then use all of the various mime functions in your application:

```
var mime = require('mime');
console.log(mime.lookup('phoenix5a.png')); // image/png
```

Packaging an Entire Directory

You can split your module into separate JavaScript files, all located within a directory. Node can load the directory contents, as long as you organize the contents in one of two ways.

The first way is to provide a *package.json* file with information about the directory. The structure contains other information, but the two entries relevant to packaging the module are `name` and `main`:

```
{ "name" : "mylibrary",
  "main" : "./mymodule/mylibrary.js"
}
```

The first property, `name`, is the name of the module. The second, `main`, indicates the entry point for the module.

The second way to load directory contents is to include either an *index.js* or *index.node* file in the directory to serve as the main module entry point.

Why would you provide a directory rather than just a single module? The most likely reason is that you're making use of existing JavaScript libraries, and just providing a "wrapper" file that wraps the exposed functions with `exports` statements. Another reason is that your library is so large that you want to break it down to make it easier to modify.

Regardless of the reason, be aware that all of the exported objects must be in the one main file that Node loads.

Preparing Your Module for Publication

If you want to make your package available to others, you can promote it on your website, but you'll be missing out on a significant audience. When you're ready to publish a module, you're going to want to publish it to the npm registry.

Earlier I mentioned the *package.json* file. The npm JSON documentation can be found online. It's actually based on the CommonJS module system recommendations.

Among the recommended fields to include in the *package.json* file are:

`name`
: The name of the package—required

`description`
: The package description

`version`
: The current version conforming to semantic version requirements—required

`keywords`
: An array of search terms

`maintainers`
: An array of package maintainers (includes name, email, and website)

`contributors`
: An array of package contributors (includes name, email, and website)

`bugs`
: The URL where bugs can be submitted

`licenses`
: An array of licenses

`repository`
: The package respository

`dependencies`
: Prerequisite packages and their version numbers

Only the `name` and `version` fields are required, though all of these fields are recommended. Thankfully, npm makes it easier to create this file. If you type the following at the command line:

```
npm init
```

the tool will run through the required/recommended fields, prompting you for each. When it's done, it generates a *package.json* file.

In Chapter 2, in Example 2-7, I created an object called `InputChecker` that checks incoming data for commands and then processes the command. The example demonstrated how to incorporate `EventEmitter`. Now we're going to modify this simple object to make it usable by other applications and modules.

First, we'll create a subdirectory in *node_modules* and name it *inputcheck*, and then move the existing `InputChecker` code file to it. We need to rename the file to *index.js*. Next, we need to modify the code to pull out the part that implements the new object. We'll save it for a future test file. The last modification we'll do is add the `exports` object, resulting in the code shown in Example 3-2.

Example 3-2. Application from Example 2-7 modified to be a module object

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

exports.InputChecker = InputChecker;

function InputChecker(name, file) {
   this.name = name;
   this.writeStream = fs.createWriteStream('./' + file + '.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});
};

util.inherits(InputChecker,eventEmitter);
InputChecker.prototype.check = function (input) {
  var command = input.toString().trim().substr(0,3);
  if (command == 'wr:') {
     this.emit('write',input.substr(3,input.length));
  } else if (command == 'en:') {
     this.emit('end');
  } else {
     this.emit('echo',input);
  }
};
```

We can't export the object function directly, because `util.inherits` expects an object to exist in the file named `InputChecker`. We're also modifying the `InputChecker` object's `prototype` later in the file. We could have changed these code references to use `exports.InputChecker`, but that's *kludgy*. It's just as easy to assign the object in a separate statement.

To create the *package.json* file, I ran `npm init` and answered each of the prompts. The resulting file is shown in Example 3-3.

Example 3-3. Generated package.json for inputChecker module

```
{
  "name": "inputcheck",
  "version": "1.0.0",
  "description": "Looks for and implements commands from input",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "command",
    "check"
  ],
  "author": "Shelley Powers",
  "license": "ISC"
}
```

The `npm init` command doesn't prompt for dependencies, so we need to add them directly to the file. A better approach, however, is to use `npm install --save` when installing the modules, which will automatically add the dependencies for us. However, the `InputChecker` module isn't dependent on any external modules, so we can leave these fields blank.

At this point, we can test the new module to make sure it actually works as a module. Example 3-4 is the portion of the previously existing `InputChecker` application that tested the new object, now pulled out into a separate test application.

Example 3-4. InputChecker test application

```
var inputChecker = require('inputcheck').InputChecker;

// testing new object and event handling
var ic = new inputChecker('Shelley','output');

ic.on('write', function(data) {
   this.writeStream.write(data, 'utf8');
});

ic.addListener('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});

ic.on('end', function() {
   process.exit();
});

process.stdin.resume();
process.stdin.setEncoding('utf8');
process.stdin.on('data', function(input) {
```

```
    ic.check(input);
});
```

We can now move the test application into a new *test* subdirectory within the module directory to be packaged with the module as an example. Good practice demands that we also provide a *test* directory with one or more testing applications, as well as a *doc* directory with documentation. For a module this small, a *README* file should be sufficient.

Since we now have a test application, we'll need to modify the *package.json* file to include a reference to it:

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

The test is rather primitive, and isn't using one of the robust Node testing environments, but it does demonstrate how testing can work. To run the test application, type the following at the command line:

```
npm test
```

Lastly, we create a gzipped tarball of the module. You can skip this step, though, when you publish the module, discussed in "Publishing the Module" on page 74.

What Is This Tarball Thing?

I've used the term *tarball* in this and previous chapters. If you've not worked in Unix, you may not be familiar with the term. A tarball is one or more files archived together into a single file using a `tar` command.

Once we've provided all we need to provide, we can publish the module.

Providing More Than Just a Module

You can, minimally, get by with a simplified module like `inputcheck`, but you'll want to provide a lot more before going online with a module. You'll need to provide a repository for your module, a URL for reporting bugs, a home page for the modules, and so on. Still, start simple, and work up.

Publishing the Module

The folks who brought us npm also provide a guide detailing what's needed to publish a Node module: the Developer Guide.

The documentation specifies some additional requirements for the *package.json* file. In addition to the fields already created, we also need to add in a `directories` field with a hash of folders, such as the previously mentioned *test* and *doc*:

```
"directories" : {
    "doc" : ".",
    "test" : "test",
    "example" : "examples"
}
```

Before publishing, the Guide recommends we test that the module can cleanly install. To test for this, type the following in the root directory for the module:

```
npm install . -g
```

At this point, we've tested the `inputChecker` module, modified the *package.json* package to add directories, and confirmed that the package successfully installs.

Next, we need to add ourselves as npm users if we haven't done so already. We do this by typing:

```
npm adduser
```

and then following the prompts to add a username, a password, and an email address.

There's one last thing to do:

```
npm publish
```

We can provide the path to the tarball or the directory. As the Guide warns us, everything in the directory is exposed unless we use a `.npmignore` with the file(s) listed in the *package.json* file to ignore material. It's better, though, just to remove anything that's not needed before publishing the module.

Once published—and once the source is also uploaded to GitHub (if that's the repository you're using)—the module is now officially ready for others to use. Promote the module on Twitter, Google+, Facebook, your website, and wherever else you think people would want to know about the module. This type of promotion isn't bragging—it's *sharing*.

Discovering Node Modules and Three Must-Have Modules

It's not difficult to find Node modules. Most that you discover will be ones friends and coworkers recommend. Others you might discover by searching for specific functionality using a search engine and they show up at the top in the search results.

At one time, there were few enough modules that the best of the different types were manually listed on a single page. Well, those days are long gone. Nowadays, if you're looking for modules, you'll most likely do so at the npm website. Not only can you

find documentation for npm at the site, but it also provides a searchable directory of Node modules.

If you're looking for modules to support the OAuth protocol, enter this as the search phrase in the search bar at the top of the page. The top result is named simply Oauth, and is the most popular OAuth module in use today. You can tell by the statistics displayed for the module.

You will want to check out the module statistics. The biggest challenge when you're searching for Node modules at npm is that many of the modules are no longer supported, or aren't considered best in breed. The only way you can tell which modules are actively supported and most widely used are the statistics listed in the right side of the page. Figure 3-2 is an example, for the OAuth module.

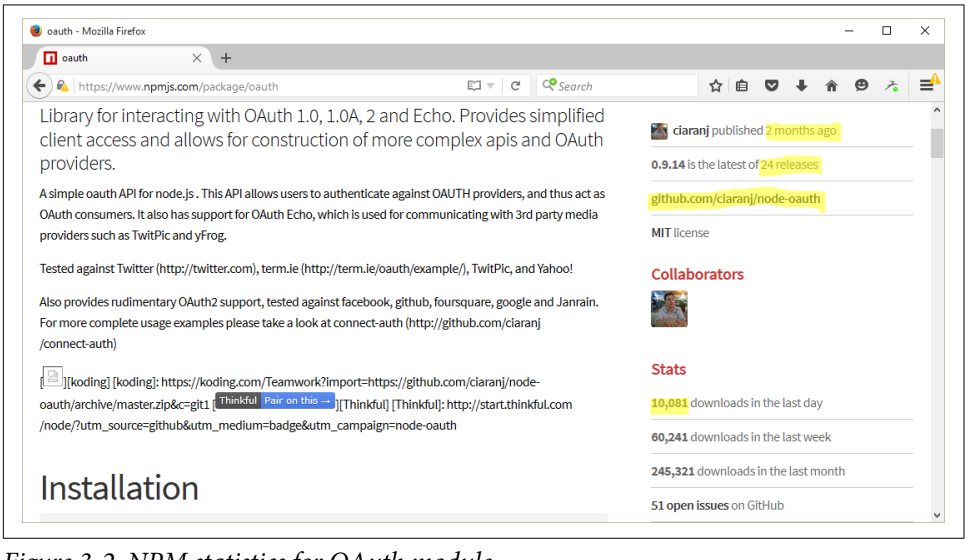


Figure 3-2. NPM statistics for OAuth module

Note the highlighted text. The module currently available was published in the last couple of months, which demonstrates that there have been recent updates. It's the last of 24 releases, which shows an active involvement. The kicker, though, is the number of downloads: over 10,000 in just a single day. That's a very actively used module, indicating that it is viable and likely maintained.

In addition, you can access the GitHub page and check out the issues to see how involved the developer(s) is with those reporting concerns or whether there are issues that make the module not a good choice (such as a security issue).

There's no question that the OAuth module is most likely both the most actively supported and best-of-breed module. That's not to say there aren't other good options,

but until you're more familiar with Node, you'll probably want to stick with the most frequently used Node modules.

Speaking of commonly used modules, the npm website also lists, on the very first page, the most popular Node modules in use today. The site also contains a list of the most-starred modules, as well as the modules most depended upon. The latter set of modules are those that are installed as dependencies for other modules. In the following five sections, I'm covering three modules you'll find on both lists: Async, Commander, and Underscore. These are in addition to other modules I use throughout the book.

Better Callback Management with Async

The application in Example 2-11, in Chapter 2, is an asynchronous pattern, where each function is called in turn and passes its results to the next function, and the entire chain stops only if an error occurs. There are several such patterns, though some are variations of others, and not everyone uses the exact same terminology.

The example also demonstrated a major problem with the use of callbacks: the *pyramid of doom* effect as the nested callbacks progressively push out to the right.

One of the most commonly used modules to work around this effect is the Async module. It replaces the typical callback pattern with more linear and manageable ones. Among the asynchronous patterns Async supports are:

`waterfall`
: Functions are called in turn, and results of all are passed as an array to the last callback (also called `series` and `sequence` by others).

`series`
: Functions are called in turn and, optionally, results are passed as an array to the last callback.

`parallel`
: Functions are run in parallel and when completed, results are passed to the last callback (though the result array isn't part of the pattern in some interpretations of the parallel pattern).

`whilst`
: Repeatedly calls one function, invoking the last callback only if a preliminary test returns `false` or an error occurs.

`queue`
: Calls functions in parallel up to a given limit of concurrency, and new functions are queued until one of the functions finishes.

`until`
: Repeatedly calls one function, invoking the last callback only if a post-process test returns `false` or an error occurs.

`auto`
: Functions are called based on requirements, each function receiving the results of previous callbacks.

`iterator`
: Each function calls the next, with the ability to individually access the next iterator.

`apply`
: A continuation function with arguments already applied combined with other control flow functions.

`nextTick`
: Calls the callback in the next loop of an event loop—based on `process.nextTick` in Node.

The Async module also provides functionality for managing collections, such as its own variation of `forEach`, `map`, and `filter`, as well as other utility functions, including ones for *memoization*. However, what we're interested in here are its facilities for handling control flow.

Async has a GitHub repository.

Install Async using npm. If you want to install it globally, use the `-g` flag. If you want to update your dependencies, use `--save` or `--save-dev`:

```
npm install async
```

As mentioned earlier, Async provides control flow capability for a variety of asynchronous patterns, including `serial`, `parallel`, and `waterfall`. As an example, the pattern of the example in Chapter 2 matches with Async's `waterfall`, so we'll be using the `async.waterfall` method. In Example 3-5, I used `async.waterfall` to open and read a data file using `fs.readFile`, perform the synchronous string substitution, and then write the string back to the file using `fs.writeFile`. Pay particular attention to the callback function used with each step in the application.

Example 3-5. Using async.waterfall to read, modify, and write a file's contents asynchronously

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

async.waterfall([
   function readData(callback) {
      fs.readFile('./data/data1.txt', 'utf8', function(err, data){
           callback(err,data);
       });
   },
   function modify(text, callback) {
      var adjdata=text.replace(/somecompany\.com/g,'burningbird.net');
      callback(null, adjdata);
   },
   function writeData(text, callback) {
       fs.writeFile('./data/data1.txt', text, function(err) {
          callback(err,text);
       });
   }
], function (err, result) {
      if (err) {
        console.error(err.message);
      } else {
        console.log(result);
      }
});
```

The `async.waterfall` takes two parameters: an array of tasks and an optional final callback function. Each asynchronous task function is an element of the `async.waterfall` array, and each function requires a callback as the last of its parameters. It is this callback function that allows us to chain the asynchronous callback results without having to physically nest the functions. However, as you can see in the code, each function's callback is handled as we would normally handle it if we were using nested callbacks—other than the fact that we don't have to test the errors in each function. Async looks for an error object as the first parameter in each callback. If we pass an error object in the callback, the process is ended at this point, and the final callback routine is called. The final callback is when we can process the error or the final result.

Example 3-5 uses named functions, whereas the Async documentation shows anonymous functions. Named functions can simplify debugging and error handling, but both work equally well.

The processing is very similar to what we had in Chapter 2 but without the nesting (and having to test for an error in each function). It may seem more complicated, and I wouldn't recommend its use for such simple nesting, but look what it can do with a more complex nested callback. Example 3-6 duplicates the exact functionality from Chapter 2, but without the callback nesting and excessive indenting.

Example 3-6. Get objects from directory, test to look for files, read file test, modify, and write back out log results

```
var fs = require('fs'),
    async = require('async'),
    _dir = './data/';

var writeStream = fs.createWriteStream('./log.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});

async.waterfall([
   function readDir(callback) {
      fs.readdir(_dir, function(err, files) {
         callback(err,files);
      });
   },
   function loopFiles(files, callback) {
      files.forEach(function (name) {
         callback (null, name);
      });
   },

   function checkFile(file, callback) {
      fs.stat(_dir + file, function(err, stats) {
         callback(err, stats, file);
      });
   },
   function readData(stats, file, callback) {
      if (stats.isFile())
         fs.readFile(_dir + file, 'utf8', function(err, data){
           callback(err,file,data);
       });
   },
   function modify(file, text, callback) {
      var adjdata=text.replace(/somecompany\.com/g,'burningbird.net');
      callback(null, file, adjdata);
   },
   function writeData(file, text, callback) {
       fs.writeFile(_dir + file, text, function(err) {
          callback(err,file);
       });
   },
   function logChange(file, callback) {
```

```
        writeStream.write('changed ' + file + '\n', 'utf8',
                          function(err) {
            callback(err);
        });
    }
], function (err) {
        if (err) {
            console.error(err.message);
        } else {
            console.log('modified files');
        }
});
```

Every last bit of functionality is present from the Chapter 2 example. The `fs.readdir` method is used to get an array of directory objects. The Node `forEach` method (not the Async `forEach`) is used to access each specific object. The `fs.stats` method is used to get the `stats` for each object. `stats` is used to check for files, and when a file is found, it's opened and its data accessed. The data is then modified, and passed on to be written back to the file via `fs.writeFile`. The operation is logged in the logfile and a successful completion is echoed to the console.

Note that more data is passed in some of the callbacks. Most of the functions need the filename as well as the text, so this is passed in the last several methods. Any amount of data can be passed in the methods, as long as the first parameter is the error object (or `null`, if no error object) and the last parameter in each function is the callback function. We don't have to check for an error in each asynchronous task function, because Async tests the error object in each callback, stops processing, and calls the final callback function if an error is found.

The other Async control flow methods, such as `async.parallel` and `async.serial`, perform in a like manner, with an array of tasks as the first method parameter and a final optional callback as the second. How they process the asynchronous tasks differs, though, as you would expect.

The `async.parallel` method calls all of the asynchronous functions at once, and when they are all finished, calls the optional final callback. Example 3-7 uses `async.parallel` to read in the contents of three files in parallel. However, rather than an array of functions, this example uses an alternative approach that Async supports: passing in an object with each asynchronous task listed as a property of the object. The results are then printed out to the console when all three tasks have finished.

Example 3-7. Opening three files in parallel and reading in their contents

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

async.parallel({
```

```
    data1 : function (callback) {
       fs.readFile('./data/fruit1.txt', 'utf8', function(err, data){
            callback(err,data);
        });
    },
    data2 : function (callback) {
       fs.readFile('./data/fruit2.txt', 'utf8', function(err, data){
            callback(err,data);
        });
    },
    data3 : function readData3(callback) {
       fs.readFile('./data/fruit3.txt', 'utf8', function(err, data){
            callback(err,data);
        });
    },


}, function (err, result) {
      if (err) {
         console.log(err.message);
      } else {
         console.log(result);
      }
});
```

The results are returned as an array of objects, with each result tied to each of the properties. If the three data files in the example had the following content:

- *fruit1.txt*: apples
- *fruit2.txt*: oranges
- *fruit3.txt*: peaches

the result of running Example 3-7 is:

```
{ data1: 'apples\n', data2: 'oranges\n', data3: 'peaches\n' }
```

I'll leave the testing of the other Async control flow methods as a reader exercise. Just remember that when you're working with the Async control flow methods, all you need is to pass a callback to each asynchronous task and to call this callback when you're finished, passing in an error object (or `null`) and whatever data you need.

Command-Line Magic with Commander

Commander provides support for command-line options. These are the flags we provide when running an application, such as the ubiquitous `-h` or `--help` for how to use a utility or application.

Install it via npm:

```
npm install commander
```

And include it using `require`:

```
var program = require('commander');
```

To use it, chain option calls listing out all of the various options supported for the application. In the following code, the application supports two default options—`-V` or `--version` for version and `-h` or `--help` for help—as well as two custom option: `-s` or `--source` for source website, and `-f` or `--file` for a filename:

```
var program = require('commander');

program
   .version ('0.0.1')
   .option ('-s, --source [web site]', 'Source web site')
   .option ('-f, --file [file name]', 'File name')
   .parse(process.argv);

console.log(program.source);
console.log(program.file);
```

We provide the custom options, while Commander provides the default version and help. If you run the application using:

```
node options -h
```

Commander prints out the available options:

```
  Usage: options [options]

  Options:

    -h, --help               output usage information
    -V, --version            output the version number
    -s, --source [web site]  Source web site
    -f, --file [file name]   File name
```

Short options can be concatenated, such as `-sf`, and Commander can handle it. It also takes a multiwork option, such as `--file-name`, and uses Camel-casing with the result: `program.fileName`.

Commander also supports coercion (type casting):

```
.option('-i, --integer <n>', 'An integer argument', parseInt)
```

regular expressions:

```
.option('-d --drink [drink]', 'Drink', /^(coke|pepsi|izze)$/i)
```

as well as a *variadic* argument in the last option, which means it will take any number of arguments. Perhaps your application can support multiple keywords, of unknown number. You can create a Command option for this using:

```
var program = require('commander');

program
  .version('0.0.1')
  .command('keyword <keywork> [otherKeywords...]')
  .action(function (keyword, otherKeywords) {
    console.log('keyword %s', keyword);
    if (otherKeywords) {
      otherKeywords.forEach(function (oKey) {
        console.log('keyword %s', oKey);
      });
    }
  });

program.parse(process.argv);
```

The following command line:

```
node options2 keyword one two three
```

results in the following printout:

```
keyword one
keyword two
keyword three
```

Download Commander

If you want to download Commander directly, you can do so at its GitHub repository. Commander is especially useful for command-line applications, as demonstrated in Chapter 6.

The Ubiquitous Underscore

Install the Underscore module with:

```
npm install underscore
```

According to the developers, Underscore is a utility-belt library for Node. It provides a lot of extended JavaScript functionality that we're used to with third-party libraries, such as jQuery or Prototype.js.

Underscore is so named because, traditionally, its functionality is accessed with an underscore (_), similar to jQuery's $. Here's an example:

```
var _ = require('underscore');
_.each(['apple','cherry'], function (fruit) { console.log(fruit); });
```

Of course, the problem with the underscore is that this character has a specific meaning in REPL, as you'll see in the next chapter. No worries, though—we can use another variable, us:

www.allitebooks.com

```
var us = require('underscore');
us.each(['apple','cherry'], function(fruit) { console.log(fruit); });
```

Underscore provides expanded functionality for arrays, collections, functions, objects, chaining, and general utility. Fortunately, there's also excellent documentation for all of its functionality, available at its website, so I'll forgo detailing any of it here.

However, I do want to mention one nice capability: a controlled way to extend Underscore with your own utility functions via the `mixin` function. We can quickly try this method, and the others, in a REPL session:

```
> var us = require('underscore');
> us.mixin({
... betterWithNode: function(str) {
..... return str + ' is better with Node';
..... }
... });
> console.log(us.betterWithNode('chocolate'));
chocolate is better with Node
```

You'll see the term *mixin* used in several Node modules. It's based on a pattern where properties of one object are added ("mixed in") to another.

Underscore isn't the only top-rated utility module. Another is lodash. Unless you've worked with one or the other, I'd check both out. The lodash website contains complete documentation for this useful module.

CHAPTER 4

Interactive Node with REPL and More on the Console

While you're exploring the use of Node and figuring out the code for your custom module or Node application, you don't have to type JavaScript into a file and run it with Node to test your code. Node also comes with an interactive component known as *REPL*, or *read-eval-print loop.*

REPL (pronounced "repple") supports simplified line editing and a small set of basic commands. Whatever you type into REPL is, for the most part, processed no differently than if you had typed the JavaScript into a file and run the file using Node. You can actually use REPL to code your entire application—literally testing the application on the fly.

In this chapter, I'll cover how to use REPL, as well as some interesting quirks of REPL and how to work with them. These workarounds include replacing the underlying mechanism that persists commands, as well as using some command-line editing. And if the built-in REPL doesn't provide exactly what you need for an interactive environment, there's also an API to create your own custom REPL.

REPL is an essential Node development tool, and so is the console. We're using the console in most applications in the book, but there's more to this helpful object than just logging messages.

REPL: First Looks and Undefined Expressions

To begin REPL, simply type **node** without providing any Node application file, like so:

```
$ node
```

REPL then provides a command-line prompt—an angle bracket (>)—by default. Anything you type from this point on is processed by the underlying V8 JavaScript engine.

REPL is very simple to use. Just start typing in your JavaScript, like you'd add it to a file:

```
> a = 2;
2
```

The tool prints out the result of whatever expression you just typed. In this session excerpt, the value of the expression is `2`. In the following, the expression result is an array with three elements:

```
> b = ['a','b','c'];
[ 'a', 'b', 'c' ]
```

To access the last expression, use the underscore/underline special variable (_). In the following, `a` is set to `2`, and the resulting expression is incremented by 1, and then 1 again:

```
> a = 2;
2
> ++_;
3
> ++_;
4
```

You can even access properties or call methods on the underscored expression:

```
> ['apple','orange','lime']
[ 'apple', 'orange', 'lime' ]
> _.length
3
> 3 + 4
7
> _.toString();
'7'
```

You can use the `var` keyword with REPL in order to access an expression or value at a later time, but you might get an unexpected result. For instance, consider the following line in REPL:

```
var a = 2;
```

It doesn't return the value `2`; it returns a value of `undefined`. The reason is that the result of the expression is undefined since variable assignment doesn't return a result when evaluated.

Consider the following instead, which is what's happening, more or less, under the hood in REPL:

```
console.log(eval('a = 2'));
console.log(eval('var a = 2'));
```

Typing the preceding line into a file and running that file using Node returns:

```
2
undefined
```

There is no result from the second call to `eval`; hence, the value returned is `undefined`. REPL is a read-eval-print loop, with emphasis on the *eval*.

You can use the variable in REPL, just as you would in a Node application:

```
> var a = 2;
undefined
> a++;
2
> a++;
3
```

The latter two command lines do have results, which are printed out by REPL.

I'll demonstrate how to create your own custom REPL—one that doesn't output `undefined`—in "Custom REPL" on page 96.

To end the REPL session, either press Ctrl-C twice, or Ctrl-D once. We'll cover other ways to end the session later, in "REPL Commands" on page 94.

Benefits of REPL: Getting a Closer Understanding of JavaScript Under the Hood

Here's a typical demonstration of REPL:

```
> 3 > 2 > 1;
false
```

This code snippet is a good example of how REPL can be useful. At first glance, we might expect the expression just typed to evaluate to `true`, since 3 is greater than 2, which is greater than 1. However, in JavaScript, relational operators are evaluated left to right, and each expression's result is returned for the next evaluation.

A better way of looking at what's happening with the preceding code snippet is this REPL session:

```
> 3 > 2 > 1;
false
> 3 > 2;
true
> true > 1;
false
```

Now the result makes more sense. What's happening is that the expression 3 > 2 is evaluated, returning `true`. But then the value of `true` is compared to the numeric 1. JavaScript provides automatic data type conversion, after which `true` and 1 are equivalent values. Hence, `true` is not greater than 1, and the result is `false`.

REPL's helpfulness is in enabling us to discover these little interesting quirks in JavaScript and our code. Hopefully, after testing our code in REPL, we don't have unexpected side effects in our applications (such as expecting a result of `true` but getting a result of `false`).

Multiline and More Complex JavaScript

You can type the same JavaScript into REPL just like you'd type it into a file, including `require` statements to import modules. A session to try out the Query String (`qs`) module is repeated in the following text:

```
$ node
> qs = require('querystring');
{ unescapeBuffer: [Function],
  unescape: [Function],
  escape: [Function],
  encode: [Function],
  stringify: [Function],
  decode: [Function],
  parse: [Function] }
> val = qs.parse('file=main&file=secondary&test=one').file;
[ 'main', 'secondary' ]
```

Since we didn't use the `var` keyword, the expression result is printed out—in this instance, it is the interface for the `querystring` object. How's that for a bonus? Not only are you getting access to the object, but you're also learning more about the object's interface while you're at it. However, if you want to forgo the potentially lengthy output of text, use the `var` keyword:

```
> var qs = require('querystring');
```

You'll be able to access the `querystring` object with the `qs` variable with either approach.

In addition to being able to incorporate external modules, REPL gracefully handles multiline expressions, providing a textual indicator of code that's nested following an opening curly brace ({):

```
> var test = function (x, y) {
... var val = x * y;
... return val;
... };
undefined
> test(3,4);
12
```

REPL displays repeating dots to indicate that everything that's being typed follows an open curly brace and hence the command isn't finished yet. It does the same for an open parenthesis, too:

```
> test(4,
... 5);
20
```

Increasing levels of nesting generates more dots; this is necessary in an interactive environment, where you might lose track of where you are as you type:

```
> var test = function (x, y) {
... var test2 = function (x, y) {
..... return x * y;
..... }
... return test2(x,y);
... }
undefined
> test(3,4);
12
>
```

You can type in, or copy and paste in, an entire Node application and run it from REPL. I trimmed the actual display values for the Server, shown in bold, both because they're so long, and because they'll most likely change by the time you read this:

```
> var http = require('http');
undefined
> http.createServer(function (req, res) {
...
...   // content header
...   res.writeHead(200, {'Content-Type': 'text/plain'});
...
...   res.end("Hello person\n");
... }).listen(8124);
{ domain:
   { domain: null,
     _events: { error: [Function] },
     _maxListeners: undefined,
     members: [] },
  _
...
  httpAllowHalfOpen: false,
  timeout: 120000,
  _pendingResponseData: 0,
```

```
  _connectionKey: '6:null:8124' }
> console.log('Server running at http://127.0.0.1:8124/');
Server running at http://127.0.0.1:8124/
Undefined
```

You can access this application from a browser no differently than if you had typed the text into a file and run it using Node.

The "downside" of not assigning an expression to the variable is that you are going to get what could be a long object display in the midst of your application. However, one of my favorite uses of REPL is to get a quick look at objects. For instance, the Node core object `global` is sparsely documented at the Node.js website. To get a better look, I opened up a REPL session and passed the object to the `console.log` method like so:

```
> console.log(global)
```

I could have done the following, which has a similar result, with the addition of the gl variable (trimmed, for space):

```
> gl = global;
...
_: [Circular],
  gl: [Circular] }
```

Just printing out `global` also provides the same information:

```
> global
```

I'm not replicating what was displayed in REPL. I'll leave that for you to try on your own installation, since the interface for `global` is so large. The important point to take away from this exercise is that we can, at any time, quickly and easily get a look at an object's interface. It's a handy way of remembering what a method is called or what properties are available.

There's more on `global` in Chapter 2.

You can use the up and down arrow keys to traverse through the commands you've typed into REPL. This can be a handy way of reviewing what you've done, as well as a way of editing what you've typed, though in a somewhat limited capacity.

Consider the following session in REPL:

```
> var myFruit = function(fruitArray,pickOne) {
... return fruitArray[pickOne - 1];
... }
undefined
> fruit = ['apples','oranges','limes','cherries'];
```

```
[ 'apples',  'oranges', 'limes',  'cherries' ]
> myFruit(fruit,2);
'oranges'
> myFruit(fruit,0);
undefined
> var myFruit = function(fruitArray,pickOne) {
... if (pickOne <= 0) return 'invalid number';
... return fruitArray[pickOne - 1];
... };
undefined
> myFruit(fruit,0);
'invalid number'
> myFruit(fruit,1);
'apples'
```

Though it's not demonstrated in this printout, when I modified the function to check the input value, I actually arrowed up through the content to the beginning function declaration, and then hit Enter to restart the function. I added the new line, and then again used the arrow keys to repeat previously typed entries until the function was finished. I also used the up arrow key to repeat the function call that resulted in an `undefined` result.

It seems like a lot of work just to avoid retyping something so simple, but consider working with regular expressions, such as the following:

```
> var ssRe = /^\d{3}-\d{2}-\d{4}$/;
undefined
> ssRe.test('555-55-5555');
true
> var decRe = /^\s*(\+|-)?((\d+(\.\d+)?)|(\.\d+))\s*$/;
undefined
> decRe.test(56.5);
true
```

I'm absolutely useless when it comes to regular expressions, and have to tweak them several times before they're just right. Using REPL to test regular expressions is very attractive. However, retyping long regular expressions would be a monstrous amount of work.

Luckily, all we have to do with REPL is arrow up to find the line where the regular expression was created, tweak it, hit Enter, and continue with the next test.

In addition to the arrow keys, you can also use the Tab key to *autocomplete* a command if there's no confusion as to what you're completing. As an example, type **`va`** at the command line and then press Tab; REPL lists out both `var` and `valueOf`, both of which can complete the typing. However, typing `querys` and hitting Tab returns the only available option, `querystring`. You can also use the Tab key to autocomplete any global or local variable. Table 4-1 offers a quick summary of keyboard commands that work with REPL.

Table 4-1. Keyboard control in REPL

Keyboard entry	What it does
Ctrl-C	Terminates current command. Pressing Ctrl-C twice forces an exit.
Ctrl-D	Exits REPL.
Tab	Autocompletes global or local variable.
Up arrow	Traverses up through command history.
Down arrow	Traverses down through command history.
Underscore (_)	References result of last expression.

If you're concerned about spending a lot of time coding in REPL with nothing to show for it when you're done, no worries: you can save the results of the current context with the `.save` command. It and the other REPL commands are covered in the next section.

REPL Commands

REPL has a simple interface with a small set of useful commands. In the preceding section, I mentioned `.save`. The `.save` command saves your inputs in the current object context into a file. Unless you specifically created a new object context or used the `.clear` command, the context should comprise all of the input in the current REPL session:

```
> .save ./dir/session/save.js
```

Only your inputs are saved, as if you had typed them directly into a file using a text editor.

Here is the complete list of REPL commands and their purposes:

`.break`
: If you get lost during a multiline entry, typing `.break` will start you over again. You'll lose the multiline content, though.

`.clear`
: Resets the context object and clears any multiline expression. This command basically starts you over again.

`.exit`
: Exits REPL.

`.help`
: Displays all available REPL commands.

`.save`
: Saves the current REPL session to a file.

`.load`
: Loads a file into the current session (`.load /path/to/file.js`).

If you're working on an application using REPL as an editor, here's a hint: save your work often using `.save`. Though current commands are persisted to history, trying to re-create your code from history would be a painful exercise.

Speaking of persistence and history, now let's go over how to customize both with REPL.

REPL and rlwrap

The Node.js website documentation for REPL mentions setting up an environmental variable so you can use REPL with `rlwrap`. What is `rlwrap`, and why would you use it with REPL?

The `rlwrap` utility is a wrapper that adds GNU `readline` library functionality to command lines that allow increased flexibility with keyboard input. It intercepts keyboard input and provides additional functionality, such as enhanced line editing, as well as a persistent history of commands.

You'll need to install `rlwrap` and `readline` to use this facility with REPL, and most flavors of Unix provide an easy package installation. For instance, in my own Ubuntu system, installing `rlwrap` was this simple:

```
apt-get install rlwrap
```

Mac users should use the appropriate installer for these applications. Windows users have to use a Unix environmental emulator, such as Cygwin.

Here's a quick and visual demonstration of using REPL with `rlwrap` to change the REPL prompt to purple:

```
NODE_NO_READLINE=1 rlwrap -ppurple node
```

If I always want my REPL prompt to be purple, I can add an alias to my *bashrc* file:

```
alias node="NODE_NO_READLINE=1 rlwrap -ppurple node"
```

To change both the prompt and the color, I'd use the following:

```
NODE_NO_READLINE=1 rlwrap -ppurple -S "::> " node
```

Now my prompt would be:

```
::>
```

but in purple.

The especially useful component of `rlwrap` is its ability to persist history across REPL sessions. By default, we have access to command-line history only within a REPL session. By using `rlwrap`, the next time we access REPL, we'll have access not only to a

history of commands within the current session, but also to a history of commands in past sessions (and other command-line entries). In the following session output, the commands shown were not typed in, but were instead pulled from history with the up arrow key, *after* I had exited REPL and then re-entered it:

```
::> e = ['a','b'];
[ 'a', 'b' ]
::> 3 > 2 > 1;
false
```

As helpful as `rlwrap` is, we still end up with `undefined` every time we type in an expression that doesn't return a value. However, we can adjust this, and other functionality, just by creating our own custom REPL, discussed next.

Custom REPL

Node provides an API that we can use to create a custom REPL. To do so, first we need to include the REPL module (`repl`):

```
var repl = require("repl");
```

To create a new REPL, we call the `start` method on the `repl` object. The syntax for this method is:

```
repl.start(options);
```

The `options` object takes several values; the ones I want to focus on are:

`prompt`
: Default is `>`.

`input`
: Readable stream; default is `process.stdin`.

`output`
: writable stream; default is `process.stdout`.

`eval`
: Default is an `async` wrapper for `eval`.

`useGlobal`
: Default is `false` to start a new context rather than use the global object.

`useColors`
: Whether `writer` function should use colors. Defaults to REPL's `terminal` value.

`ignoreUndefined`
: Default is `false`: don't ignore the `undefined` responses.

`terminal`
: Set to `true` if stream should be treated like a tty (terminal), including support for ANSI/VT100 escape codes.

`writer`
: Function to evaluate each command, and return formatting. Defaults to `util.inspect`.

`replMode`
: Whether REPL runs in strict mode, default, or hybrid.

Starting in Node 5.8.0, `repl.start()` no longer requires an `options` object.

I find the `undefined` expression result in REPL to be unedifying, so I created my own REPL. I also redefined the prompt and set the mode to strict, which means each line executed is done so under "use strict."

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

repl.start( {
  prompt: 'my repl> ',
  replMode: repl.REPL_MODE_STRICT,
  ignoreUndefined: true,
});
```

I ran the file, *repl.js*, using Node:

```
node repl
```

I can use the custom REPL just like I use the built-in version, except now I have a different prompt and no longer get the annoying `undefined` after the first variable assignment. I do still get the other responses that aren't `undefined`:

```
my repl> let ct = 0;
my repl> ct++;
0
my repl> console.log(ct);
1
my repl> ++ct;
2
my repl> console.log(ct);
2
```

In my code, I wanted the defaults used for all but the listed properties. Not listing any other property in the `options` object causes each to use the default.

You can replace the `eval` function with your custom REPL. The only requirement is that it has a specific format:

```
function eval(cmd, callback) {
   callback(null, result);
}
```

The `input` and `output` options are interesting. You can run multiple versions of REPL, taking input from both the standard input (the default), as well as sockets. The documentation for REPL at the Node.js site provides an example of a REPL listening in on a TCP socket, using the following code:

```
var net = require("net"),
    repl = require("repl");

connections = 0;

repl.start({
  prompt: "node via stdin> ",
  input: process.stdin,
  output: process.stdout
});

net.createServer(function (socket) {
  connections += 1;
  repl.start({
    prompt: "node via Unix socket> ",
    input: socket,
    output: socket
  }).on('exit', function() {
    socket.end();
  })
}).listen("/tmp/node-repl-sock");

net.createServer(function (socket) {
  connections += 1;
  repl.start({
    prompt: "node via TCP socket> ",
    input: socket,
    output: socket
  }).on('exit', function() {
    socket.end();
  });
}).listen(5001);
```

When you run the application, you get the standard input prompt where the Node application is running. However, you can also access REPL via TCP. I used PuTTY as a Telnet client to access this TCP-enabled version of REPL. It does work…to a point. I had to issue a `.clear` first, the formatting is off, and when I tried to use the underscore to reference the last expression, Node didn't know what I was talking about. I

also tried with the Windows Telnet client, and the response was even worse. However, using my Linux Telnet client worked without a hitch.

The problem here, as you might expect, is Telnet client settings. However, I didn't pursue it further, because running REPL from an exposed Telnet socket is not something I plan to implement, and not something I would recommend, either—at least, not without heavy security. It's like using `eval()` in your client-side code, and not scrubbing the text your users send you to run—but worse.

You could keep a running REPL and communicate via a Unix socket with something like the GNU Netcat utility:

```
nc -U /tmp/node-repl-sock
```

You can type in commands no differently than typing them in using `stdin`. Be aware, though, if you're using either a TCP or Unix socket, that any `console.log` commands are printed out to the server console, not to the client:

```
console.log(someVariable); // actually printed out to server
```

An application option that I consider to be more useful is to create a REPL application that preloads modules. In the application in Example 4-1, after the REPL is started, the third-party modules of Request (powerful HTTP client), Underscore (utility library), and Q (promise management) modules are loaded and assigned to context properties.

Example 4-1. Creating a custom REPL that preloads modules

```
var repl = require('repl');
var context = repl.start({prompt: '>> ',
                          ignoreUndefined: true,
                          replMode: repl.REPL_MODE_STRICT}).context;

// preload in modules
context.request = require('request');
context.underscore = require('underscore');
context.q = require('q');
```

Running the application with Node brings up the REPL prompt, where we can then access the modules:

```
>> request('http://blipdebit.com/phoenix5a.png')
.pipe(fs.createWriteStream('bird.png'))
```

The core Node modules don't need to be specifically included; just access them by their module name directly.

If you want to run the REPL application like an executable in Linux, add the following line as the first line in the application:

```
#!/usr/local/bin/node
```

Modify the file to be an executable and run it:

```
$ chmod u+x replcontext.js
$ ./replcontext.js
>>
```

Stuff Happens—Save Often

Node's REPL is a handy interactive tool that can make our development tasks a little easier. REPL allows us not only to try out JavaScript before including it in our files, but also to actually create our applications interactively and then save the results when we're finished.

Another useful REPL feature is that it enables us to create a custom REPL so that we can eliminate the unhelpful `undefined` responses, preload modules, change the prompt or the `eval` routine we use, and more.

I also strongly recommend that you look into using REPL with `rlwrap` in order to persist commands across sessions. This could end up being a major time saver. Plus, who among us doesn't like additional editing capability?

As you explore REPL further, there's one very important thing to keep in mind from this chapter: stuff happens. Save often.

If you'll be spending a lot of time developing in REPL, even with the use of `rlwrap` to persist history, you're going to want to frequently save your work. Working in REPL is no different than working in other editing environments, so I'll repeat: *stuff happens—save often*.

The Necessity of the Console

There are few examples in this book that don't make use of the console. The console is a way of printing out values, checking operations, verifying the asynchronous nature of an application, and providing some kind of feedback.

For the most part, we're using `console.log()` and just printing out messages. But there's more to the console than a server version of the alert box in the browser.

Console Message Types, Console Class, and Blocking

In most of the examples in the book, we're using `console.log()` because we're only interested in feedback while we're experimenting with Node. This function outputs the message to `stdout`, typically the terminal. When you start putting together Node applications for a production environment, though, you're going to want to make use of other console messaging functions.

The `console.info()` function is equivalent to `console.log()`. Both write to `stdout`; both output a newline character as part of the message. The `console.error()` function differs in that it outputs the message (again, with a newline character) to `stderr`, rather than `stdout`:

```
console.error("An error occurred...");
```

The `console.warn()` function does the same.

Both types of messaging appear in the terminal, so you might wonder what the difference is. In actuality, there really is no difference. To understand that, we need to look more closely at the `console` object.

Using a Logging Module

You're not limited to the built-in `console` object for logging. There are more sophisticated tools available, such as the Bunyan and Winston modules.

First of all, the `console` object is a global object instantiated from the `Console` class. We can, if we wish, actually create our own version of the `console` using this same class. And we can do so in two different ways.

To create a new instance of `Console`, we either need to import the `Console` class or access it via the global `console` object. Both of the following result in a new console-like object:

```
// using require
var Console = require('console').Console;

var cons = new Console(process.stdout, process.stderr);
cons.log('testing');

// using existing console object
var cons2 = new console.Console(process.stdout, process.stderr);

cons2.error('test');
```

Notice how in both instances, the `process.stdout` and `process.stderr` properties are passed as writable stream instances for writing log messages and error messages, respectively. The `console` global object is created in just this manner.

I covered `process.stdout` and `process.stderror` in Chapter 2. What we know about both is that they map to the `stdout` and `stderr` file descriptors in the environment, and that they're different from most streams in Node in that they typically block—they're synchronous. The only time they're not synchronous is when the streams are directed at a *pipe*. So, for the most part, the `console` object blocks for

both `console.log()` and `console.error()`. However, this isn't an issue unless you're directing a large amount of data at the stream.

So why use `console.error()` when an error occurs? Because in environments where the two streams are different, you want to ensure the proper behavior. If you're in an environment where log messages don't block but an error does, you want to ensure a Node error does block. In addition, when you run a Node application, you can direct the output for `console.log()` and `console.error()` to different files using command-line redirection. The following directs the `console.log()` messages to a logfile, and the errors to an error file:

```
node app.js 1> app.log 2> error.log
```

The following Node application:

```
// log messages
console.log('this is informative');
console.info('this is more information');

// error messages
console.error('this is an error');
console.warn('but this is only a warning');
```

directs the first two lines to *app.log*, and the second two to *error.log*.

To return to the `Console` class, you can duplicate the functionality of the global `console` object by using the `Console` class and passing in `process.stdout` and `process.stderr`. You can also create a new console-like object that directs the output to different streams, such as log and error files. The `Console` documentation provided by the Node Foundation provides just such an example:

```
var output = fs.createWriteStream('./stdout.log');
var errorOutput = fs.createWriteStream('./stderr.log');
// custom simple logger
var logger = new Console(output, errorOutput);
// use it like console
var count = 5;
logger.log('count: %d', count);
// in stdout.log: count 5
```

The advantage to using this type of object is that you can use the global `console` for general feedback, reserving the newly created object for more formal reporting.

Process and Streams

As noted, the `process` object is covered in Chapter 2, and streams are covered in Chapter 6.

Formatting the Message, with Help from util.format() and util.inspect()

All four `console` functions, `log()`, `warn()`, `error()`, and `info()`, can take any data type, including an object. Non-object values that aren't a string are coerced to a string. If the data type is an object, be aware that Node only prints out two levels of nesting. If you want more, you should use `JSON.stringify()` on the object, which then prints out a more readable indented tree:

```
var test = {
   a : {
       b : {
          c : {
            d : 'test'
          }
       }
   }
}

// only two levels of nesting are printed out
console.log(test);

// three levels of nesting are printed
var str = JSON.stringify(test, null, 3);

console.log(str);
```

The output of the application is:

```
{ a: { b: { c: [Object] } } }
{
   "a": {
      "b": {
         "c": {
            "d": "test"
         }
      }
   }
}
```

If you use a string, you can use printf-like formatting with the string for all four functions:

```
var val = 10.5;
var str = 'a string';

console.log('The value is %d and the string is %s', val, str);
```

This approach is advantageous if you're working with data passed as function arguments or collected from a web request. The type of formatting allowed is based on the

formatting supported for the `util.format()` utilities module, which you could also use directly to create the string:

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

var val = 10.5,
    str = 'a string';

var msg = util.format('The value is %d and the string is %s',val,str);
console.log(msg);
```

Though if you're only using the one function, it's simpler to just use the formatting in `console.log()`. Allowable format values are:

`%s`
: string

`%d`
: number (both integer and float)

`%j`
: JSON. Replaced with `['circular']` if the argument contains circular references

`%%`
: to use a literal percentage sign (%)

Extra arguments are converted to strings and concatenated to the output. If there are too few arguments, the placeholder itself is printed out:

```
var val = 3;

// results in 'val is 3 and str is %s'
console.log('val is %d and str is %s', val);
```

The four functions covered are not the only ones used to provide feedback. There's also `console.dir()`.

The `console.dir()` function differs from the other feedback functions in that whatever object is passed to it is passed, in turn, to `util.inspect()`. This Utilities module function provides more finite control over how an object is displayed via a secondary `options` object. Like `util.format()`, it can also be used directly. An example is:

```
var test = {
   a : {
       b : {
          c : {
            d : 'test'
          }
       }
   }
}
```

```
var str = require('util').inspect(test, {showHidden: true, depth: 4 });
console.log(str);
```

The object is inspected and the result is returned as a string based on what is passed in the `options` object. Options are:

- `showHidden`: to display non-enumerable or symbolic properties (default is `false`)
- `depth`: how many times to recurse to inspect object (default is `2`)
- `colors`: if `true`, output is styled with ANSI color codes (default is `false`)
- `customInspect`: if `false`, then custom inspect functions defined on the objects being inspected won't be called (default is `true`)

The color mapping can be defined globally using the `util.inspect.styles` object. You can modify the global colors, too. Use `console.log()` to print out the object properties:

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

console.log(util.inspect.styles);
console.log(util.inspect.colors);
```

The application in Example 4-2 modifies the object being printed to add a date, a number, and a boolean. In addition, the color value for the boolean is changed from yellow to blue, to differentiate it from the number (by default, they're both yellow). The object is printed out using various methods: after processing with `util.inspect()`, using `console.dir()` with same options, using the basic `console.log()` function, and using the `JSON.stringify()` function on the object.

Example 4-2. Different formatting options for printing out an object

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

var today = new Date();

var test = {
   a : {
       b : {
          c : {
            d : 'test'
          },
          c2 : 3.50
       },
       b2 : true
   },
   a2: today
}

util.inspect.styles.boolean = 'blue';
```

```
// output with util.inspect direct formatting
var str = util.inspect(test, {depth: 4, colors: true });
console.log(str);

// output using console.dir and options
console.dir(test, {depth: 4, colors: true});

// output using basic console.log
console.log(test);

// and JSON stringify
console.log(JSON.stringify(test, null, 4));
```

The result is shown in Figure 4-1 so you can see the color effects. I use a white background for my terminal window, with black text.

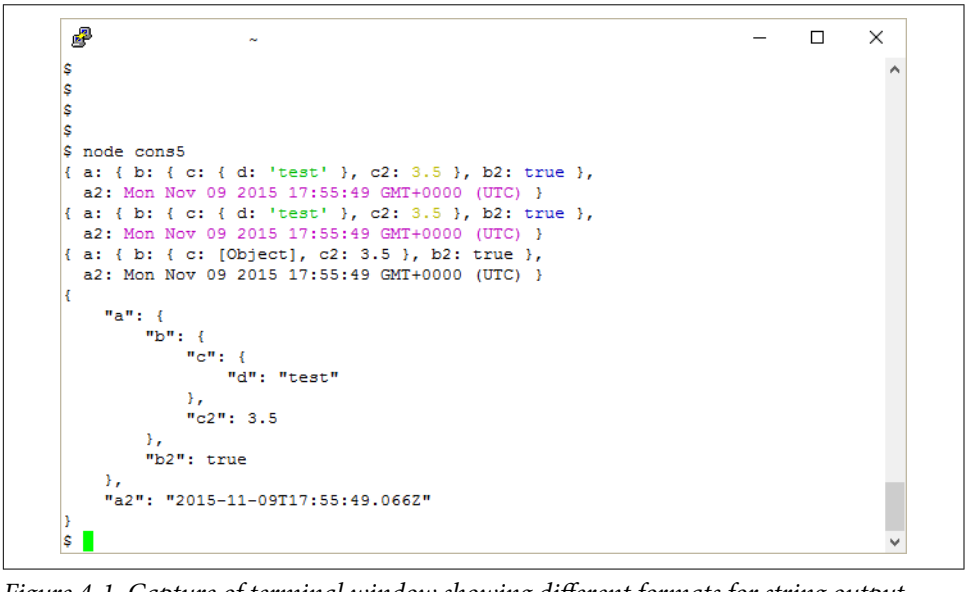


Figure 4-1. Capture of terminal window showing different formats for string output

The `console.dir()` function supports three of the four `util.inspect()` options: `showHidden`, `depth`, and `colors`. It doesn't support `customInspect`. If set to `true`, this option indicates that the object is actually supplying its own inspection function.

Providing Richer Feedback with console and a Timer

Returning to the `console` object, one additional approach that can provide a more in-depth picture of what's happening with an application is to add a timer and output a begin and end time.

The two console functions we're using for this functionality are `console.time()` and `console.timeEnd()`, passing a timer name to both.

In the following code snippet, the code is using a longer-lasting loop so that enough time passes for the timer functions to register a time difference.

```
console.time('the-loop');

for (var i = 0; i < 10000; i++) {
   ;
}

console.timeEnd('the-loop');
```

Even with a largish loop, the time involved will barely register. How long depends on the load in the machine, as well as the process. But the timer functionality isn't limited to synchronous events. Being able to use a specific name with the timer means we can use the functionality with asynchronous events.

Here, I modify the Hello World application from Chapter 1, adding in a start to the timer at the beginning of the application, ending it for each web request, and then restarting it. It times the gaps in time between each request. Sure, we could use a `Date()` function to do a more representative timer, but what's the fun of that?

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

console.time('hello-timer');
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
  console.timeEnd('hello-timer');
  console.time('hello-timer');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

In all seriousness, what the code demonstrates is that you can incorporate the timer into asynchronous operations, thanks to the ability to individually name the timers.

CHAPTER 5

Node and the Web

Node isn't yet at the point where it can replace the fairly ubiquitous Apache/PHP killer combination, but it is gaining in popularity, especially considering the ease in creating cross-platform applications and the support it has among the big tech companies.

In this chapter, we'll explore Node's webby roots, including a more in-depth look at the HTTP module, as well as try our hand at creating a very simple static web server. We'll also look at core Node modules that simplify the web development experience.

The HTTP Module: Server and Client

Don't expect to use the HTTP module to re-create a web server with the sophistication of Apache or Nginx. As the Node documentation notes, it's low-level, focusing on stream handling and message parsing. Still, it provides the foundational support for more sophisticated functionality, such as Express, covered in Chapter 10.

The HTTP module supports several objects, including `http.Server`, which is what's returned when you use the `http.createServer()` function demonstrated in Chapter 1. In that example, we embedded a callback function to handle the web request, but we can also use separate events, since `http.Server` inherits from `EventEmitter`.

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

var server = http.createServer().listen(8124);

server.on('request', function(request,response) {

   response.writeHead(200, {'Content-Type': 'text/plain'});
   response.end('Hello World\n');
});

console.log('server listening on 8214');
```

We can also listen for other events, such as when a connection is made (`connect`), or the client requests an upgrade (`upgrade`). The latter occurs when the client asks for an upgrade to the version of HTTP or to a different protocol.

HTTP Innards

For those who want to know a little more about the innards of HTTP, the `HTTP.Server` class is actually an implementation of the TCP-based `Net.Server`, which is covered in Chapter 7. TCP provides the transportation layer, while HTTP provides the application layer.

The callback function used when responding to a web request supports two parameters: request and response. The second parameter, the `response`, is an object of type `http.ServerReponse`. It's a *writable stream* with support for several functions, including `response.writeHead()` to create the response header, `response.write()` to write the response data, and `response.end()` to end the response.

Readable and Writeable Streams

Chapter 6 covers both readable and writeable streams.

The first parameter, the `request`, is an instance of the `IncomingMessage` class, which is a *readable stream*. Among some of the information you can access from the request is the following:

`request.headers`
: The request/response header objects

`request.httpVersion`
: HTTP request version

`request.method`
: Only for a `http.Server` request, and returns HTTP verb (GET or POST)

`request.rawHeaders`
: Raw headers

`request.rawTrailers`
: Raw trailers

To see the difference between `request.headers` and `request.rawHeaders`, print them out using `console.log()` within the request. Note that values are given as properties with `request.headers`, but as array elements for `request.rawHeaders`, with the property in the first array element and the value in the second, in case you want to access individual values:

```
console.log(request.headers);
console.log(request.rawHeaders);

// pulling host
console.log(request.headers.host);
console.log(request.rawHeaders[0] +  ' is ' + request.rawHeaders[1]);
```

In the Node documentation, note that some of the `IncomingMessage` properties (`statusCode` and `statusMessage`) are only accessible for a *response* (not a request) obtained from an `HTTP.ClientRequest` object. In addition to creating a server that listens for requests, you can also create a client that *makes* requests. You do so with the `ClientRequest` class, which you instantiate using the `http.request()` function.

To demonstrate both types of functionality, server and client, I'm going to take example code for creating a client from the Node documentation and modify it to access a server that's local to the client. In the client, I'm creating a POST method to send the data, which means I need to modify my server to read this data. This is where the readable stream part of `IncomingMessage` comes in. Instead of listening for a `request` event, the application listens for one or more `data` events, which it uses to get *chunks* (yes, that's actually a technical term) of data in the request. The application continues getting chunks until it receives an `end` event on the request object. It then uses another helpful Node module, Query String (covered in more detail in "Parsing the Query with Query String" on page 125), to parse the data, and print it out to the console. Only then does it send back the response.

The code for the modified server is shown in Example 5-1. Note it's very similar to what you've tried previously, except for the new event handling for POSTed data.

Example 5-1. Server that listens for a POST and processes posted data

```
var http = require('http');
var querystring = require('querystring');

var server = http.createServer().listen(8124);

server.on('request', function(request,response) {

   if (request.method == 'POST') {
        var body = '';

        // append data chunk to body
        request.on('data', function (data) {
            body += data;

        });

        // data transmitted
        request.on('end', function () {
            var post = querystring.parse(body);
            console.log(post);
            response.writeHead(200, {'Content-Type': 'text/plain'});
            response.end('Hello World\n');
        });
    }
});
console.log('server listening on 8214');
```

The code for the new client is shown in Example 5-2. Again, it uses `http.ClientRequest`, which is an implementation of a writable stream, as apparent from the `req.write()` method used in the example.

The code is almost a direct copy of the code in the Node documentation example except for the server we're accessing. Since both the server and client are on the same machine, we're using `localhost` as the host. In addition, we're not specifying the `path` property in the options, because we're accepting the default of `/`. The headers for the request are set to a content type of `application/x-www-form-urlencoded`, used with POSTed data. Also note that the client receives data back from the server via the `response`, which is the only argument in the `http.request()` function's callback function. The POSTed data is again pulled as chunks from the response, but this time, each chunk is printed out to the console as soon as it is received. Since the returned message is so short, only one `data` event is triggered.

The actual POSTed request isn't handled asynchronously because we're initiating an action, not blocking while waiting on an action.

Example 5-2. HTTP client POSTing data to a server

```
var http = require('http');
var querystring = require('querystring');

var postData = querystring.stringify({
  'msg' : 'Hello World!'
});

var options = {
  hostname: 'localhost',
  port: 8124,
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': postData.length
  }
};

var req = http.request(options, function(res) {
  console.log('STATUS: ' + res.statusCode);
  console.log('HEADERS: ' + JSON.stringify(res.headers));
  res.setEncoding('utf8');

  // get data as chunks
  res.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });

  // end response
  res.on('end', function() {
    console.log('No more data in response.')
  })
});

req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});

// write data to request body
req.write(postData);
req.end();
```

Pop open a second terminal, start the server in the first, and then run the client in the second. The client basically says hello to the server, which responds by saying hello back.

All of this, just to have two processes say hello to each other. But you've also managed to implement two-way client/server communication, as well as have a chance to work with POSTed data, instead of just GETs. And you were introduced to all but one of

the HTTP module's classes: `http.ClientRequest`, `http.Server`, `http.IncomingMessage`, and `http.ServerResponse`.

The only class we haven't looked at is `http.Agent`, which is used for pooling sockets. Node maintains a pool of connection sockets for handling requests that were made using `http.request()` or `http.get()`. The latter request is a simplified request, for a GET request that has no body. If the application is making a lot of requests, they can get bottlenecked because of the limited pool. The way around this is to disable connection pooling by setting the `agent` to `false` in the outgoing request properties. In Example 5-2, this would require changing the options to the following (the change is bolded):

```
var options = {
  hostname: 'localhost',
  port: 8124,
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': postData.length
  },
  agent: false
};
```

You can also change the maximum socket pool, using `agent.maxFreeSockets`. By default, it's set to `256`. Be aware, though, that changing the outgoing request connection pool could have adverse effects on memory and other resource use.

We'll get a chance to work with even more sophisticated communications in Chapter 7, but for now, let's return to what it means to actually create an HTTP server that returns something more than "hi there."

What's Involved in Creating a Static Web Server

We have all the functionality we need to build a simple router or to serve static files built directly into Node. But *being able* to do so and doing so *easily* are two different things.

When thinking of what's necessary to build a simple but functional static file server, we might come up with the following set of steps:

1. Create an HTTP server and listen for requests.
2. When a request arrives, parse the request URL to determine the location for the file.
3. Check to make sure the file exists.
4. If the file doesn't exist, respond accordingly.

5. If the file does exist, open the file for reading.
6. Prepare a response header.
7. Write the file to the response.
8. Wait for the next request.

We'll only need core modules to perform all of these functions (with one exception, as we'll see later in this section). Creating an HTTP server and reading files requires the HTTP and File System modules. In addition, we'll want to define a global variable for the base directory, or use the predefined `__dirname` (more on this in the upcoming sidebar "Why Not Use __dirname?" on page 123).

The top of the application has the following code at this point:

```
var http = require('http'),
    fs = require('fs'),
    base = '/home/examples/public_html';
```

We created a web server in Chapter 1's Hello World example, and we'll build on that for this chapter's work. The `http.createServer()` function creates the server, with a callback with two values: the web request and the response we'll create. The application can get the document requested by directly accessing the HTTP request object's `url` property. To double-check the response compared to requests, we'll also throw in a `console.log` of the requested file's path name. This is in addition to the `console.log` message that's written when the server is first started:

```
var http = require('http'),
    fs = require('fs'),
    base = '/home/examples/public_html';

http.createServer(function (req, res) {

   pathname = base + req.url;
   console.log(pathname);

}).listen(8124);

console.log('Server web running at 8124');
```

Running the application starts up a web server that listens for web requests on the 8124 port.

About the Ports We Use

When you access a website using a browser, you don't usually have to provide a port number. The reason is that there are two *well-known ports*—80 for HTTP requests, and 443 for HTTPS—that browsers incorporate automatically into the request. Wikipedia maintains a list of all well-known ports.

We're listening on the 8214 port, because the default web port of 80 is typically being managed by whatever is the primary web server in your development environment—whether it's Ngnix, Apache, lighttpd, the newer Caddy, or a production Node application.

We could use the well-known port of 80, but ports under 1024 require root (administration) permission. Not even Apache provides server instances that run as root. Apache itself runs as root, but spawns child threads that run with more restricted permission levels. You really don't want to run web servers with root permissions.

You can tweak server internals, such as iptables, to redirect port 1024 access to port 80, but this isn't an approach I would take. A better approach is to set up your web service (Apache, Ngnix, lighttpd, or Caddy), and have it proxy requests to that web server. I'll cover how to use Apache as proxy for Node applications in more detail in "Using Apache to Proxy a Node Application" on page 124.

Testing the application with an *index.html* file (for instance, *http://blipdebit.com:8124/index.html*) will result in something like the following printout (depending on your particular environment):

```
/home/examples/public_html/index.html
```

Of course, the browser is left hanging, since we didn't prepare a response, but we'll get to that next.

We could test to see if a requested file is available before opening and reading it. The `fs.stat()` function returns an error if the file doesn't exist.

File Removed After Check

One hazard with using `fs.stat()` is that something can cause the file to go away between the status check and the actual file opening. Another approach is to just open the file directly, which will return an error if the file is missing. The issue, though, with using something like `fs.open()` directly and then passing a file descriptor to `fs.createReadStream()` is that they don't provide information about whether the file is a directory or file (in addition to whether it's missing or locked). So, I use `fs.stat()` but also check for an error when I open the read stream, which allows me more control of the error handling.

Speaking of reading from the file, we might try to use `fs.readFile()` to read the file contents. The problem with `fs.readFile()`, though, is that it wants to read the file completely into memory before making it available. Documents served over the Web can be quite large. In addition, there can be many requests for a document at any given time. Something like `fs.readFile()` just won't scale.

Instead of using `fs.readFile()`, the application creates a read stream via the `fs.createReadStream()` method, using the default settings. Then it's a simple matter to just *pipe* the file contents directly to the HTTP response object. Since the stream sends an end signal when it's finished, we don't need to use the `end` method call with the read stream:

```
res.setHeader('Content-Type', 'test/html');

// create and pipe readable stream
var file = fs.createReadStream(pathname);
file.on("open", function() {
   // 200 status - found, no errors
   res.statusCode = 200;
   file.pipe(res);
});
file.on("error", function(err) {
  res.writeHead(403);
  res.write('file missing, or permission problem');
  console.log(err);
});
```

The read stream has two events of interest: `open` and `error`. The `open` event is emitted when the stream is ready, and the `error` if a problem occurs. What kind of error? The file could have vanished between the status check, the permissions could prevent access, or the file could be a subdirectory. Since we don't know what happened at this point, we're just giving a general `403` error, which covers most of the potential problems. We're writing a message, which may or may not display, depending on how the browser responds to the error.

The application calls the `pipe` method in the callback function for the `open` event.

At this point, the static file server looks like the application in Example 5-3.

Example 5-3. A simple static file web server

```
var http = require('http'),
    fs   = require('fs'),
    base = '/home/examples/public_html';

http.createServer(function (req, res) {

   pathname = base + req.url;
   console.log(pathname);

   fs.stat(pathname, function(err,stats) {
      if (err) {
        console.log(err);
        res.writeHead(404);
        res.write('Resource missing 404\n');
        res.end();
      } else {
         res.setHeader('Content-Type', 'text/html');

         // create and pipe readable stream
         var file = fs.createReadStream(pathname);

         file.on("open", function() {
            res.statusCode = 200;
            file.pipe(res);
         });

         file.on("error", function(err) {
           console.log(err);
           res.writeHead(403);
           res.write('file missing or permission problem');
           res.end();
         });

       }
    });
}).listen(8124);

console.log('Server running at 8124');
```

I tested it with a simple HTML file, which has nothing more than an `img` element, and the file loaded and displayed properly:

```
    <!DOCTYPE html>
    <head>
       <title>Test</title>
       <meta charset="utf-8" />
```

```
</head>
<body>
<img src="./phoenix5a.png" />
</body>
```

I also tried it with a file that doesn't exist. The error is caught when I used `fs.stat()`, returning a `404` message to the browser, and printing an error about no such file or directory found to the console.

I next tried it with a file where all read permissions had been removed. This time, the read stream caught the error, sending a message about not being able to read the file to the browser and an error about permissions to the console. I also returned a more appropriate HTTP status of `403` for the permission-denied file.

Lastly, I tried the server application with another example file I had, which contained an HTML5 `video` element:

```
<!DOCTYPE html>
<head>
   <title>Video</title>
   <meta charset="utf-8" />
</head>
<body>
   <video id="meadow" controls>
      <source src="videofile.mp4" />
      <source src="videofile.ogv" />
      <source src="videofile.webm" />
   </video>
</body>
```

Though the file would open and the video would display when I tried the page with Chrome, the `video` element did not work when I tested the page with Internet Explorer 10. Looking at the console output provided the reason:

```
Server running at 8124/
/home/examples/public_html/html5media/chapter1/example2.html
/home/examples/public_html/html5media/chapter1/videofile.mp4
/home/examples/public_html/html5media/chapter1/videofile.ogv
/home/examples/public_html/html5media/chapter1/videofile.webm
```

Though IE10 is capable of playing the MP4 video, it tests all three of the videos because the content type of the response header is `text/html` for each. Though other browsers will ignore the incorrect content type and display the media correctly, IE 10 does not—appropriately, in my opinion, because I may not have quickly found the error in the application otherwise.

Microsoft has since adapted the handling of the content type in its newer Edge browser, and does display the appropriate video. Still, we want this application to do the right thing. It has to be modified to test for the file extension for each file and

then return the appropriate MIME type in the response header. We could code this functionality ourselves, but I'd rather make use of an existing module: Mime.

You can install Mime using npm: `npm install mime`. The GitHub site has it.

The Mime module can return the proper MIME type given a filename (with or without path), and can also return file extensions given a content type. The Mime module is included:

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

The returned content type is used in the response header, and also output to the console, so we can check the value as we test the application:

```
// content type
var type = mime.lookup(pathname);
console.log(type);
res.setHeader('Content-Type', type);
```

Now when we access the file with the video element in IE10, the video file works. For all browsers, the proper content-type setting is returned.

What doesn't work, though, is when we access a directory instead of a file. When this happens, an error is output to the console:

```
{ [Error: EISDIR, illegal operation on a directory] errno: 28, code: 'EISDIR' }
```

What happens with the browsers, though, varies. Edge displays a general 403 error, as shown in Figure 5-1, while Firefox and Chrome each display my generic "something is wrong" message. But there is a difference between not being able to access a subdirectory and something being wrong with the file.

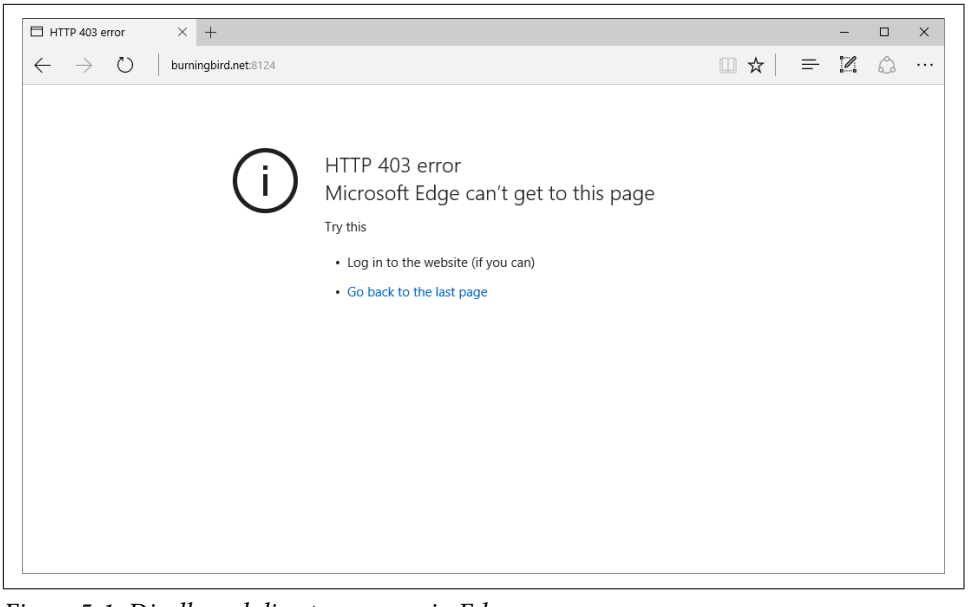


Figure 5-1. Disallowed directory access in Edge

We not only need to check if the resource being accessed exists, but we also need to check whether it's a file or a directory. If it's a directory being accessed, we can either display its contents or we can output an error—it's the developer's choice.

There's still one more change. The base path I'm using works great on my Linux server, but it's going to have issues if I try it out on my Windows machine. For one, I don't have a */home/examples/public_html* directory on my Windows machine. For another, my Windows machine supports the backslash (\), not the forward slash.

To ensure the application works in both environments, I first create the directory structure in Windows. Secondly, I use the core Path module to *normalize* the path string so it works in both environments. The `path.normalize()` function takes a string and returns either the same string, if it's already normalized for the environment, or a transformed string.

```
var pathname = path.normalize(base + req.url);
```

Now the application works in both of my machines.

More on the Path module in Chapter 6.

The final version of a minimal static file server, in Example 5-4, uses `fs.stats` to check for the existence of the requested object and whether it's a file. If the resource doesn't exist, an HTTP status of `404` is returned. If the resource exists but it's a directory, an HTTP error status code of `403`—forbidden—is returned. The same occurs if the file is locked, but the message changes. In all cases, the request response is more appropriate.

Example 5-4. Final version of minimal static file server

```
var http = require('http'),
    url =  require('url'),
    fs   = require('fs'),
    mime = require('mime'),
    path = require('path');

var base = '/home/examples/public_html';

http.createServer(function (req, res) {

   pathname = path.normalize(base + req.url);
   console.log(pathname);

   fs.stat(pathname, function(err, stats) {
      if (err) {
        res.writeHead(404);
        res.write('Resource missing 404\n');
        res.end();
      } else if (stats.isFile()) {
         // content type
         var type = mime.lookup(pathname);
         console.log(type);
         res.setHeader('Content-Type', type);

         // create and pipe readable stream
         var file = fs.createReadStream(pathname);
         file.on("open", function() {
            // 200 status - found, no errors
            res.statusCode = 200;
            file.pipe(res);
         });
         file.on("error", function(err) {
           console.log(err);
           res.statusCode = 403;
           res.write('file permission');
           res.end();
         });
       } else {
        res.writeHead(403);
        res.write('Directory access is forbidden');
        res.end();
```

```
        }
    });
}).listen(8124);
console.log('Server running at 8124');
```

The following is the console output for accessing a web page that contains both image and video file links using Firefox:

```
/home/examples/public_html/video.html
text/html
/home/examples/public_html/phoenix5a.png
image/png
/home/examples/public_html/videofile.mp4
video/mp4
/home/examples/public_html/favicon.ico
image/x-icon
/home/examples/public_html/favicon.ico
image/x-icon
```

Note the proper handling of the content types—not to mention the automatic *favicon* access both Firefox and Chrome attach to page requests.

You get a better feel for how the read stream works when you load a page that has a video element and begin to play it. The browser grabs the read stream output at a speed it can manage, filling its own internal buffer, and then pauses the output. If you close the server while the video content is playing, the video continues to play up to the point where it exhausts its current video buffer. The video element then goes blank because the read stream is no longer available. It's actually a little fascinating to see how well everything works with so little effort on our part.

Why Not Use __dirname?

In some of the examples in this book, I hardcode the base location for web documents, typically as */home/examples/public_html*. You might be wondering why I'm not using `__dirname`.

In Node, you can use the predefined `__dirname` as a way of specifying the current working directory for a Node application. However, in this chapter's examples, I'm accessing files located separately from my Node application, which is why I'm not using it. Except in this circumstance, you should use `__dirname`. It provides a way to test applications and then move them to production without having to change the value of a base location variable.

Use `__dirname` in the following manner:

```
var pathname = __dirname + req.url;
```

Note the double underscore in `__dirname`.

Though the application works when tested with several different documents, it's not perfect. It doesn't handle many other types of web requests, it doesn't handle security or caching, and it doesn't properly handle the video requests. One web page application I tested that uses HTML video also makes use of the HTML5 video element API to output the state of the video load process. This application didn't get the information it needed to work.

There are many little gotchas that can trip us when it comes to creating a static file server, which is why people use more complex systems, such as Express, which I introduce in Chapter 9.

Using Apache to Proxy a Node Application

I don't think anyone sees Node replacing the more established web servers, such as Apache, in the near future. But having Node handle some functionality while the server handles the rest is a viable option. Since Apache is still the most popular server, I'm focusing on it in this section.

The thing is, how do we run a Node server at the same time we're running something like Apache, and without forcing our users to specify a port? Only one web server can answer requests on port 80, the default port. And it's true that if we provide an API for web browsers, the port number can be somewhat hidden, but what if we want to use both Apache and Node at the same time without port conflict?

Other Production Issues

Keeping Node running forever and recovering from a crash are other issues, and are covered in Chapter 11.

The simplest approach to running a Node application with Apache is to have Apache *proxy* the Node service's requests. This means all requests to the Node application go through Apache first.

This is both a good and not-so-good solution. The good is it's extremely simple, and we have a very robust, well-known web server fielding the requests before our Node application gets hit. Apache provides security and other functionality that would be extremely difficult to implement in a Node application. The not-so-good is that Apache spawns a new thread for every request, and there are only so many threads to go around.

Still, most sites run on Apache and manage to get by without slowing to a crawl. Unless you're about to become a major player, this is a good option.

To have Apache proxy for Node, you first of all have to enable Apache proxying. Issue the following commands at the command line:

```
a2enmod proxy
a2enmod proxy_http
```

Then, in the subdomain, add the proxying. For instance, when I used my server, I set up shelleystoybox.com as a host for my Node application:

```
<VirtualHost ipaddress:80>
    ServerAdmin shelleyp@burningbird.net
    ServerName shelleystoybox.com

    ErrorLog path-to-logs/error.log
    CustomLog path-to-logs/access.log combined

    ProxyRequests off

    <Location />
            ProxyPass http://ipaddress:2368/
            ProxyPassReverse http://ipaddress:2368/
    </Location>
</VirtualHost>
```

Change the subdomain, administrator email, port, and IP address to match your environment. Then it's a matter of loading the new subdomain:

```
a2ensite shelleystoybox.com
service apache2 reload
```

again, changing the subdomain to match your environment.

Accessing Site via Port

Using the proxy doesn't prevent people from accessing the site explicitly using the port. You can prevent this, but it takes advanced server tinkering. For instance, in my Ubuntu server, I could modify the iptables:

```
iptables -A input -i eth0 -p tcp --dport 2368 -j DROP
```

But doing so does require server administrative skills.

Parsing the Query with Query String

We had a chance to use the Query String module earlier in the chapter. Its sole purpose is to prepare and process query strings.

When you receive a query string, you can convert it to an object using `querystring.parse()`, as demonstrated in Example 5-1. The default query string separator (`'&'`) can be overridden in an optional second function parameter, and the default assignment (`'='`) can be overridden in the third parameter. The fourth optional parameter contains a `decodeURIComponent`, which is `querystring.unescape()` by default. Change it if the query string won't be UTF-8. However, it's likely

your query strings will use the standard separator, and assignment, and be UTF-8, so you can accept the defaults.

To see how the `querystring.parse()` function works, consider the following query:

```
somedomain.com/?value1=valueone&value1=valueoneb&value2=valuetwo
```

The `querystring.parse()` function would return the following object:

```
{
   value1: ['valueone','valueoneb'],
   value2: 'valuetwo'
}
```

When preparing a query string to send, as demonstrated in Example 5-2, use `querystring.stringify()`, passing in the object to encode. So, if you have an object like the one we just parsed, pass that to `querystring.stringify()`, and you'll get the properly formatted query string to transmit. In Example 5-2, `querystring.stringify()` returned:

```
msg=Hello%20World!
```

Notice how the space is escaped. The `querystring.stringify()` function takes the same optional parameters, with the exception of the last option's object. In this case, you'd provide a custom `encodeURIComponent`, which is `stringify.escape()` by default.

DNS Resolution

It's unlikely most of your applications are going to need any direct DNS services, but if they do, this type of functionality is provided in the core DNS module.

We'll look at two of the DNS module functions: `dns.lookup()` and `dns.resolve()`. The `dns.lookup()` function can be used to give the first returned IP address a domain name. In the following code, the example returns the first found IP address for oreilly.com:

```
dns.lookup('oreilly.com', function(err,address,family) {
     if (err) return console.log(err);

     console.log(address);
     console.log(family);
});
```

The address is the returned IP address, and the family value is either 4 or 6, depending on whether the address is IPv4 or IPv6. You can also specify an `options` object:

`family`
: A number, 4 or 6, representing the type of address you want (IPv4 or IPv6)

`hints`
: Supported `getaddrinfo` flags, a number

`all`
: If `true`, returns all addresses (default is `false`)

I was curious about all the IP addresses, so I modified the code to get all addresses. When I did so, though, I eliminated the `family` parameter, which is `undefined` when accessing all of the addresses. I get back an array of applicable IP address objects, with IP address and family:

```
dns.lookup('oreilly.com', {all: true}, function(err,family) {
     if (err) return console.log(err);

     console.log(family);
});
```

returns:

```
[ { address: '209.204.146.71', family: 4 },
  { address: '209.204.146.70', family: 4 } ]
```

The `dns.resolve()` function resolves a hostname into record types. The types (as strings) are:

`A`
: Default IPv4 address

`AAAA`
: IPv6 address

`MX`
: Mail exchange record

`TXT`
: Text records

`SRV`
: SRV records

`PTR`
: Used for reverse IP lookup

`NS`
: Name server

`CNAME`
: Canonical name records

`SOA`
: Start of authority record

In the following, I'm using `dns.resolve()` to get all of the MX records for oreilly.com:

```
dns.resolve('oreilly.com','MX', function(err,addresses) {
   if (err) return err;
   console.log(addresses);
});
```

which then returns:

```
[ { exchange: 'aspmx.l.google.com', priority: 1 },
  { exchange: 'alt1.aspmx.l.google.com', priority: 5 },
  { exchange: 'aspmx2.googlemail.com', priority: 10 },
  { exchange: 'alt2.aspmx.l.google.com', priority: 5 },
  { exchange: 'aspmx3.googlemail.com', priority: 10 } ]
```

CHAPTER 6

Node and the Local System

Node's File System module is used throughout the book. There are few resources more essential to applications than those from the filesystem. The only other resources more widely used are the network resources, which are covered in Chapters 5 and 7.

What's terrific about Node is that the File System works, for the most part, the same way across different operating systems. Node also works hard to ensure other functionality built on the technology is operating-system-agnostic. Sometimes it succeeds, and sometimes we need a little help from third-party modules.

This chapter provides a more formal introduction to the File System. In addition, it looks more closely at OS-specific functionality and differences. Lastly, we look at two modules, ReadLine and ZLib, which provide interactive command-line communication, as well as compression capabilities, respectively.

Exploring the Operating System

Some technologies manage to hide every last bit of operating system differences, while others require significant work to manage OS-specific quirks. Node falls somewhere in between. For the most part, you can create an application and it runs everywhere. But there are certain functionalities where OS differences intrude. As I mentioned at the beginning of the chapter, sometimes Node handles them well and sometimes you need a helpful third-party module.

Accessing information directly about the operating system comes to us via the OS core module. It's one of the useful tools that helps us create cross-platform-capable applications. It also helpfully provides information about the current environment's resource usage and capability.

Access to the OS module begins with `require`:

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

The OS module's functionality is informative only. For instance, if you want to ensure cross-platform capability, you can discover the end-of-line character supported in the system, discover if the system is big-endian (BE) or little-endian (LE), and get direct access to the temporary folder and home directories:

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

console.log('Using end of line' + os.EOL + 'to insert a new line');
console.log(os.endianness());
console.log(os.tmpdir());
console.log(os.homedir());
```

Both my Ubuntu server and my Windows 10 machine are LE, and the EOL character works as expected in both (the second part of the line begins on a new line). The only difference is the temporary folder directory and home directories, naturally.

The Temporary Folder

The temporary folder is where files are temporarily stored. The contents are deleted when the system is restarted, or at other intervals.

The OS module also provides a way to check out the resources available for the current machine.

```
console.log(os.freemem());
console.log(os.loadavg());
console.log(os.totalmem());
```

The `os.loadavg()` function is specific to Unix; it just returns zeros for Windows. It's the 1-, 5-, and 15-minute load averages, which reflect system activity. To get a percentage, multiply the three numbers by 100. The memory functions `os.freemen()` and `os.totalmem()` return memory in bytes.

Another function, `os.cpus()`, returns information about the machine's CPUs. It returns the number of milliseconds the CPU spent in `user`, `nice`, `sys`, `idle`, and `irq`. If you're not familiar with the concepts, the `user` value is the amount of time the CPU spent running user space processes, `idle` is how often the CPU was idle, and `sys` is how much time the CPU spent on system processes (kernel). The `nice` value reflects how much of the user space processes were `niced`: priority adjusted so the process doesn't run as frequently. The `irq` is an interrupt request for service at the hardware level.

The CPU times aren't as helpful as knowing percentages. We can determine these by adding all the values and then finding out each percentage. We can also make use of third-party modules that provide the percentages, as well as other information. Microsoft provides a nice write-up on how this can work in Azure, but the information and modules listed should work in all environments.

Streams and Pipes

Streaming technology appears all throughout the Node core, providing functionality for HTTP, as well as other forms of networking. It also provides File System functionality, which is why I'm covering it before we dig more deeply into the File System module.

The Stream object is an abstract interface, which means you're not going to be directly creating streams. Instead, you'll be working with various objects that implement Stream, such as an HTTP request, File System read or write stream, ZLib compression, or `process.stdout`. The only time you'll actually implement the Stream API is if you're creating a custom stream. Since that's beyond an introductory text, I'll leave that for a follow-up exercise. For now, we'll focus on how streams exposed in other functionality generally behave.

Because so many objects in Node implement the stream interface, there is basic functionality available in all streams in Node:

- You can change the encoding for the stream data with `setEncoding`.
- You can check whether the stream is readable, writable, or both.
- You can capture stream events, such as *data received* or *connection closed*, attaching callback functions for each.
- You can pause and resume the stream.
- You can pipe data from a readable stream to a writable stream.

Note the item about streams being readable, writable, or both. The latter stream type is known as a *duplex* stream. There's also a variation of the duplex stream, known as a `transform`, for when the input and output are causally related. We'll look at this type of stream when I cover ZLib compression, later in the chapter.

A readable stream starts out in a paused mode, which means no data is sent until some form of explicit read (`stream.read()`) or command for the stream to resume (`stream.resume()`) occurs. However, the stream implementations we use, such as the File System's readable stream, are switched to flowing mode as soon as we code for a data event (the way to get access to the data in a readable stream). In flowing mode, the data is accessed and sent to the application as soon as it's available.

Readable streams support several events, but in most of our readable stream usage, we're interested in three events: data, end, and error. The data event is sent when there's a new chunk of data ready to be utilized, and the end event is sent when all of the data has been consumed. The error event is sent when an error occurs, of course. We saw a readable stream in action with Example 5-1 in Chapter 5, and we'll see it in action later with File System.

A writable stream is a destination data that is being sent (written) to. Among the events we'll listen for is `error`, and also `finish`, when an `end()` method is called and all the data has been flushed. Another possible event is `drain`, when an attempt to write data returns `false`. We used a writable stream when we created an HTTP client in Example 5-2 in Chapter 5, and we'll also see it in action with the File System module covered in "A Formal Introduction to the File System" on page 133.

A duplex stream has elements of both the readable and writable stream. A transform stream is a duplex stream except that—unlike duplex streams, where internal input and output buffers exist independently from one another—a transform stream directly connects the two, with an intermediate step transforming the data. Under the hood, transform streams must implement a function, `_transform()`, that takes incoming data, does something with it, and pushes it to the output.

To better understand a transform stream, we need to take a closer look at functionality supported in all streams: the `pipe()` function. We saw it used in Example 5-1, when a readable stream directly piped the contents of a file to the HTTP response object:

```
// create and pipe readable stream
        var file = fs.createReadStream(pathname);
        file.on("open", function() {
           file.pipe(res);
        });
```

What the pipe does is take the data out of the file (the stream) and outputs it to the `http.ServerResponse` object. In the Node documentation, we learn that this object implements the writable stream interface, and we'll see later that `fs.createReadStream()` returns a `fs.ReadStream`, which is an implementation of a readable stream. One of the methods a readable stream supports is `pipe()` to a writable stream.

Later, we'll look at an example of using the Zlib module to compress a file, but for now, a quick peek. It's an excellent demonstration of a transform stream.

```
var gzip = zlib.createGzip();
var fs = require('fs');
var inp = fs.createReadStream('input.txt');
var out = fs.createWriteStream('input.txt.gz');

inp.pipe(gzip).pipe(out);
```

The input is a readable stream, and the output a writable stream. The content of one is piped to the other, but first it goes through a compression routine. That's the transform.

A Formal Introduction to the File System

Node's File System module (fs) provides all the functionality you need to work with a machine's filesystem, regardless of operating system. We've seen it in use all throughout the book. Now we'll take a look more formally at the functionality.

First, as the Node documentation states, the File System is a set of wrappers that works with POSIX functions. What this means is that it supports POSIX-standardized (cross-platform-compatible) filesystem access functionality that works out of the box in all supported operating systems. So your application can work in OS X, Linux, and Windows, as well as the newer environments such as Android, and microcomputers like Raspberry Pi.

The File System module provides synchronized versions of functions, as well as the Node-traditional asynchronous versions. Whether this is a good idea or not is moot, as they exist and we're free to use them or not.

The asynchronous functions take an error-first callback as last argument, while the synchronous functions immediately throw an error if one occurs. You can use the traditional `try...catch` with synchronous File System functions, while accessing the error object with the asynchronous versions. In the rest of this section, I'm focusing solely on the asynchronous functions. Just be aware, though, that synchronous versions of the functions do exist.

In addition to numerous functions, the File System supports four classes:

`fs.FSWatcher`
: Supports events for watching for file changes

`fs.ReadStream`
: A readable stream

`fs.WriteStream`
: A writable stream

`fs.Stats`
: Information returned from the `*stat` functions

The fs.Stats Class

The `fs.Stats` object is returned if you use `fs.stat()`, `fs.lstat()`, and `fs.fstat()`. It can be used to check whether a file (or directory) exists, but it also returns information such as whether a filesystem object is a file or a directory, a UNIX domain socket, permissions associated with the file, when an object was last accessed or modified, and so on. Node provides some functions for accessing information, such as `fs.isFile()` and `fs.isDirectory()` to determine if the object is a file or directory. You can also access the stats data directly, using the following:

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

fs.stat('./phoenix5a.png', function(err,stats) {
   if (err) return console.log(err);
   console.log(util.inspect(stats));
});
```

You'll get back a data structure similar to the following in Linux:

```
{ dev: 2048,
  mode: 33204,
  nlink: 1,
  uid: 1000,
  gid: 1000,
  rdev: 0,
  blksize: 4096,
  ino: 1419012,
  size: 219840,
  blocks: 432,
  atime: Thu Oct 22 2015 19:46:41 GMT+0000 (UTC),
  mtime: Thu Oct 22 2015 19:46:41 GMT+0000 (UTC),
  ctime: Mon Oct 26 2015 13:38:03 GMT+0000 (UTC),
  birthtime: Mon Oct 26 2015 13:38:03 GMT+0000 (UTC) }
```

This is basically a dump of the POSIX `stat()` function, which returns the status of a file. The dates are pretty intuitive, but the other values may be confusing. The `size` is in bytes, the `blksize` is the size of the block in the operating system, and `blocks` is the number of object blocks. Both of the latter values are `undefined` in Windows.

One of the more interesting components is the `mode`. This value contains the object's permissions. The problem is it's not entirely readable.

This is where I call for a helper function, or in Node's case, a helper module. The stat-mode module is a purposed module that takes the stat object returned from a function such as `fs.stat()` and allows us to directly query values. Example 6-1 demonstrates how to use it to extract useful file permission information.

Example 6-1. Getting file permissions using stat-mode

```
var fs = require('fs');
var Mode = require('stat-mode');

fs.stat('./phoenix5a.png', function(err,stats) {
   if (err) return console.log(err);

   // get permissions
   var mode = new Mode(stats);

   console.log(mode.toString());
   console.log('Group execute ' + mode.group.execute);
   console.log('Others write ' + mode.others.write);
   console.log('Owner read ' + mode.owner.read);
});
```

The following is the output for the given file:

```
-rw-rw-r--
Group execute false
Others write false
Owner read true
```

The File System Watcher

It's not uncommon for an application to "listen" for file or directory changes, and then perform some task when a change occurs. The `fs.FSWatcher` is the interface that handles this in Node. Unfortunately, as Node developers note, it's inconsistent across platforms and not all that useful.

We're going to ignore it and the associated `fs.watch()` that returns the object. Instead, we're going to look at a third-party module. At two-million-plus downloads of Chokidar a month, this module is one of the more heavily utilized (not to mention that it's incorporated into the popular application Gulp).

Install it using (adding `-g` for global):

```
npm install chokidar
```

The following code will add a watcher on the current directory. It checks for directory changes, including changes to files. It performs a recursive watch, picking up any new subdirectories contained under the parent and new files in those subdirectories. The raw event picks up all events, while the other event handlers look for higher-level events.

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

var watcher = chokidar.watch('.', {
  ignored: /[\/\\]\./,
  persistent: true
});

var log = console.log.bind(console);

watcher
  .on('add', function(path) { log('File', path, 'has been added'); })
  .on('unlink', function(path) { log('File', path, 'has been removed'); })
  .on('addDir', function(path) { log('Directory', path, 'has been added'); })
  .on('unlinkDir', function(path) {
       log('Directory', path, 'has been removed'); })
  .on('error', function(error) { log('Error happened', error); })
  .on('ready', function() { log('Initial scan complete. Ready for changes.'); })
  .on('raw', function(event, path, details) {
       log('Raw event info:', event, path, details); });

watcher.on('change', function(path, stats) {
  if (stats) log('File', path, 'changed size to', stats.size);
});
```

Adding and removing files or directories shows up in the console, as does changing the file size. The `unlink()` and `unlinkDir()` functions reflect that "removal" means the objects are no longer linked to the current directory. If this is the last link to the files/subdirectories, they're deleted.

Catching all the raw events could lead to rather bloated output. Still, do check them out when first playing with Chokidar.

File Read and Write

Include the module before use:

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

Most of the examples that use the File System throughout the book use the non-streamed read and write methods. We can take two approaches for reading from or writing to a file using this non-streamed functionality.

The first read/write approach is to use the very simple `fs.readFile()` or `fs.writeFile()` methods (or their synchronous counterparts). These functions open the file, do their read or write, and then close the file. In the following code, a file is opened for writing, truncating its contents if there are any. Once the write is finished, the file is opened for reading, and the contents are read and printed out to the console.

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

fs.writeFile('./some.txt','Writing to a file',function(err) {
```

```
        if (err) return console.error(err);
        fs.readFile('./some.txt','utf-8', function(data,err) {
           if (err) return console.error(err);
           console.log(data);
        });
});
```

Since input and output to a file is via the buffer, by default, I read the file in using the `'utf-8'` option as the second argument to the `fs.readFile()` function. I could also just convert the buffer to a string.

The second read/write approach is to open a file and assign a file descriptor (`fd`). Use the file descriptor to write and/or read from the file. The advantage to this approach is that you have more finite control over how the file is opened and what you can do with it when it is.

In the following code, a file is created, written to, and then read. The second parameter to the `fs.open()` function is the flag that determines what actions can be taken with the file, in this case a value of `'a+'`, to open the file for appending and/or reading, and create the file if it doesn't exist. The third parameter sets the file's permissions (both reading and writing allowed).

```
"use strict";

var fs = require('fs');

fs.open('./new.txt','a+',0x666, function(err, fd) {
     if (err) return console.error(err);
     fs.write(fd, 'First line', 'utf-8', function(err,written, str) {
        if (err) return console.error(err);
        var buf = new Buffer(written);
        fs.read(fd, buf, 0, written, 0, function (err, bytes, buffer) {
           if (err) return console.error(err);
           console.log(buf.toString('utf8'));
        });
      });
});
```

The file descriptor is returned in the callback and then used with the `fs.write()` function. A string is written to the file, beginning at position 0. Note, though, that according to the Node documentation, the data is always written to the end of the file in Linux (the positional indicator is ignored) when the file is opened in append mode.

The callback function for `fs.write()` returns an error (if any occur), the number of bytes written, and the string that was written. Lastly, a `fs.read()` is used to read the line back into a buffer, which is then written out to the console.

Of course, we wouldn't read a line we just wrote, but the example does demonstrate the three primary types of methods used with this approach to reading and writing to a file. You can also directly manipulate a directory in addition to a file.

Directory Access and Maintenance

You can create a directory, remove it, or read the files in it. You can also create a symbolic link or unlink a file, which results in it being deleted (as long as no program has it open). If you want to truncate the file (set it to zero bytes), you can use `truncate()` instead. This leaves the file but removes the content.

To demonstrate some of the directory capability, in the following code the files in the current directory are listed out, and if any are compressed (extension of *.gz*), they're unlinked. The task is simplified through the Path module, covered in "Resource Access with Path" on page 141.

```
'use strict';

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

fs.readdir ('./',function(err, files) {
   for (let file of files) {
      console.log(file);
      if (path.extname(file) == '.gz') {
         fs.unlink('./' + file);
      }
   }
});
```

File Streams

We've been working with readable and writable streams, but I want to take a little time to provide a more in-depth introduction.

You can create a readable stream using `fs.createReadStream()`, passing in a path and `options` object, or specifying a file description in the options and leaving the path null. The same applies to a writable stream, created using `fs.createWriteStream()`. Both support an `options` object. By default, the readable stream is created with the following options:

```
{ flags: 'r',
  encoding: null,
  fd: null,
  mode: 0o666,
  autoClose: true
}
```

If you want to use a file descriptor, you can set it in the options. The `autoClose` option automatically closes the file once the read is finished. If you want to read a section of the file, you can set a begin and end (in bytes) using `start` and `end` in the options. You can specify a `'utf8'` or other encoding, but you can also set this later, using `setEncoding()`.

Streams Are Documented in Multiple Places

The `setEncoding()` function you can call on a readable stream created using `fs.createReadStream()` is actually documented in the Stream section of the Node documentation. If you're working with Node streams, be aware that documentation is spread about and you might have to dig a bit for it.

I'll provide an example of File System readable streams in a bit, but first let's look at the options for the File System's writable stream, created with `fs.createWriteStream()`. Its default options are:

```
{ flags: 'w',
  defaultEncoding: 'utf8',
  fd: null,
  mode: 0o666 }
```

Again, you can use a file descriptor rather than a path. And encoding in a writable stream is set with `defaultEncoding`, rather than `encoding`. If you want to write starting at a specific position after the beginning of the file, you can do so by setting the `start` option. The `end` option isn't specified because the end is whatever the end is once you've written the new content.

Now let's put this all together. In Example 6-2, I'll open a file for modification using a writable stream. All the modification will do is insert a string into a specific position in the file. I'll use a file descriptor in this example, which means when the application calls `fs.createWriteStream()`, it doesn't initiate a file open at the same time that it creates a writable stream.

Example 6-2. Modifying an existing file by inserting a string

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

fs.open('./working.txt', 'r+',function (err, fd) {
   if (err) return console.error(err);

   var writable = fs.createWriteStream(null,{fd: fd, start: 10,
                                        defaultEncoding: 'utf8'});

   writable.write(' inserting this text ');

});
```

Note that the file is opened with the `r+` flag. This allows the application to both read and modify the file.

In Example 6-3, I'll open the same file, but this time I'll read the contents. I use the default `r` flag, since I'm only reading from the file. And I'm reading all of the contents.

I do, however, change the encoding to `utf8`, using `setEncoding()`. In Example 6-2, I changed the encoding for the write to `utf8` by adjusting the `defaultEncoding` flag.

Example 6-3. Reading contents from a file using a stream

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

var readable =
  fs.createReadStream('./working.txt').setEncoding('utf8');

var data = '';
readable.on('data', function(chunk) {
   data += chunk;
});

readable.on('end', function() {
   console.log(data);
});
```

Running the read application before and after the application that modifies the file shows the modification. Running the read application the first time returns the contents of *working.txt*:

```
Now let's pull this all together, and read and write with a stream.
```

A second time returns:

```
Now let's  inserting this text  and read and write with a stream.
```

Now, it would save us all a lot of time if we could just open a file for reading and pipe the results to a writable stream. We can, easily, using the readable stream's `pipe()` function. However, we can't modify the results midstream because the writable stream is just that: writable. It's not a duplex stream or, specifically, a transform stream that can somehow modify the contents. But you can copy the contents from one file to another.

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

var readable =
  fs.createReadStream('./working.txt');

var writable = fs.createWriteStream('./working2.txt');

readable.pipe(writable);
```

We'll see a true transform stream later, in "Compression/Decompression with ZLib" on page 144.

Resource Access with Path

The Node Path utility module is a way of transforming and extracting data from filesystem paths. It also provides an environmentally neutral way of dealing with filesystem paths, so you don't have to code one module for Linux and another for Windows.

We saw the extraction capability earlier when we extracted the file extension from a file while traversing files in a directory:

```
'use strict';

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

fs.readdir ('./',function(err, files) {
   for (let file of files) {
      console.log(file);
      if (path.extname(file) == '.gz') {
         fs.unlink('./' + file);
      }
   }
});
```

If you're interested in getting the base name for the file, use the following code:

```
'use strict';

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

fs.readdir ('./',function(err, files) {
   for (let file of files) {
      let ext = path.extname(file);
      let base = path.basename(file, ext);
      console.log ('file ' + base + ' with extension of ' + ext);
   }
});
```

The second argument to the `path.basename()` function results in just the filename being returned, without the extension.

An example of the environmental neutrality capability in Path is the `path.delimeter` property. This is the delimiter that's system-specific. In Linux, it's a colon (:), while in Windows, it's a semicolon (;). If we wanted to split out the values in the `PATH` environmental variable in an application that can work in both operating systems, we'd use `path.delimeter`:

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

console.log(process.env.PATH);
console.log(process.env.PATH.split(path.delimiter));
```

Now the last works in both environments, returning an array of PATH variables.

Another difference is whether the system uses a forward slash (/) or a backslash (\). In Chapter 5, I created a simple web server that used a filesystem path to serve a resource in response to a web request. In my Windows machine file paths are separated with the backslash, whereas in my Linux machine they're separated by the forward slash. I was able to get the application to work in both environments by using `path.normalize()`:

```
pathname = path.normalize(base + req.url);
```

The key to the Path module isn't that it does amazing string transformations we can't do with the `String` object or RegExp. What it does is transform filesystem paths in an agnostic (operating-system-neutral) manner.

If you want to parse a filesystem path into its components, you can use the `path.parse()` function. The results differ, rather significantly, depending on the operating system. In Windows, when I use `require.main.filename` or the shorthand `__filename` (the path and name of the application I'm executing) as the argument, I get the following:

```
{ root: 'C:\\',
  dir: 'C:\\Users\\Shelley',
  base: 'work',
  ext: '.js',
  name: 'path1' }
```

In my Ubuntu-based server, I get the following:

```
{ root: '/',
  dir: '/home/examples/public_html/learnnode2',
  base: 'path1.js',
  ext: '.js',
  name: 'path1' }
```

Creating a Command-Line Utility

In a Unix environment, you can easily create a Node application that you can run directly without having to use the `node` command.

Windows Command-Line Utilities

To create a Windows command-line utility, you'll need to create a batch file that contains a call to Node in addition to the application.

To demonstrate, I'll use the Commander module, covered in Chapter 3, and a *child process* to access ImageMagick, a powerful graphics tool.

Child Process

Child processes are covered in Chapter 8.

In my application, I'm using ImageMagick to take an existing image file and add a Polaroid effect, saving the result to a new file. As shown in Example 6-4, I use Commander to handle the command-line argument processing and to provide help in using the utility.

Example 6-4. Node as command-line utility application

```
#!/usr/bin/env node

var spawn = require('child_process').spawn;
var program = require('commander');

program
   .version ('0.0.1')
   .option ('-s, --source [file name]', 'Source graphic file name')
   .option ('-f, --file [file name]', 'Resulting file name')
   .parse(process.argv);

if ((program.source === undefined) || (program.file === undefined)) {
   console.error('source and file must be provided');
   process.exit();
}

var photo = program.source;
var file = program.file;

// conversion array
var opts = [
photo,
"-bordercolor", "snow",
"-border", "20",
"-background","gray60",
"-background", "none",
"-rotate", "6",
"-background", "black",
"(", "+clone", "-shadow", "60x8+8+8", ")",
"+swap",
"-background", "none",
"-thumbnail", "240x240",
"-flatten",
file];

var im = spawn('convert', opts);
```

```
im.stderr.on('data', (data) => {
  console.log(`stderr: ${data}`);
});

im.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});
```

To convert it to a command-line utility, I place the following line at the top of the file:

```
#!/usr/bin/env node
```

The characters "#!" are called a *shebang*. What follows is the application that should be used to execute the file, in this case Node. The subdirectory is the path where the application resides.

The file is saved without the *.js* extension. It's also converted into an executable via *chmod*:

```
chmod a+x polaroid
```

Now I can run the utility using the following:

```
./polaroid -h
```

to get help for the utility (thanks to Commander), or:

```
./polaroid -s phoenix5a.png -f phoenix5apolaroid.png
```

to create a new image with a Polaroid effect. The utility doesn't work in Windows, but where it does work, it works well.

Creating a command-line utility is not the same thing as creating a standalone application. The latter implies that you can install the application without having Node (or other dependencies) preinstalled.

Creating a Standalone Application with NW.js

At this time, the only functionality I know that can create a standalone Node application is Intel's NW.js (formerly named node-webkit). You can use it to package your files and then run the package using NW.js, which provides the "innards" to make the whole thing work.

Compression/Decompression with ZLib

The ZLib module provides compression/decompression functionality. It's also based on a transform stream, which is immediately apparent when you see the example the Node documentation provides for compressing a file. I modified it slightly to work with a large file.

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

var gzip = zlib.createGzip();

var inp = fs.createReadStream('test.png');
var out = fs.createWriteStream('test.png.gz');

inp.pipe(gzip).pipe(out);
```

The input stream is directly connected to the output with the gzip compression in the middle, transforming the content—in this case, compressing the PNG file.

Zlib provides support to use `zlib` compression or `deflate`, which is a more complex, controllable compression algorithm. Note that, unlike `zlib` where you can uncompress the file using the gunzip (or unzip) command-line utility, you don't have that opportunity with `deflate`. You'll have to use Node or some other functionality to uncompress a file that was compressed with `deflate`.

To demonstrate the functionality to both compress and uncompress a file, we'll create two command-line utilities: compress and uncompress. The first will compress a file using `gzip` or `deflate` as an option. Since we're dealing with options, we'll also use the Commander module to handle command-line options:

```
var zlib = require('zlib');
var program = require('commander');
var fs = require('fs');

program
   .version ('0.0.1')
   .option ('-s, --source [file name]', 'Source File Name')
   .option ('-f, --file [file name]', 'Destination File name')
   .option ('-t, --type <mode>', /^(gzip|deflate)$/i)
   .parse(process.argv);


var compress;
if (program.type == 'deflate')
   compress = zlib.createDeflate();
else
   compress = zlib.createGzip();

var inp = fs.createReadStream(program.source);
var out = fs.createWriteStream(program.file);

inp.pipe(compress).pipe(out);
```

The utility applications are interesting and useful (especially in a Windows environment, which doesn't have native access to this type of compression functionality), but a popular use for compression technologies is in web requests. The Node documentation contains examples for using Zlib functionality with web requests. There are also

several examples for retrieving a compressed file using the Request module (covered in Chapter 5) and the `http.request()` function.

Instead of retrieving a compressed file, I'm going to demonstrate how to send a compressed file through to a server which will then decompress it. I'm adapting the server and client applications from Examples 5-1 and 5-2 but modifying the code to compress a large PNG file and send it through via an HTTP request, where the server will then decompress the data and save the file.

The server code is given in Example 6-5. Note that the data being sent is retrieved as an array of chunks, which eventually are used to create a new Buffer using `buffer.concat()`. Since we're dealing with a Buffer and not a stream, I can't use the `pipe()` function. Instead, I'll use the Zlib convenience function, `zlib.unzip`, passing in the Buffer and a callback function. The callback function has an error and a result as arguments. The result is also a Buffer, which is written out to a newly created writable stream using the `write()` function. To ensure separate instances of the file, I use a timestamp to modify the filename.

Example 6-5. Creating a web server that accepts compressed data and decompresses it to a file

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

var server = http.createServer().listen(8124);

server.on('request', function(request,response) {

   if (request.method == 'POST') {
      var chunks = [];

      request.on('data', function(chunk) {
         chunks.push(chunk);
      });

      request.on('end', function() {
         var buf = Buffer.concat(chunks);
         zlib.unzip(buf, function(err, result) {
            if (err) {
               response.writeHead(500);
               response.end();
               return console.log('error ' + err);
            }
            var timestamp = Date.now();
            var filename = './done' + timestamp + '.png';
            fs.createWriteStream(filename).write(result);
         });
```

```
        response.writeHead(200, {'Content-Type': 'text/plain'});
        response.end('Received and undecompressed file\n');
      });
  }
});

console.log('server listening on 8214');
```

The key in the client code, given in Example 6-6, is to ensure that the proper `Content-Encoding` is given in the header. It should be `'gzip,deflate'`. The `Content-Type` is also changed to `'application/javascript'`.

Example 6-6. Client that compresses a file and pipes it to a web request

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

var gzip = zlib.createGzip();

var options = {
  hostname: 'localhost',
  port: 8124,
  method: 'POST',
  headers: {
    'Content-Type': 'application/javascript',
    'Content-Encoding': 'gzip,deflate'
  }
};

var req = http.request(options, function(res) {
  res.setEncoding('utf8');
  var data = '';
  res.on('data', function (chunk) {
      data+=chunk;
  });


  res.on('end', function() {
    console.log(data)
  })

});

req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});

// stream gzipped file to server
var readable = fs.createReadStream('./test.png');
readable.pipe(gzip).pipe(req);
```

The client opens the file to be compressed and pipes it to a Zlib compression transform stream, which then pipes the result to the web request (which is a writable stream). We're dealing purely with streams in this code, so we can use the `pipe()` functionality we used earlier. We can't use it with the server because the data is transmitted as buffer chunks.

Buffering the file in memory can be a scaling problem, so another approach is to save the uncompressed file, uncompress it, and then delete the temporary uncompressed file. I'll leave that for an off-book exercise.

Pipes and ReadLine

We've been using pipes in Chapter 5 and this chapter, and one of the simplest demonstrations of a pipe is to open a REPL session and type in the following:

```
> process.stdin.resume();
> process.stdin.pipe(process.stdout);
```

Everything you type from this point on is echoed back to you.

If you want to keep the output stream open for continued data, pass an option, `{ end: false }`, to the output stream:

```
process.stdin.pipe(process.stdout, { end : false });
```

REPL's line-by-line processing is actually implemented with the last core module we'll look at in this chapter: Readline. Merely importing Readline also starts a seemingly neverending communication thread. You include the Readline module with code like the following:

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

Be aware, though, that once you include this module, the Node program doesn't terminate until you close the interface.

The Node site documentation contains an example of how to begin and terminate a Readline interface, which I have adapted in Example 6-7. The application asks a question as soon as you run it and then outputs the answer. It also listens for any "command," which is really any line that terminates with `\n`. If the command is `.leave`, it leaves the application; otherwise, it just repeats the command and prompts the user for more. A Ctrl-C or Ctrl-D key combination also causes the application to terminate, albeit less gracefully.

Example 6-7. Using Readline to create a simple, command-driven user interface

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

// create a new interface
var rl = readline.createInterface(process.stdin, process.stdout);

// ask question
rl.question(">>What is the meaning of life?  ", function(answer) {
   console.log("About the meaning of life, you said " + answer);
   rl.setPrompt(">> ");
   rl.prompt();
});

// function to close interface
function closeInterface() {
   rl.close();
   console.log('Leaving Readline');
}

// listen for .leave
rl.on('line', function(cmd) {
   if (cmd.trim() == '.leave') {
      closeInterface();
      return;
   }
   console.log("repeating command: " + cmd);
   rl.prompt();
});

rl.on('close', function() {
    closeInterface();
});
```

Here's an example session:

```
>>What is the meaning of life?  ===
About the meaning of life, you said ===
>>This could be a command
repeating command: This could be a command
>>We could add eval in here and actually run this thing
repeating command: We could add eval in here and actually run this thing
>>And now you know where REPL comes from
repeating command: And now you know where REPL comes from
>>And that using rlwrap replaces this Readline functionality
repeating command: And that using rlwrap replaces this Readline functionality
>>Time to go
repeating command: Time to go
>>.leave
Leaving Readline...
Leaving Readline...
```

This should look familiar. Remember from Chapter 4 that we can use `rlwrap` to override the command-line functionality for REPL. We use the following to trigger its use:

```
env NODE_NO_READLINE=1 rlwrap node
```

And now we know what the flag is triggering—it's instructing REPL not to use Node's Readline module for command-line processing, and to use `rlwrap` instead.

CHAPTER 7

Networking, Sockets, and Security

The core of a Node application invariably relies on two primary infrastructure components: networks and security. And you can't talk about networks without also discussing sockets.

I group networks and security because once you start moving beyond a single, isolated machine, security should be paramount in your mind at all times. Every time you finish a new piece of the application, the first question you should ask yourself is: is this secure? No amount of graceful coding can compensate for letting crap in.

Servers, Streams, and Sockets

Much of the Node core API is related to creating services that listen to specific types of communications. In Chapters 1 and 5 we used the HTTP module to create web servers listening for HTTP requests. Other modules can create a Transmission Control Protocol (TCP) server, a Transport Layer Security (TLS) server, and a User Datagram Protocol (UDP) socket. I'll cover TLS later in the chapter, but in this section I want to introduce the TCP and UDP Node core functionality. First, though, a quick look at sockets.

Sockets and Streams

A *socket* is an endpoint in a communication, and a *network socket* is an endpoint in a communication between applications running on two different computers on the network. The data flows between the sockets in what's known as a *stream*. The data in the stream can be transmitted as binary data in a buffer, or in Unicode as a string. Both types of data are transmitted as *packets*: parts of the data split off into similar-sized pieces. There is a special kind of packet, a finish packet (FIN), that is sent by a socket to signal that the transmission is done.

Think of two people talking by walkie-talkie. The walkie-talkies are the endpoints, the *sockets* of the communication. When two or more people want to speak to each other, they have to tune into the same radio frequency. Then when one person wants to communicate with the other, they push a button on the walkie-talkie, and connect to the appropriate person using some form of identification. They also use the word "over" to signal that they're no longer talking, but listening instead. The other person pushes the talk button on their walkie-talkie, acknowledges the communication, and again uses "over" to signal when they are done and in listening mode. The communication continues until one person signals "over and out," which means that the communication is finished. Only one person can talk at a time.

Walkie-talkie communication streams are known as *half-duplex*, because communication can only occur in one direction at a time. *Full-duplex* communication streams allow two-way communication.

The concepts also apply to Node streams. We've already worked with half- and full-duplex streams in Chapter 6. The streams used to write to and read from files were examples of half-duplex streams: the streams supported either the readable interface or the writable interface but not both at the same time. The zlib compression stream is an example of a full-duplex stream, allowing simultaneous reading and writing.

Now we're taking what we've learned and applying it to networked (TCP) as well as encrypted (Crypto) streams. We'll look at Crypto later in the chapter, but first let's dive into TCP.

TCP Sockets and Servers

TCP provides the communication platform for most Internet applications such as web service and email. It provides a way of reliably transmitting data between client and server sockets. TCP provides the infrastructure on which the application layer, such as HTTP, resides.

We can create a TCP server and client just as we did with HTTP, with some differences. When creating a TCP server, rather than passing a `requestListener` to the server creation function, with its separate response and request objects, the TCP callback function's sole argument is an instance of a socket that can both send and receive.

To better demonstrate how TCP works, Example 7-1 contains the code to create a TCP server. Once the server socket is created, it listens for two events: when data is received and when the client closes the connection. It then prints out the data it received to the console and repeats the data back to the client.

The TCP server also attaches an event handler for both the `listening` event and an `error`. In previous chapters, I just plopped a `console.log()` message after the server was created, following pretty standard practice in Node. However, since the `listen()`

event is asynchronous, technically this practice is incorrect—the message prints out technically before the listen event has happened. Instead, you can incorporate a message in the callback function for the `listen` function, or you can do what I'm doing here, which is attach an event handler to the `listening` event and correctly providing feedback.

In addition, I'm also providing more sophisticated error handling modeled after that in the Node documentation. The application is processing an `error` event, and if the error is because the port is currently in use, it waits a small amount of time and tries again. For other errors—such as accessing a port like 80, which requires special privileges—the full error message is printed out to the console.

Example 7-1. A simple TCP server, with a socket listening for client communication on port 8124

```
var net = require('net');
const PORT = 8124;

var server = net.createServer(function(conn) {
   console.log('connected');

   conn.on('data', function (data) {
      console.log(data + ' from ' + conn.remoteAddress + ' ' +
        conn.remotePort);
      conn.write('Repeating: ' + data);
   });

   conn.on('close', function() {
        console.log('client closed connection');
   });

}).listen(PORT);

server.on('listening', function() {
    console.log('listening on ' + PORT);
});

server.on('error', function(err){
  if (err.code == 'EADDRINUSE') {
    console.warn('Address in use, retrying...');
    setTimeout(() => {
      server.close();
      server.listen(PORT);
    }, 1000);
  }
  else {
    console.error(err);
  }
});
```

When creating the TCP socket, you can pass in an optional parameters object, consisting of two values: `pauseOnConnect` and `allowHalfOpen`. The default value for both is `false`:

```
{ allowHalfOpen: false,
pauseOnConnect: false }
```

Setting `allowHalfOpen` to `true` instructs the socket not to send a FIN when it receives a FIN packet from the client. Doing this keeps the socket open for writing (not reading). To close the socket, you must use the `end()` function. Setting `pauseOnConnect` to `true` allows the connection to be passed, but no data is read. To begin reading data, call the `resume()` method on the socket.

To test the server, you can use a TCP client application such as the netcat utility (nc) in Linux or OS X, or an equivalent Windows application. Using netcat, the following connects to the server application at port 8124, writing data from a text file to the server:

```
nc burningbird.net 8124 < mydata.txt
```

In Windows, there are TCP tools, such as SocketTest, that you can use for testing.

Rather than using a tool to test the server, you can create your own TCP client application. The TCP client is just as simple to create as the server, as shown in Example 7-2. The data is transmitted as a buffer, but we can use `setEncoding()` to read it as a `utf8` string. The socket's `write()` method is used to transmit the data. The client application also attaches listener functions to two events: `data`, for received data, and `close`, in case the server closes the connection.

Example 7-2. The client socket sending data to the TCP server

```
var net = require('net');
var client = new net.Socket();
client.setEncoding('utf8');

// connect to server
client.connect ('8124','localhost', function () {
   console.log('connected to server');
   client.write('Who needs a browser to communicate?');
});

// when receive data, send to server
process.stdin.on('data', function (data) {
   client.write(data);
});

// when receive data back, print to console
client.on('data',function(data) {
   console.log(data);
```

```
});

// when server closed
client.on('close',function() {
   console.log('connection is closed');
});
```

The data being transmitted between the two sockets is typed in at the terminal and transmitted when you press Enter. The client application sends the text you type, and the TCP server writes out to the console when it receives it. The server repeats the message back to the client, which in turn writes the message out to its console. The server also prints out the IP address and port for the client using the socket's `remoteAddress` and `remotePort` properties. Following is the console output for the server after several strings were sent from the client:

```
Hey, hey, hey, hey-now.
 from ::ffff:127.0.0.1 57251
Don't be mean, we don't have to be mean.
 from ::ffff:127.0.0.1 57251
Cuz remember, no matter where you go,
 from ::ffff:127.0.0.1 57251
there you are.
 from ::ffff:127.0.0.1 57251
```

The connection between the client and server is maintained until you kill one or the other using Ctrl-C. Whichever socket is still open receives a `close` event that's printed out to the console. The server can also serve multiple connections from multiple clients, since all the relevant functions are asynchronous.

IPv4 Mapped to IPv6 Addresses

The sample output from running the client/server TCP applications in this section demonstrate an IPv4 address mapped to IPv6, with the addition of `::ffff`.

Instead of binding to a port with the TCP server, we can bind directly to a socket. To demonstrate this capability, I modified the TCP server from the earlier examples, but instead of binding to a port, the new server binds to a *Unix socke*t, as shown in Example 7-3. The Unix socket is a pathname somewhere on your server. The read and write permissions can be used to finitely control application access, which makes the approach more advantageous than Internet sockets.

I also had to modify the error handling to unlink the Unix socket if the application is restarted and the socket is already in use. In a production environment, you'd want to make sure no other clients are using the socket before you do something so abrupt.

Example 7-3. TCP server bound to a Unix socket

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

const unixsocket = '/somepath/nodesocket';

var server = net.createServer(function(conn) {
   console.log('connected');

   conn.on('data', function (data) {
      conn.write('Repeating: ' + data);
   });

   conn.on('close', function() {
        console.log('client closed connection');
   });

}).listen(unixsocket);

server.on('listening', function() {
    console.log('listening on ' + unixsocket);
});



// if exit and restart server, must unlink socket
server.on('error',function(err) {
   if (err.code == 'EADDRINUSE') {
      fs.unlink(unixsocket, function() {
          server.listen(unixsocket);
      });
   } else {
      console.log(err);
   }
});

process.on('uncaughtException', function (err) {
    console.log(err);
});
```

I also used `process` as an added precaution against an exception that isn't managed by the application being thrown.

Check If Another Instance of the Server Is Running

Before unlinking the socket, you can check to see if another instance of the server is running. A solution at Stack Overflow provides an alternative clean-up technique that accounts for this situation.

The client application is shown in Example 7-4. It's not that much different than the earlier client that communicated with the port. The only difference is adjusting for the connection point.

Example 7-4. Connecting to the Unix socket and printing out received data

```
var net = require('net');
var client = new net.Socket();
client.setEncoding('utf8');

// connect to server
client.connect ('/somepath/nodesocket', function () {
   console.log('connected to server');
   client.write('Who needs a browser to communicate?');
});

// when receive data, send to server
process.stdin.on('data', function (data) {
   client.write(data);
});

// when receive data back, print to console
client.on('data',function(data) {
   console.log(data);
});

// when server closed
client.on('close',function() {
   console.log('connection is closed');
});
```

"Guards at the Gate" on page 159 covers the SSL version of HTTP, HTTPS, along with Crypto and TLS/SSL.

UDP/Datagram Socket

TCP requires a dedicated connection between the two endpoints of the communication. UDP is a connectionless protocol, which means there's no guarantee of a connection between the two endpoints. For this reason, UDP is less reliable and robust than TCP. On the other hand, UDP is generally faster than TCP, which makes it more popular for real-time uses, as well as technologies such as Voice over Internet Protocol (VoIP), where the TCP connection requirements could adversely impact the quality of the signal.

Node core supports both types of sockets. In the last section, I demonstrated the TCP functionality. Now, it's UDP's turn.

The UDP module identifier is `dgram`:

```
require ('dgram');
```

To create a UDP socket, use the `createSocket` method, passing in the type of socket—either `udp4` or `udp6`. You can also pass in a callback function to listen for events. Unlike messages sent with TCP, messages sent using UDP must be sent as buffers, not strings.

Example 7-5 contains the code for a demonstration UDP client. In it, data is accessed via `process.stdin` and then sent as is via the UDP socket. Note that we don't have to set the encoding for the string, since the UDP socket accepts only a buffer, and the `process.stdin` data *is* a buffer. We do, however, have to convert the buffer to a string, using the buffer's `toString` method, in order to get a meaningful string for echoing the data via `console.log()`.

Example 7-5. A datagram client that sends messages typed into the terminal

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

var client = dgram.createSocket("udp4");

process.stdin.on('data', function (data) {
   console.log(data.toString('utf8'));
   client.send(data, 0, data.length, 8124, "examples.burningbird.net",
      function (err, bytes) {
        if (err)
          console.error('error: ' + err);
        else
          console.log('successful');
   });
});
```

The UDP server, shown in Example 7-6, is even simpler than the client. All the server application does is create the socket, bind it to a specific port (8124), and listen for the `message` event. When a message arrives, the application prints it, as well as the IP address and sender port, using `console.log`. No encoding is necessary to print out the message—it's automatically converted from a buffer to a string.

We didn't have to bind the socket to a port. However, without the binding, the socket would attempt to listen in on every port.

Example 7-6. A UDP socket server, bound to port 8124, listening for messages

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

var server = dgram.createSocket("udp4");

server.on ("message", function(msg, rinfo) {
   console.log("Message: " + msg + " from " + rinfo.address + ":"
               + rinfo.port);
});

server.bind(8124);
```

I didn't call the `close` method on either the client or the server after sending/receiving the message: no connection is being maintained between the client and server—just the sockets capable of sending a message and receiving communication.

Guards at the Gate

Security in web applications goes beyond ensuring that people don't have access to the application server. Security can be complex, and even a little intimidating. Luckily when it comes to Node applications, several of the components we need for security have already been created. We just need to plug them in, in the right place and at the right time.

Setting Up TLS/SSL

Secure, tamper-resistant communication between a client and server occurs over Secure Sockets Layer (SSL) and its upgrade, Transport Layer Security (TLS). TLS/SSL provides the underlying encryption for HTTPS, which I cover in the next section. However, before we can develop for HTTPS, we have to do some environmental setup.

A TLS/SSL connection requires a *handshake* between client and server. During the handshake, the client (typically a browser) lets the server know what kind of security functions it supports. The server picks a function and then sends an *SSL certificate*, which includes a public key. The client confirms the certificate and generates a random number using the server's key, sending it back to the server. The server then uses its private key to decrypt the number, which in turn is used to enable the secure communication.

For all this to work, you'll need to generate both the public and private key as well as the certificate. For a production system, the certificate would be signed by a *trusted authority*, such as a domain registrar, but for development purposes you can make use of a *self-signed certificate*. Doing so generates a rather significant warning in the

browser for anyone accessing the application, but since the development site isn't being accessed by users, this shouldn't be an issue.

Preventing Self-Signed Certificate Warnings

If you're using a self-signed certificate, you can avoid browser warnings if you access the Node application via localhost (i.e., *https://localhost:8124*). You can also avoid using self-signed certificates without the cost of a commercial signing authority by using Lets Encrypt, currently in open beta. The site provides documentation for setting up the certificate.

The tool used to generate the necessary files is OpenSSL. If you're using Linux, it should already be installed; there's a binary installation for Windows, and Apple is pursuing its own Crypto library. In this section, I'm just covering setting up a Linux environment.

To start, type the following at the command line:

```
openssl genrsa -des3 -out site.key 2048
```

The command generates the private key, encrypted with Triple-DES and stored in privacy-enhanced mail (PEM) format, making it ASCII-readable.

You'll be prompted for a password, which you'll need for the next task, creating a certificate-signing request (CSR).

When generating the CSR, you'll be prompted for the password you just created. You'll also be asked a lot of questions, including the country designation (such as US for United States), your state or province, city name, company name and organization, and email address. The question of most importance is the one asking for the Common Name. This is asking for the hostname of the site—for example, *burningbird.net* or *yourcompany.com*. Provide the hostname where the application is being served. In my example code, I created a certificate for `examples.burningbird.net`.

```
openssl req -new -key site.key -out site.csr
```

The private key wants a *passphrase*. The problem is, every time you start up the server you'll have to provide this passphrase. Having to provide this passphrase is an issue in a production system. In the next step, you'll remove the passphrase from the key. First, rename the key:

```
mv site.key site.key.org
```

Then type:

```
openssl rsa -in site.key.org -out site.key
```

If you do remove the passphrase, make sure your server is secure by ensuring that the file is readable only by trusted users/groups.

The next task is to generate the self-signed certificate. The following command creates one that's good for 365 days:

```
openssl x509 -req -days 365 -in site.csr -signkey site.key -out final.crt
```

Now you have all the components you need in order to use TLS/SSL and HTTPS.

Working with HTTPS

Web pages that ask for user login or credit card information should be served as HTTPS. If they aren't, your data is being transmitted in the open and easily scooped up. HTTPS is a variation of the HTTP protocol that's combined with SSL to ensure that the website is who and what we think it is, that the data is encrypted during transit, and the data arrives intact and without any tampering.

Adding support for HTTPS is similar to adding support for HTTP, with the addition of an `options` object that provides the public encryption key and the signed certificate. The default port for an HTTPS server differs, too: HTTP is served via port 80 by default, while HTTPS is served via port 443.

Avoiding the Port Number

One advantage to using SSL with your Node application is that the default port for HTTPS is 443, which means you don't have to specify the port number when accessing the application, and you won't conflict with your Apache or other web server. Unless you're also utilizing HTTPS with your non-Node web server, of course.

Example 7-7 demonstrates a very basic HTTPS server. It does little beyond sending a variation of our traditional Hello World message to the browser.

Example 7-7. Creating a very simple HTTPS server

```
var   fs = require("fs"),
      https = require("https");

var privateKey = fs.readFileSync('site.key').toString();
var certificate = fs.readFileSync('final.crt').toString();

var options = {
   key: privateKey,
   cert: certificate
};

https.createServer(options, function(req,res) {
```

```
    res.writeHead(200);
    res.end("Hello Secure World\n");
}).listen(443);
```

The public key and certificate are opened and their contents read synchronously. The data is attached to the `options` object, passed as the first parameter in the `https.createServer` method. The callback function for the same method is the one we're used to, with the server request and response object passed as parameters.

When you run the Node application, you'll need to do so with root permissions. That's because the server is running bound to the default of port 443. Binding to any port less than 1024 requires root privilege. You can run it using another port, such as 3000, and it works fine except that when you access the site, you'll need to use the port:

```
https://examples.burningbird.net:3000
```

Accessing the page demonstrates what happens when we use a self-signed certificate, as shown in Figure 7-1. It's easy to see why a self-signed certificate should be used only during testing. Accessing the page using localhost also disables the security warning.

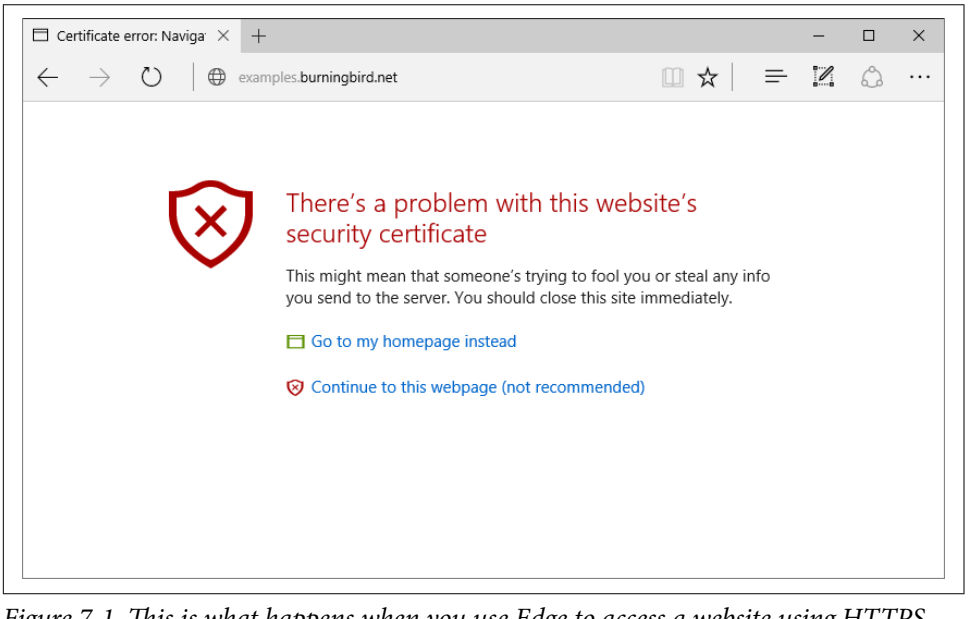


Figure 7-1. This is what happens when you use Edge to access a website using HTTPS with a self-signed certificate

The browser address bar demonstrates another way that the browser signals that the site's certificate can't be trusted, as shown in Figure 7-2. Rather than displaying a lock

indicating that the site is being accessed via HTTPS, it displays a lock with a red *x* showing that the certificate can't be trusted. Clicking the icon opens an information window with more details about the certificate.

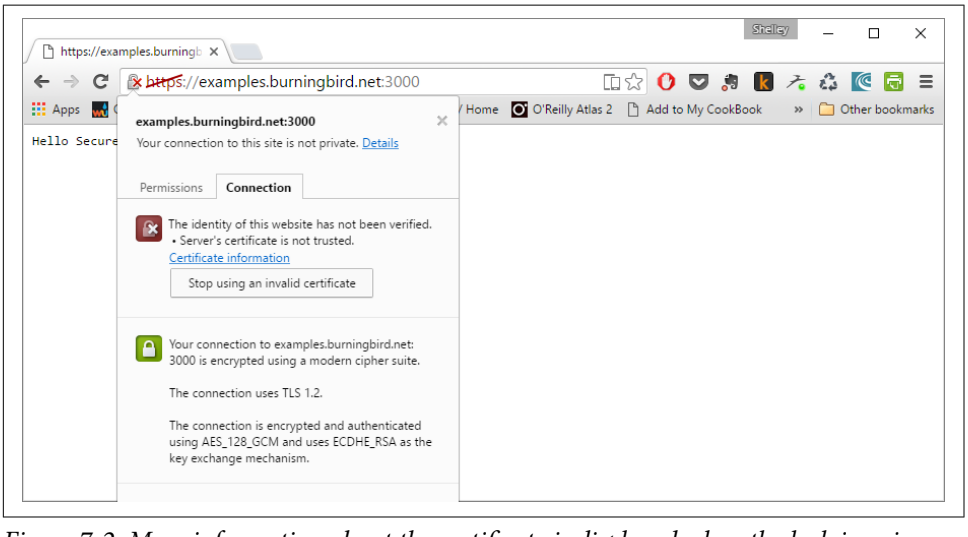


Figure 7-2. More information about the certificate is displayed when the lock icon is clicked, as demonstrated in Chrome

Again, using a trusted certificate authority for a signed certificate eliminates all of these rather intimidating warnings. And since we now have no-cost options for the certificate, you might consider implementing HTTPS for all of your web-facing applications, not just those dealing with payments or passwords.

The Crypto Module

Node provides a module used for cryptography, Crypto, which is an interface for OpenSSL functionality. This includes wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign, and verify functions. The Node component of the technology is actually rather simple to use, but there is a strong underlying assumption that the Node developer knows and understands OpenSSL and what all the various functions are.

Become Familiar with OpenSSL

I cannot recommend strongly enough that you spend time becoming familiar with OpenSSL before working with Node's Crypto module. You can explore OpenSSL's documentation, and you can also access a free book on OpenSSL, *OpenSSL Cookbook*, by Ivan Ristić.

I'm covering a relatively straightforward use of the Crypto module to create a password hash using OpenSSL's `hash` functionality. The same type of functionality can also be used to create a hash for use as a *checksum* to ensure that data that's stored or transmitted hasn't been corrupted in the process.

MySQL

The example in this section makes use of MySQL. For more information on the node-mysql module used in this section, see the module's GitHub repository. If you don't have access to MySQL, you can store the username, password, and salt in a local file and modify the example accordingly.

You can use the Crypto module's `createHash` method to create a password hash for storing in a database. An example is the following, which creates the hash using the `sha1` algorithm, uses the hash to encode a password, and then extracts the digest of the data to store in the database:

```
var hashpassword = crypto.createHash('sha1')
                                 .update(password)
                                 .digest('hex');
```

The digest encoding is set to hexadecimal. Encoding is binary, by default, and base64 can also be used.

Encrypting Password or Storing a Hash

Storing encrypted passwords is better than storing the passwords as plain text, but passwords can be cracked if an agency or individual has the encryption key. Storing the password as a hash is a safer approach, albeit one that can't be reversed. If a person loses a password, the system allows them to reset it rather than attempt to recover it. For more on this topic, check out "Safely Storing User Passwords: Hashing vs. Encrypting".

Many applications use a hash for this purpose. However, there's a problem with storing plain hashed passwords in a database, a problem that goes by the innocuous name of *rainbow table*.

Put simply, a rainbow table is basically a table of precomputed hash values for every possible combination of characters. So even if you have a password that you're sure can't be cracked—and let's be honest, most of us rarely do—chances are that sequence of characters has a place somewhere in a rainbow table, which makes it much simpler to determine what your password is.

The way around the rainbow table is with *salt* (no, not the crystalline variety), a unique generated value that is concatenated to the password before encryption. It can be a single value that is used with all the passwords and stored securely on the server. A better option, though, is to generate a unique salt for each user password and then store it with the password. True, the salt can also be stolen at the same time as the password, but it would still require the person attempting to crack the password to generate a rainbow table specifically for the one and only password—adding immensely to the complexity of cracking any individual password.

Example 7-8 is a simple application that takes a username and password passed as command-line arguments, generates the password hash, and then stores both as a new user in a MySQL database table. I'm using MySQL rather than any other database because it is ubiquitous on most systems, and a system with which most people are familiar.

To follow along with the example, use the node-mysql module by installing it as follows:

```
npm install node-mysql
```

The table is created with the following SQL:

```
CREATE TABLE user (userid INT NOT NULL AUTO_INCREMENT, PRIMARY KEY(userid),
username VARCHAR(400) NOT NULL, passwordhash VARCHAR(400) NOT NULL,
salt DOUBLE NOT NULL );
```

The salt consists of a date value multiplied by a random number and rounded. It's concatenated to the password before the password hash is created. All the user data is then inserted into the MySQL user table.

Example 7-8. Using Crypto's createHash method and a salt to encrypt a password

```
var mysql = require('mysql'),
    crypto = require('crypto');

var connection = mysql.createConnection({
   host: 'localhost',
   user: 'username',
   password: 'userpass'
  });

connection.connect();

connection.query('USE nodedatabase');

var username = process.argv[2];
var password = process.argv[3];

var salt = Math.round((Date.now() * Math.random())) + '';
```

```
var hashpassword = crypto.createHash('sha512')
                   .update(salt + password, 'utf8')
                   .digest('hex');
// create user record
connection.query('INSERT INTO user ' +
   'SET username = ?, passwordhash = ?, salt = ?
   [username, hashpassword, salt], function(err, result) {
      if (err) console.error(err);
      connection.end();
});
```

Following the code from the top, first a connection is established with the database. Next, the database with the newly created table is selected. The username and password are pulled in from the command line, and then the crypto magic begins.

The salt is generated and passed into the function to create the hash using the sha512 algorithm. Functions to update the password hash with the salt and set the hash encoding are chained to the function to create the hash. The newly encrypted password hash is then inserted into the newly created table along with the username.

The application to test a username and password, shown in Example 7-9, queries the database for the password hash and salt based on the username. It uses the salt to, again, generate the hash. Once the password hash has been re-created, it's compared to the password stored in the database. If the two don't match, the user isn't validated. If they match, then the user's in.

Example 7-9. Checking a username and password

```
var mysql = require('mysql'),
    crypto = require('crypto');

var connection = mysql.createConnection({
   user: 'username',
   password: 'userpass'
  });

connection.query('USE nodedatabase');

var username = process.argv[2];
var password = process.argv[3];

connection.query('SELECT password, salt FROM user WHERE username = ?',
   [username], function(err, result, fields) {
   if (err) return console.error(err);

   var newhash = crypto.createHash('sha512')
                 .update(result[0].salt + password, 'utf8')
                 .digest('hex');

   if (result[0].password === newhash) {
```

```
      console.log("OK, you're cool");
   } else {
      console.log("Your password is wrong. Try again.");
   }
   connection.end();
});
```

Trying out the applications, we first pass in a username of `Michael`, with a password of `apple*frk13*`:

```
node password.js Michael apple*frk13*
```

We then check the same username and password:

```
node check.js Michael apple*frk13*
```

and get back the expected result:

```
OK, you're cool
```

Trying it again, but with a different password:

```
node check.js Michael badstuff
```

we get back the expected result again:

```
Your password is wrong. Try again
```

The crypto hash can also be used in a stream. As an example, consider a checksum, which is an algorithmic way of determining if data has transmitted successfully. You can create a hash of the file, and pass this along with the file when transmitting it. The person who downloads the file can then use the hash to verify the accuracy of the transmission. The following code uses the `pipe()` function and the duplex nature of the Crypto functions to create such a hash.

```
var crypto = require('crypto');
var fs = require('fs');
var hash = crypto.createHash('sha256');
hash.setEncoding('hex');

var input = fs.createReadStream('main.txt');
var output = fs.createWriteStream('mainhash.txt');

input.pipe(hash).pipe(output);
```

You can also use `md5` as the algorithm in order to generate a MD5 checksum, which is popular in most environments and applications because it's faster, though not as secure.

```
var hash = crypto.createHash('md5');
```

CHAPTER 8

Child Processes

Operating systems provide access to a great deal of functionality, but much of it is only accessible via the command line. It would be nice to be able to access this functionality from a Node application. That's where *child processes* come in.

Node allows us to run a system command within a child process and listen in on its input/output. This includes being able to pass arguments to the command, and even pipe the results of one command to another. The next several sections explore this functionality in more detail.

All but the last examples demonstrated in this chapter use Unix commands. They work on a Linux system and should also work in OS X. They won't, however, work in a Windows Command window.

child_process.spawn

There are four different techniques you can use to create a child process. The most common one is using the `spawn` method. This technique launches a command in a new process, passing in any arguments. Pipes are established between the parent application and the child process for `stdin`, `stdout`, and `stderr`.

In the following, we create a child process to call the Unix `pwd` command to print the current directory. The command takes no arguments:

```
var spawn = require('child_process').spawn,
    pwd = spawn('pwd');

pwd.stdout.on('data', function (data) {
  console.log('stdout: ' + data);
});

pwd.stderr.on('data', function (data) {
  console.error('stderr: ' + data);
});

pwd.on('close', function (code) {
  console.log('child process exited with code ' + code);
});
```

Notice the events that are captured on the child process's `stdout` and `stderr`. If no error occurs, any output from the command is transmitted to the child process's `stdout`, triggering a `data` event on the process. If an error occurs, such as in the following where we're passing an invalid option to the command:

```
var spawn = require('child_process').spawn,
    pwd = spawn('pwd', ['-g']);
```

then the error gets sent to `stderr`, which prints out the error to the console:

```
stderr: pwd: invalid option -- 'g'
Try `pwd --help' for more information.

child process exited with code 1
```

The process exited with a code of `1`, which signifies that an error occurred. The exit code varies depending on the operating system and error. When no error occurs, the child process exits with a code of `0`.

The earlier code demonstrated sending output to the child process's `stdout` and `stderr`, but what about `stdin`? The Node documentation for child processes includes an example of directing data to `stdin`. It's used to emulate a Unix pipe (`|`) whereby the result of one command is immediately directed as input to another command. I adapted the example in order to demonstrate one use of the Unix pipe—being able to look through all subdirectories, starting in the local directory, for a file with a specific word (in this case, *test*) in its name:

```
find . -ls | grep test
```

Example 8-1 implements this functionality as child processes. Note that the first command, which performs the `find`, takes two arguments, whereas the second one takes just one: a term passed in via user input from `stdin`. Also note that the `grep` child process's `stdout` encoding is changed via `setEncoding`. Otherwise, when the data is printed out, it would be printed out as a buffer.

Example 8-1. Using child processes to find files in subdirectories with a given search term, "test"

```
var spawn = require('child_process').spawn,
    find = spawn('find',['.','-ls']),
    grep = spawn('grep',['test']);

grep.stdout.setEncoding('utf8');

// pipe find output to grep input
find.stdout.pipe(grep.stdin);

// now run grep and output results
grep.stdout.on('data', function (data) {
  console.log(data);
});

// error handling for both
find.stderr.on('data', function (data) {
  console.log('grep stderr: ' + data);
});
grep.stderr.on('data', function (data) {
  console.log('grep stderr: ' + data);
});

// and exit handling for both
find.on('close', function (code) {
  if (code !== 0) {
    console.log('find process exited with code ' + code);
  }
});

grep.on('exit', function (code) {
  if (code !== 0) {
    console.log('grep process exited with code ' + code);
  }
});
```

When you run the application, you'll get a listing of all files in the current directory and any subdirectories that contain *test* in their filename.

The Node documentation contains a warning that some programs use line-buffered I/O internally. This could result in the data that is being sent to the program not being immediately consumed. What this means for us is that, with some child processes, the data is buffered in blocks before processing. The `grep` child process is one such process.

In the current example, the output from the find process is limited, so the input to the `grep` process does not exceed the block size (typically 4096, but it can differ based on system and individual settings).

More on stdio Buffering

For a more in-depth look at buffering, check out "How to fix stdio buffering" and its companion piece "Buffering in standard streams".

We can turn off line buffering for `grep` by using the `--line-buffered` option. In the following application—using the process status (`ps`) command to examine running processes, and then searching for instances of apache2—line buffering is turned off for `grep`, and data is printed out immediately rather than when the buffer is full:

```
var spawn = require('child_process').spawn,
    ps    = spawn('ps', ['ax']),
    grep  = spawn('grep', ['--line-buffered', 'apache2']);

ps.stdout.pipe(grep.stdin);

ps.stderr.on('data', function (data) {
  console.log('ps stderr: ' + data);
});

ps.on('close', function (code) {
  if (code !== 0) {
    console.log('ps process exited with code ' + code);
  }
});

grep.stdout.on('data', function (data) {
  console.log('' + data);
});

grep.stderr.on('data', function (data) {
  console.log('grep stderr: ' + data);
});

grep.on('close', function (code) {
  if (code !== 0) {
    console.log('grep process exited with code ' + code);
  }
});
```

Now the output isn't buffered, and outputs immediately.

The `child_process.spawn()` command does not run the command in a shell by default. However, beginning with Node 5.7.0 and higher, you can specify the `shell` option, and the child process will then generate a shell for the process. Later, in "Running a Child Process Application in Windows" on page 176, I'll demonstrate how this works within a Node application designed to work in Windows. There are also other options, including the following (which is not a complete list; check Node documentation for a comprehensive and up-to-date list):

- `cwd`: change working directory
- `env`: an array of environment key/value pairs
- `detached`: prepare the child to run separate from the parent
- `stdio`: array of child's `stdio` options

The `detached` option, like `shell`, has interesting differences in a Windows environment compared to a non-Windows environment. Again, we'll look into the option in section "Running a Child Process Application in Windows" on page 176.

The `child_process.spawnSync()` is a synchronous version of the same function.

child_process.exec and child_process.execFile

In addition to spawning a child process, you can also use `child_process.exec()` and `child_process.execFile()` to run a command.

The `child_process.exec()` method is similar to `child_process.spawn()` except that `spawn()` starts returning a stream as soon as the program executes, as noted in Example 8-1. The `child_process.exec()` function, like `child_process.execFile()`, buffers the results. However, `exec()` spawns a shell to process the application, unlike `child_process.execFile()`, which directly runs the process. This makes `child_process.execFile()` more efficient than either `child_process.spawn()` with the `shell` option set or `child_process.exec()`.

The first parameter in `child_process.exec()` or `child_process.execFile()` is either the command (for `exec()`) or the file and its location (`execFile()`); the second parameter is options for the command; and the third is a callback function. The callback function takes three arguments: `error`, `stdout`, and `stderr`. The data is buffered to `stdout` if no error occurs.

If the executable file contains:

```
#!/usr/bin/node

console.log(global);
```

the following application prints out the buffered results:

```
var execfile = require('child_process').execFile,
    child;

child = execfile('./app', function(error, stdout, stderr) {
  if (error == null) {
    console.log('stdout: ' + stdout);
  }
});
```

which could also be accomplished using `child_process.exec()`:

```
var exec = require('child_process').exec,
    child;

child = exec('./app', function(error, stdout, stderr) {
   if (error) return console.error(error);
   console.log('stdout: ' + stdout);

});
```

The difference is that `child_process.exec()` spawns a shell, whereas the `child_process.execFile()` function does not.

The `child_process.exec()` function takes three parameters: the command, an `options` object, and a callback. The `options` object takes several values, including `encoding` and the `uid` (user id) and `gid` (group identity) of the process. In Chapter 6, I created an application that copies a PNG file and adds a Polaroid effect. It uses a child process (spawn) to access ImageMagick, a powerful command-line graphics tool. To run it using `child_process.exec()`, use the following, which incorporates a command-line argument:

```
var exec = require('child_process').exec,
    child;

child = exec('./polaroid -s phoenix5a.png -f phoenixpolaroid.png',
         {cwd: 'snaps'}, function(error, stdout, stderr) {
                if (error) return console.error(error);
                console.log('stdout: ' + stdout);
});
```

The `child_process.execFile()` has an additional parameter: an array of command-line options to pass to the application. The equivalent application using this function is:

```
var execfile = require('child_process').execFile,
    child;

child = execfile('./snapshot',
                 ['-s', 'phoenix5a.png', '-f', 'phoenixpolaroid.png'],
                  {cwd: 'snaps'}, function(error, stdout, stderr) {
   if (error) return console.error(error);
   console.log('stdout: ' + stdout);
});
```

Note that the command-line arguments are separated into different array elements with the value for each argument following the argument.

Because `child_process.execFile()` does not spawn a shell it can't be used in some circumstances. Node documentation notes you can't use I/O redirection and *file globbing* using pathname expansion (via regular expression or wildcard). However, if you're attempting to run a child process (or application) interactively, then use

`child_process.execFile()` rather than `child_process.exec()`. The following code, created by Node Foundation member Colin Ihrig, demonstrates this nicely:

```
'use strict';
const cp = require('child_process');
const child = cp.execFile('node', ['-i'], (err, stdout, stderr) => {
  console.log(stdout);
});

child.stdin.write('process.versions;\n');
child.stdin.end();
```

The application starts up an interactive Node session, asks for the process versions, and then ends the input.

There are synchronous versions—`child_process.execSync()` and `child_process.execFileSync()`—of both functions.

child_process.fork

The last child process method is `child_process.fork()`. This variation of `spawn()` is for spawning Node processes.

What sets the `child_process.fork()` process apart from the others is that there's an actual communication channel established to the child process. Note, though, that each process requires a whole new instance of V8, which takes both time and memory.

One use of `child_process.fork()` is to spin off functionality to completely separate Node instances. Let's say you have a server on one Node instance, and you want to improve performance by integrating a second Node instance for answering server requests. The Node documentation features just such an example using a TCP server. Could it also be used to create parallel HTTP servers? Yes, and by using a similar approach.

I want to thank Jiale Hu for giving me the idea when I saw his demonstration of parallel HTTP servers in separate instances. Jiale uses a TCP server to pass on the socket endpoints to two separate child HTTP servers.

Similar to what is demonstrated in the Node documentation for master/child parallel TCP servers, in the master in my example, I create the HTTP server and then use the `child_process.send()` function to send the server to the child process.

```
var cp = require('child_process'),
    cp1 = cp.fork('child2.js'),
    http = require('http');
```

```
var server = http.createServer();

server.on('request', function (req, res) {
   res.writeHead(200, {'Content-Type': 'text/plain'});
   res.end('handled by parent\n');
});

server.on('listening', function () {
    cp1.send('server', server);
});

server.listen(3000);
```

The child process receives the message with the HTTP server via the `process` object. It listens for the connection event and, when it receives it, triggers the connection event on the child HTTP server, passing to it the socket that forms the connection endpoint.

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

var server = http.createServer(function (req, res) {
   res.writeHead(200, {'Content-Type': 'text/plain'});
   res.end('handled by child\n');
});

process.on('message', function (msg, httpServer) {
   if (msg === 'server') {
      httpServer.on('connection', function (socket) {
          server.emit('connection', socket);
      });
   }
});
```

If you test the application by accessing the domain with the 3000 port, you'll see that sometimes the parent HTTP server handles the request, and sometimes the child server does. If you check for running processes, you'll see two: one for the parent, and one for the child.

Node Cluster

The Node Cluster module is based on `child_process.fork()`, in addition to other functionality.

Running a Child Process Application in Windows

Earlier I warned you that child processes that invoke Unix system commands won't work with Windows, and vice versa. I know this sounds obvious, but not everyone

knows that, unlike with JavaScript in browsers, Node applications can behave differently in different environments.

Aside from operating system and command differences, when working in Windows, you either need to use `child_process.exec()`—which spawns a shell in order to run the application—or use the shell option with newer versions of `child_process.spawn()`. Otherwise, you'll need to invoke whatever command you want to run via the Windows command interpreter `cmd.exe`.

As an example of using the shell option with `child_process.spawn()`, the following application prints out the directory contents in a Windows machine:

```
var spawn = require('child_process').spawn,
    pwd = spawn('echo', ['%cd%'], {shell: true});

pwd.stdout.on('data', function (data) {
  console.log('stdout: ' + data);
});

pwd.stderr.on('data', function (data) {
  console.log('stderr: ' + data);
});

pwd.on('close', function (code) {
  console.log('child process exited with code ' + code);
});
```

The `echo` command echos out the result of the Windows `cd` command, which prints out the current directory. If I didn't set the `shell` option to `true`, it would have failed.

A similar result can be obtained with `child_process.exec()`. Note, though, that I didn't need to use the `echo` command with `child_process.exec()`, as the output is buffered with the latter function:

```
var exec = require('child_process').exec,
    pwd = exec('cd');

pwd.stdout.on('data', function (data) {
  console.log('stdout: ' + data);
});

pwd.stderr.on('data', function (data) {
  console.log('stderr: ' + data);
});

pwd.on('close', function (code) {
  console.log('child process exited with code ' + code);
});
```

Example 8-2 demonstrates the third option: running a Windows command using the approach where the command we're running is `cmd`, for the Windows `cmd.exe` appli-

cation; whatever follows in the argument is what's executed in the command shell. In the application, Windows `cmd.exe` is used to create a directory listing, which is then printed out to the console via the data event handler.

Example 8-2. Running a child process application in Windows

```
var cmd = require('child_process').spawn('cmd', ['/c', 'dir\n']);

cmd.stdout.on('data', function (data) {
    console.log('stdout: ' + data);
});

cmd.stderr.on('data', function (data) {
    console.log('stderr: ' + data);
});

cmd.on('exit', function (code) {
    console.log('child process exited with code ' + code);
});
```

The `/c` flag passed as the first argument to `cmd.exe` instructs it to carry out the command and then terminate. The application doesn't work without this flag. You especially don't want to pass in the `/K` flag, which tells `cmd.exe` to execute the application and then remain, because then your application won't terminate.

The equivalent using `child_process.exec()` is:

```
    var cmd = require('child_process').exec('dir');
```

We can run a cmd or bat file using `child_process.execFile()`, just as we can run a file in a Unix-like environment. Given the following bat file (*my.bat*):

```
    @echo off
    REM Next command generates a list of program files
    dir
```

run the file with the following application:

```
    var execfile = require('child_process').execFile,
        child;

    child = execfile('my.bat', function(error, stdout, stderr) {
      if (error == null) {
        console.log('stdout: ' + stdout);
      }
    });
```

CHAPTER 9

Node and ES6

Most of the examples in the book use JavaScript that has been widely available for many years. And it's perfectly acceptable code, as well as very familiar to people who have been coding with the language in a browser environment. One of the advantages to developing in a Node environment, though, is you can use more modern JavaScript, such as ECMAScript 2015 (or ES6, as most people call it), and even later versions, and not have to worry about compatibility with browser or operating system. Many of the new language additions are an inherent part of the Node functionality.

In this chapter, we're going to take a look at some of the newer JavaScript capabilities that are implemented, by default, with the versions of Node we're covering in this book. We'll look at how they can improve a Node application, and we'll look at the gotchas we need to be aware of when we use the newer functionality.

List of Supported ES6 Functionality

I'm not covering all the ES6 functionality supported in Node—just the currently implemented bits that I've seen frequently used in Node applications, modules, and examples. For a list of E6 shipped features, see the Node documentation.

Strict Mode

JavaScript strict mode has been around since ECMAScript 5, but its use directly impacts on the use of ES6 functionality, so I want to take a closer look at it before diving into the ES6 features.

Strict mode is turned on when the following is added to the top of the Node application:

```
"use strict";
```

You can use single or double quotes.

There are other ways to force strict mode on all of the application's dependent modules, such as the `--strict_mode` flag, but I recommend against this. Forcing strict mode on a module is likely to generate errors or have unforeseen consequences. Just use it in your application or modules, where you control your code.

Strict mode has significant impacts on your code, including throwing errors if you don't define a variable before using it, a function parameter can only be declared once, you can't use a variable in an `eval` expression on the same level as the `eval` call, and so on. But the one I want to specifically focus on in this section is that you can't use *octal literals* in strict mode.

In previous chapters, when setting the permissions for a file, you can't use an octal literal for the permission:

```
"use strict";

var fs = require('fs');

fs.open('./new.txt','a+',0666, function(err, fd) {
     if (err) return console.error(err);
     fs.write(fd, 'First line', 'utf-8', function(err,written, str) {
        if (err) return console.error(err);
        var buf = new Buffer(written);
        fs.read(fd, buf, 0, written, 0, function (err, bytes, buffer) {
           if (err) return console.error(err);
           console.log(buf.toString('utf8'));
        });
      });
});
```

In strict mode, this code generates a syntax error:

```
fs.open('./new.txt','a+',0666, function(err, fd) {
                         ^^^^

SyntaxError: Octal literals are not allowed in strict mode.
```

You can convert the octal literal to a safe format by replacing the leading zero with `0o` —a zero following by a lowercase *o*. The strict mode application works if you set the file permission using `0o666` rather than `0666`:

```
fs.open('./new.txt','a+',0o666, function(err, fd) {
```

The `fs.open()` function also accepts the octal value as a string:

```
fs.open('./new.txt','a+','0666', function(err, fd) {
```

But this syntax is frowned on, so use the previously mentioned format.

Strict mode is also necessary in order to use some ES6 extensions. If you want to use ES6 classes, discussed later in the chapter, you must use strict mode. If you also want to use `let`, discussed next, you must use strict mode.

Discussing Octal Literals

If you're interested in exploring the roots for the octal literal conversion, I recommend reading an ES Discuss thread on the subject, "Octal literals have their uses (you Unix haters skip this one)".

let and const

A limitation with JavaScript applications in the past is the inability to declare a variable at the block level. One of the most welcome additions to ES6 has to be the `let` statement. Using it, we can now declare a variable within a block, and its scope is restricted to that block. When we use `var`, the value of 100 is printed out in the following:

```
if (true) {
  var sum = 100;
}

console.log(sum); // prints out 100
```

When you use `let`:

```
"use strict";

if (true) {
  let sum = 100;
}

console.log(sum);
```

you get a completely different result:

```
ReferenceError: sum is not defined
```

The application must be in strict mode for us to use `let`.

Aside from block-level scoping, `let` differs from `var` in that variables declared with `var` are *hoisted* to the top of the execution scope before any statements are executed. The following results in `undefined` being printed out in the console, but no runtime error occurs:

```
console.log(test);
var test;
```

whereas the following code using `let` results in a runtime `ReferenceError` stating that `test` isn't defined.

```
"use strict";

console.log(test);
let test;
```

Should you always use `let`? Some programmers say yes; others, no. You can also use both and limit the use of `var` for those variables that need application or function-level scope, and save `let` for block-level scope only. Chances are, though, the coding practices established for your organization will define what you use. Or if you know that your environment supports it, use `let` and embrace the E6 side.

Moving on from `let`, the `const` statement declares a read-only value reference. If the value is a primitive, it is immutable. If the value is an object, you can't assign a new object or primitive, but you can change object properties.

In the following, if you try to assign a new value to a `const`, the assignment silently fails:

```
const MY_VAL = 10;

MY_VAL = 100;

console.log(MY_VAL); // prints 10
```

It's important to note that `const` is a value reference. If you assign an array or object to a `const`, you can change object/array members:

```
const test = ['one','two','three'];

const test2 = {apples : 1, peaches: 2};

test = test2; //fails

test[0] = test2.peaches;

test2.apples = test[2];

console.log(test); // [ 2, 'two', 'three' ]
console.log(test2); // { apples: 'three', peaches: 2 }
```

Unfortunately, there's been a significant amount of confusion about `const` because of the differing behaviors between primitive and object values, and the actual name itself, which seems to imply a *constant* (static) assignment. However, if immutability is your ultimate aim and you're assigning an object to the `const`, you can use `object.freeze()` on the object to provide at least shallow immutability.

I have noticed that the Node documentation shows the use of `const` when importing modules. While assigning an object to a `const` can't prevent its properties from being modified, it can imply a level of semantics that tells another coder, at a glance, that this item won't be reassigned a new value later.

More on const and Lack of Immutability

Mathias Bynens, a web standards proponent, has a good in-depth discussion on `const` and immutability.

Like `let`, `const` has also block-level scope. Unlike `let`, it doesn't require strict mode.

You can use `let` and `const` in your applications for the same reason you'd use them in the browser, but I haven't found any additional benefit specific to Node. Some folks report better performance with `let` and `const`, whereas others have actually noted a performance decrease. I haven't found a change at all, but your experience could vary. As I stated earlier, chances are your development team will have requirements about which to use and you should defer to these.

Arrow Functions

If you look at the Node API documentation, the most frequently used ES6 enhancement has to be *arrow functions*. Arrow functions do two things. First, they provide a simplified syntax. For instance, in previous chapters, I used the following to create a new HTTP server:

```
http.createServer(function (req, res) {

   res.writeHead(200);
   res.write('Hello');
   res.end();

}).listen(8124);
```

Using an arrow function, I can rewrite this to:

```
http.createServer( (req, res) => {

   res.writeHead(200);
   res.write('Hello');
   res.end();

}).listen(8124);
```

The `function` keyword is removed and the *fat arrow* (=>) is used to represent the existence of the anonymous function, passing in the given parameters. The simplification can be extended further. For example, the following very familiar function pattern:

```
var decArray = [23, 255, 122, 5, 16, 99];
var hexArray = decArray.map(function(element) {
   return element.toString(16);
});
```

```
console.log(hexArray); // ["17", "ff", "7a", "5", "10", "63"]
```

is simplified to:

```
var decArray = [23, 255, 122, 5, 16, 99];
var hexArray = decArray.map(element => element.toString(16));

console.log(hexArray); // ["17", "ff", "7a", "5", "10", "63"]
```

The curly brackets, the `return` statement, and the `function` keyword are all removed, and the functionality is stripped to its minimum.

Arrow functions aren't only syntax simplifications; they also redefine how `this` is defined. In JavaScript, before arrow functions, every function defined its own value for `this`. So in the following example code, instead of my name printing out to the console, I get an `undefined`:

```
function NewObj(name) {
   this.name = name;
}

NewObj.prototype.doLater = function() {
   setTimeout(function() {
      console.log(this.name);
   }, 1000);
};

var obj = new NewObj('shelley');
obj.doLater();
```

The reason is `this` is defined to be the object in the object constructor but the `setTimeout` function in the later instance. We get around the problem by using another variable, typically `self`, which could be *closed over*—attached to the given environment. The following results in expected behavior, with my name printing out:

```
function NewObj(name) {
   this.name = name;
}

NewObj.prototype.doLater = function() {
   var self = this;
   setTimeout(function() {
      console.log(self.name);
   }, 1000);
};

var obj = new NewObj('shelley');
obj.doLater();
```

In an arrow function, `this` is always set to the value it would normally have within the enclosing context; in this case, the `new` object:

```
function NewObj(name) {
   this.name = name;
}

NewObj.prototype.doLater = function() {
   setTimeout(()=> {
      console.log(this.name);
   }, 1000);
};

var obj = new NewObj('shelley');
obj.doLater();
```

Working Around the Arrow Function Quirks

The arrow function does have quirks, such as how do you return an empty object, or where are the arguments? Strongloop has a nice writeup on the arrow functions that discusses the quirks and the workarounds.

Classes

JavaScript has now joined its older siblings in support for classes. No more interesting twists to emulate the more familiar class behavior.

Earlier, in Chapter 3, I created a "class," `InputChecker`, using the older syntax:

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

exports.InputChecker = InputChecker;

function InputChecker(name, file) {
   this.name = name;
   this.writeStream = fs.createWriteStream('./' + file + '.txt',
                      {'flags' : 'a',
                       'encoding' : 'utf8',
                       'mode' : 0666});
};

util.inherits(InputChecker,eventEmitter);

InputChecker.prototype.check = function check(input) {
   var command = input.toString().trim().substr(0,3);
   if (command == 'wr:') {
      this.emit('write',input.substr(3,input.length));
   } else if (command == 'en:') {
      this.emit('end');
   } else {
      this.emit('echo',input);
```

```
    }
};
```

I modified it slightly to embed the `check()` function directly into the object definition. I then converted the result into an ES6 class, starting with using strict mode, which is required for using the new class functionality.

Rather than having to use `util.inherits()` to inherit the prototype methods from the superconstructor to the constructor, the new class extends the `EventEmitter` object's methods. I added the constructor method for creating and initializing the new object. In this constructor, I called `super()` to invoke the functions of the parent class.

```
'use strict';

const util = require('util');
const eventEmitter = require('events').EventEmitter;
const fs = require('fs');

class InputChecker extends eventEmitter {

   constructor(name, file) {
      super()
      this.name = name;

      this.writeStream = fs.createWriteStream('./' + file + '.txt',
         {'flags' : 'a',
          'encoding' : 'utf8',
          'mode' : 0o666});
      }

   check (input) {
      let command = input.toString().trim().substr(0,3);
      if (command == 'wr:') {
        this.emit('write',input.substr(3,input.length));
      } else if (command == 'en:') {
        this.emit('end');
      } else {
        this.emit('echo',input);
      }
   }
};

exports.InputChecker = InputChecker;
```

The `check()` method isn't added using the prototype object but is just included in the class as a method. There's no use of var or function with `check()`—you just define the method. However, all the working logic is the same or very close to the original object.

The application that uses the new *classified* (sorry, pun) module is unchanged:

```
var InputChecker = require('./class').InputChecker;

// testing new object and event handling
var ic = new InputChecker('Shelley','output');

ic.on('write', function (data) {
   this.writeStream.write(data, 'utf8');
});
ic.addListener('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});

ic.on('end', function() {
   process.exit();
});

process.stdin.setEncoding('utf8');
process.stdin.on('data', function(input) {
    ic.check(input);
});
```

As the code demonstrates, I did not have to place the application into strict mode in order to use a module that's defined in strict mode.

Mozilla's Always Helpful Documentation

The Mozilla Developer Network is always a first-go source on anything JavaScript, and the new class functionality is no exception. They provide a very nice reference for this new functionality.

Promises with Bluebird

During the early stages of Node development, the creators debated whether to go with callbacks or promises. Callbacks won, which pleased some folks and disappointed others.

Promises are now part of ES6, and you can certainly use them in your Node applications. However, if you want to use ES6 promises with the Node core functionality, you'll either need to implement the support from scratch or use a module to provide this support. Though I've tried to avoid using third-party modules as much as possible in this book, in this case I recommend using the module. In this section, we'll look at using a very popular promises module: Bluebird.

Performance Enhanced with Bluebird

Another reason to use Bluebird is performance. The author of Bluebird explains why in StackExchange.

Rather than nested callbacks, ES6 promises feature branching, with success handled by one branch, failure by another. The best way to demonstrate it is by taking a typical Node filesystem application and then *promisfying* it—converting the callback structure to promises.

Using native callbacks, the following application opens a file and reads in the contents, makes a modification, and then writes it to another file.

```
var fs = require('fs');
fs.readFile('./apples.txt','utf8', function(err,data) {
   if (err) {
      console.error(err.stack);
   } else {

     var adjData = data.replace(/apple/g,'orange');

     fs.writeFile('./oranges.txt', adjData, function(err) {

        if (err) console.error(err);
     });
   }
});
```

Even this simple example shows nesting two levels deep: reading the file in and then writing the modified content.

Now, we'll use Bluebird to promisify the example.

In the code I'm using, the Bluebird `promisifyAll()` function is used to promisify all of the File System functions. Instead of `readFile()`, we'll then use `readFileAsync()`, which is the version of the function that supports promises.

```
var promise = require('bluebird');
var fs = promise.promisifyAll(require('fs'));

fs.readFileAsync('./apples.txt','utf8')
   .then(function(data) {
      var adjData = data.replace(/apple/g, 'orange');
      return fs.writeFileAsync('./oranges.txt', adjData);
   })
   .catch(function(error) {
      console.error(error);
   });
```

In the example, when the file contents are read, a successful data operation is handled with the `then()` function. If the operation isn't successful, the `catch()` function handles the error. If the read is successful, the data is modified and the promisify version of `writeFile()`, `writeFileAsync()`, is called, writing the data to the file. From the previous nested callback example, we know that `writeFile()` just returns an error. This error would also be handled by the single `catch()` function.

Though the nested example isn't large, you can see how much clearer the promise version of the code is. You can also start to see how the nested problem is resolved—especially with only one error-handling routine necessary for any number of calls.

What about a more complex example? I modified the previous code to add an additional step to create a new subdirectory to contain the *oranges.txt* file. In the code, you can see that there are now two `then()` functions. The first processes the successful response to making the subdirectory, and the second creates the new file with the modified data. The new directory is made using the promisifyed `mkdirAsync()` function, which is returned at the end of the process. This is the key to making the promises work, because the next `then()` function is actually attached to the returned function. The modified data is still passed to the promise function where the data is being written. Any errors in either the read file or directory-making process are handled by the single `catch()`.

```
var promise = require('bluebird');
var fs = promise.promisifyAll(require('fs'));

fs.readFileAsync('./apples.txt','utf8')
   .then(function(data) {
      var adjData = data.replace(/apple/g, 'orange');
      return fs.mkdirAsync('./fruit/');
   })
   .then(function(adjData) {
      return fs.writeFileAsync('./fruit/oranges.txt', adjData);
   })
   .catch(function(error) {
      console.error(error);
   });
```

How about handling instances where an array of results is returned, such as when we're using the File System function `readdir()` to get the contents of a directory?

That's where array-handling functions such as `map()` come in handy. In the following code, the contents of a directory are returned as an array, and each file in that directory is opened, its contents modified and written to a comparably named file in another directory. The inner `catch()` function handles errors for reading and writing files, while the outer one handles the directory access.

```
var promise = require('bluebird');
var fs = promise.promisifyAll(require('fs'));

fs.readdirAsync('./apples/').map(filename => {
   fs.readFileAsync('./apples/'+filename,'utf8')
      .then(function(data) {
         var adjData = data.replace(/apple/g, 'orange');
         return fs.writeFileAsync('./oranges/'+filename, adjData);
      })
      .catch(function(error) {
         console.error(error);
      })
   })
   .catch(function(error) {
      console.error(error);
   })
```

I've only touched on the capability of Bluebird and the very real attraction of using promises in Node. Do take some time to explore the use of both, in addition to the other ES6 features, in your Node applications.

CHAPTER 10

Full-Stack Node Development

Most of this book focuses on the core modules and functionality that make up Node. I've tried to avoid covering third-party modules because Node is still a very dynamic environment and support for the third-party modules can change quickly and drastically.

But I don't think you can cover Node without at least briefly mentioning the wider context of Node applications, which means you need to be familiar with full-stack Node development. This means being familiar with data systems, APIs, client-side development—a whole range of technologies with only one commonality: Node.

The most common form of full-stack development with Node is MEAN—MongoDB, Express, AngularJS, and Node. However, full-stack development can encompass other tools, such as MySQL or Redis for database development, and other client-side frameworks in addition to AngularJS. The use of Express, though, has become ubiquitous. You have to become familiar with Express if you're going to work with Node.

Further Explorations of MEAN

For additional explorations of MEAN, full-stack development, and Express, I recommend *Web Development with Node and Express: Leveraging the JavaScript Stack* (O'Reilly, 2014), by Ethan Brown; *AngularJS: Up and Running* (O'Reilly, 2014), by Shyam Seshadri and Brad Green; and the video *Architecture of the MEAN Stack* (O'Reilly, 2015), by Scott Davis.

The Express Application Framework

In Chapter 5, I covered a small subset of the functionality you need to implement a Node web application. The task to create a Node web app is daunting at best. That's why an application framework like Express has become so popular: it provides most of the functionality with minimal effort.

Express is almost ubiquitous in the Node world, so you'll need to become familiar with it. We're going to look at the most bare-bones Express application we can in this chapter, but you will need additional training once you're finished.

Express Now Part of Node.js Foundation

Express has had a rocky start but is now part of the Node.js Foundation. Future development should be more consistent and its support more reliable.

Express provides good documentation, including how to start an application. We'll follow the steps the documentation outlines and then expand on the basic application. To start, create a new subdirectory for the application and name it whatever you want. Use npm to create a *package.json* file, using *app.js* as the entry point for the application. Lastly, install Express, saving it to your dependencies in the *package.json* file by using the following command:

```
npm install express --save
```

The Express documentation contains a minimal Hello World Express application, typed into the *app.js* file:

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

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

The `app.get()` function handles all GET web requests, passing in the request and response objects we're familiar with from our work in earlier chapters. By convention, Express applications use the abbreviated forms of `req` and `res`. These objects have the same functionality as the default request and response objects, with the addition of new functionality provided by Express. For instance, you can use `res.write()` and

`res.end()` to respond to the web request, which we've used in past chapters, but you can also use `res.send()`, an Express enhancement, to do the same in one line.

Instead of manually creating the application, we can also use the Express application generator to generate the application skeleton. We'll use that next, as it provides a more detailed and comprehensive Express application.

First, install the Express application generator globally:

```
sudo npm install express-generator -g
```

Next, run the application with the name you want to call your application. For demonstration purposes, I'll use `bookapp`:

```
express bookapp
```

The Express application generator creates the necessary subdirectories. Change into the *bookapp* subdirectory and install the dependencies:

```
npm install
```

That's it, you've created your first skeleton Express application. You can run it using the following if you're using an OS X or Linux environment:

```
DEBUG=bookapp:* npm start
```

Run the following if you're in a Windows Command window:

```
set DEBUG=bookapp:* & npm start
```

You could also start the application with just `npm start` and forgo the debugging.

The application is started and listens for requests on the default Express port of 3000. Accessing the application via the Web returns a simple web page with the "Welcome to Express" greeting.

Several subdirectories and files are generated by the application:

```
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.jade
    ├── index.jade
    └── layout.jade
```

We'll get into more detail about many of the components, but the public-facing static files are located in the *public* subdirectory. As you'll note, the graphics and CSS files are placed in this location. The dynamic content template files are located in *views*. The *routes* subdirectory contains the web endpoint applications that listen for web requests and render web pages.

Jade is Now Pug

Because of trademark violation, the creators of Jade can no longer use "Jade" as the name for the template engine used with Express and other applications. However, the process of converting Jade to Pug is still ongoing. At the time this went to production, the Express Generator still generates Jade, but attempting to install Jade as a dependency generates the following error:

```
Jade has been renamed to pug, please install the latest
version of pug instead of jade
```

The web site for Pug is still named Jade, but the documentation and functionality remains the same. Hopefully this will all resolve itself sooner, rather than later.

The *www* file in the *bin* subdirectory is a startup script for the application. It's a Node file that's converted into a command-line application. When you look in the generated *package.json* file, you'll see it listed as the application's start script.

```
{
  "name": "bookapp",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "debug": "~2.2.0",
    "express": "~4.13.1",
    "jade": "~1.11.0",
    "morgan": "~1.6.1",
    "serve-favicon": "~2.3.0"
  }
}
```

You install other scripts to test, restart, or otherwise control your application in the *bin* subdirectory.

To begin a more in-depth view of the application we'll look at the application's entry point, the *app.js* file.

When you open the *app.js* file, you're going to see considerably more code than the simple application we looked at earlier. There are several more modules imported, most of which provide the *middleware* support we'd expect for a web-facing application. The imported modules also include application-specific imports, given under the *routes* subdirectory:

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var routes = require('./routes/index');
var users = require('./routes/users');

var app = express();
```

The modules and their purposes are:

`express`
: The Express application

`path`
: Node core module for working with file paths

`serve-favicon`
: Middleware to serve the *favicon.ico* file from a given path or buffer

`morgon`
: An HTTP request logger

`cookie-parser`
: Parses cookie header and populates `req.cookies`

`body-parser`
: Provides four different types of request body parsers (but does not handle multipart bodies)

Each of the middleware modules works with a vanilla HTTP server as well as Express.

What Is Middleware?

Middleware is the intermediary between the system/operating system/database and the application. With Express, the middleware is part of a chain of applications, each of which does a certain function related to an HTTP request—either processing it, or performing some manipulation on the request for future middleware applications. The set of middleware that works with Express is quite comprehensive.

The next section of code in *app.js* mounts the middleware (makes it available in the application) at a given path via the `app.use()` function. The order in which the middleware is mounted is important so if you add additional middleware functionality, be sure that it is in relation to other middleware according to the developer recommendations.

The code snippet also includes code that defines the view engine setup, which I'll get to in a moment.

```
// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

The last call to `app.use()` references one of the few built-in Express middleware, `express.static`, which is used to handle all static files. If a web user requests an HTML, JPEG, or other static file, `express.static` processes the request. All static files are served relative to the path specified when the middleware is mounted, in this case, in the `public` subdirectory.

Returning to the `app.set()` function calls defining the views engine in the code snippet, you'll use a *template engine* that helps map the data to the delivery. One of the most popular, Jade, is integrated by default, but others such as Mustache and EJS can be used just as easily. The engine setup defines the subdirectory where the template files (*views*) are located and which view engine to use (Jade).

Reminder: Jade is Now Pug

As mentioned earlier in the chapter, Jade is now Pug. Check with both the Express documentation, and Pug, in order to update examples to work with the newly named template engine

As this book went to production, I modified the generated package.json file to replace the Jade module with Pug:

```
<p>"pug": "2.0.0-alpha8",</p>
```

And in the `app.js` file, replace the `jade` reference with `pug`:

```
app.set('view engine', 'pug');
```

And the application seemed to work without problem.

In the *views* subdirectory, you'll find three files: *error.jade*, *index.jade*, and *layout.jade*. These will get you started, though you'll need to provide much more when you start integrating data into the application. The content for the generated *index.jade* file is given here:

```
extends layout

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

The line that reads `extends layout` incorporates the Jade syntax from the *layout.jade* file. You'll recognize the HTML header (`h1`) and paragraph (`p`) elements. The `h1` header is assigned the value passed to the template as `title`, which is also used in the paragraph element. How these values get rendered in the template requires us to return to the *app.js* file for the next bit of code:

```
app.use('/', routes);
app.use('/users', users);
```

These are the application-specific endpoints, which is the functionality that responds to client requests. The top-level request (`'/'`) is satisfied by the *index.js* file in the *routes* subdirectory, the users, by the *users.js* file.

In the *index.js* file, we're introduced to the Express *router*, which provides the response-handling functionality. As the Express documentation notes, the router behavior fits the following pattern:

```
app.METHOD(PATH, HANDLER)
```

The method is the HTTP method, and Express supports several, including the familiar `get`, `post`, `put`, and `delete`, as well as the possibly less familiar `merge`, `search`, `head`, `options`, and so on. That path is the web path, and the handler is the function that processes the request. In *index.js*, the method is `get`, the path is the application root, and the handler is a callback function passing request and response:

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

The data (local variables) and view meet in the `res.render()` function call. The view used is the *index.jade* file we looked at earlier, and you can see that the value for the `title` attribute in the template is passed as data to the function. In your copy, try changing "Express" to whatever you'd like and reloading the page to see the modification.

The rest of the *app.js* file is error handling, and I'll leave that for you to explore on your own. This is a quick and very abbreviated introduction to Express, but hopefully even with this simple example, you're getting a feel for the structure of an Express application.

Incorporating Data

I'll toot my horn and recommend my book, the *JavaScript Cookbook* (O'Reilly, 2010) if you want to learn more about incorporating data into an Express application. Chapter 14 demonstrates extending an existing Express application to incorporate a MongoDB store, as well as incorporating the use of controllers, for a full model-view-controller (MVC) architecture.

MongoDB and Redis Database Systems

In Chapter 7, Example 7-8 featured an application that inserted data into a MySQL database. Though sketchy at first, support for relational databases in Node applications has grown stronger, with robust solutions such as the MySQL driver for Node, and newer modules such as the Tedious package for SQL Server access in a Microsoft Azure environment.

Node applications also have access to several other database systems. In this section I'm going to briefly look at two: MongoDB, which is very popular in Node development, and Redis, a personal favorite of mine.

MongoDB

The most popular database used in Node applications is MongoDB. MongoDB is a document-based database. The documents are encoded as BSON, a binary form of JSON, which probably explains its popularity among JavaScript developers. With MongoDB, instead of a table row, you have a BSON document; instead of a table, you have a collection.

MongoDB isn't the only document-centric database. Other popular versions of this type of data store are CouchDB by Apache, SimpleDB by Amazon, RavenDB, and even the legendary Lotus Notes. There is some Node support of varying degrees for most modern document data stores, but MongoDB and CouchDB have the most.

MongoDB is not a trivial database system, and you'll need to spend time learning its functionality before incorporating it into your Node applications. When you're ready, though, you'll find excellent support for MongoDB in Node via the MongoDB Native NodeJS Driver and additional, object-oriented support with Mongoose.

Though I'm not going to get into detail on how to use MongoDB with Node, I am going to provide one example, just so you can get an idea of how it works. Though the underlying data structure differs from relational databases, the concepts are still the same: you create a database, you create collections of records, and you add individual records. You can then update, query, or delete the records. In the MongoDB example shown in Example 10-1, I'm connecting to an example database, accessing a

Widgets collection, removing any existing records, inserting two, and then querying for the two and printing them out.

Example 10-1. Working with a MongoDB database

```
var MongoClient = require('mongodb').MongoClient;

// Connect to the db
MongoClient.connect("mongodb://localhost:27017/exampleDb",
                                        function(err, db) {
   if(err) { return console.error(err); }

   // access or create widgets collection
   db.collection('widgets', function(err, collection) {
      if (err) return console.error(err);

      // remove all widgets documents
      collection.remove(null,{safe : true}, function(err, result) {
         if (err) return console.error(err);
         console.log('result of remove ' + result.result);

         // create two records
         var widget1 = {title : 'First Great widget',
                        desc : 'greatest widget of all',
                        price : 14.99};
         var widget2 = {title : 'Second Great widget',
                        desc : 'second greatest widget of all',
                        price : 29.99};

         collection.insertOne(widget1, {w:1}, function (err, result) {
            if (err) return console.error(err);
            console.log(result.insertedId);

            collection.insertOne(widget2, {w:1}, function(err, result) {
               if (err) return console.error(err);
               console.log(result.insertedId);

               collection.find({}).toArray(function(err,docs) {
                  console.log('found documents');
                  console.dir(docs);

                  //close database
                  db.close();
               });
            });
         });
      });
   });
});
```

Yes, Node callback hell, but you can incorporate the use of promises.

The MongoClient object is the object you'll use the most for connecting to the database. Note the port number given (27017). This is the default port for the MongoDB system. The database is `exampleDB`, given as part of the connection URL, and the collection is `widgets`, in honor of the class Widget factory known far and wide among developers.

The MongoDB functions are asynchronous, as you'd expect. Before the records are inserted, the application first deletes all existing records in the collection using the `collection.remove()` function with no specific query. If we didn't, we'd have duplicate records, as the MongoDB assigns system-generated unique identifiers for each new record and we're not explicitly making the title or other field a unique identifier.

We add each new record using `collection.insertOne()`, passing in the JSON that defines the object. The option, `{w:1}` specifies *write concern*, describing the level of acknowledgment from MongoDB for the write operation.

Once the records are inserted, the application uses `collection.find()`, again without a specific query, to find all records. The function actually creates a *cursor*, and the `toArray()` function returns the cursor results as an array, which we can then print out to the console using `console.dir()`. The result of the application looks similar to the following:

```
result of remove 1
56c5f535c51f1b8d712b6552
56c5f535c51f1b8d712b6553
found documents
[ { _id: ObjectID { _bsontype: 'ObjectID', id: 'VÅõ5Å\u001f\u001bq+eR' },
    title: 'First Great widget',
    desc: 'greatest widget of all',
    price: 14.99 },
  { _id: ObjectID { _bsontype: 'ObjectID', id: 'VÅõ5Å\u001f\u001bq+eS' },
    title: 'Second Great widget',
    desc: 'second greatest widget of all',
    price: 29.99 } ]
```

The object identifier is actually an object, and the identifier is in BSON, which is why it doesn't cleanly print. If you want to provide a cleaner output, you can access each individual field and covert the BSON identifier into a hexadecimal string with `toHexString()`:

```
docs.forEach(function(doc) {
                    console.log('ID : ' + doc._id.toHexString());
                    console.log('desc : ' + doc.desc);
                    console.log('title : ' + doc.title);
                    console.log('price : ' + doc.price);
                });
```

Now the result is:

```
result of remove 1
56c5fa40d36a4e7b72bfbef2
56c5fa40d36a4e7b72bfbef3
found documents
ID : 56c5fa40d36a4e7b72bfbef2
desc : greatest widget of all
title : First Great widget
price : 14.99
ID : 56c5fa40d36a4e7b72bfbef3
desc : second greatest widget of all
title : Second Great widget
price : 29.99
```

You can see the records in the MongoDB using the command-line tool. Use the following sequence of commands to start it and look at the records:

1. Type **`mongo`** to start the command-line tool.
2. Type **`use exampleDb`** to change to the exampleDb database.
3. Type **`show collections`** to see all collections.
4. Type **`db.widgets.find()`** to see the Widget records.

If you'd prefer a more object-based approach to incorporating MongoDB into your application, you'll want to use Mongoose. It may also be a better fit for integrating into Express.

When you're not playing around with MongoDB, remember to shut it down.

MongoDB in Node Documentation

The MongoDB driver for Node is documented online, and you can access the documentation from its GitHub repository. But you can also access documentation for the driver at the MongoDB site. I prefer the MongoDB site's documentation, especially for those just starting out.

Redis Key/Value Store

When it comes to data, there's relational databases and Everything Else, otherwise known as NoSQL. In the NoSQL category, a type of structured data is based on key/value pairs, typically stored in memory for extremely fast access. The three most popular in-memory key/value stores are Memcached, Cassandra, and Redis. Happily for Node developers, there is Node support for all three stores.

Memcached is primarily used as a way of caching data queries for quick access in memory. It's also quite good with distributed computing, but has limited support for more complex data. It's useful for applications that do a lot of queries but less so for

applications doing a lot of data writing and reading. Redis is the superior data store for the latter type of application. In addition, Redis can be persisted, and it provides more flexibility than Memcached—especially in its support for different types of data. However, unlike Memcached, Redis works only on a single machine.

The same factors also come into play when we compare Redis and Cassandra. Like Memcached, Cassandra has support for clusters. However, also like Memcached, it has limited data structure support. It's good for ad hoc queries—a use that does not favor Redis. However, Redis is simple to use, uncomplicated, and typically faster than Cassandra. For these reasons and others, Redis has gained a greater following among Node developers.

EARN

I was delighted to read the acronym EARN, or Express, AngularJS, Redis, and Node. An example of EARN is covered in "The EARN Stack".

My preferred Node Redis module is installed through npm:

```
npm install redis
```

If you plan on using big operations on Redis, I also recommend installing the Node module support hiredis, as it's non-blocking and can improve performance:

```
npm install hiredis redis
```

The Redis module is a relatively thin wrapper around Redis itself. Thus, you'll need to spend time learning the Redis commands and how the Redis data store works.

To use Redis in your Node applications, you first include the module:

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

Then you'll need to create a Redis client. The method used is `createClient`:

```
var client = redis.createClient();
```

The `createClient` method can take three optional parameters: `port`, `host`, and options (outlined shortly). By default, the host is set to `127.0.0.1`, and the port is set to `6379`. The port is the one used by default for a Redis server, so these default settings should be fine if the Redis server is hosted on the same machine as the Node application.

The third parameter is an object that supports several options, outlined in detail in the modules documentation. Use the default settings until you're more comfortable with Node and Redis.

Once you have a client connection to the Redis data store, you can send commands to the server until you call the `client.quit()` method call, which closes the connection

to the Redis server. If you want to force a closure, you can use the `client.end()` method instead. However, the latter method doesn't wait for all replies to be parsed. The `client.end()` method is a good one to call if your application is stuck or you want to start over.

Issuing Redis commands through the client connection is a fairly intuitive process. All commands are exposed as methods on the client object, and command arguments are passed as parameters. Since this is Node, the last parameter is a callback function, which returns an error and whatever data or reply is given in response to the Redis command.

In the following code, the `client.hset()` method is used to set a *hash* property. In Redis, a hash is a mapping between string fields and values, such as "lastname" representing your last name, "firstname" for your first name, and so on:

```
client.hset("hashid", "propname", "propvalue", function(err, reply) {
   // do something with error or reply
});
```

The `hset` command sets a value, so there's no return data—only the Redis acknowledgment. If you call a method that gets multiple values, such as `client.hvals`, the second parameter in the callback function will be an array—either an array of single strings or an array of objects:

```
client.hvals(obj.member, function (err, replies) {
   if (err) {
      return console.error("error response - " + err);
   }

   console.log(replies.length + " replies:");
   replies.forEach(function (reply, i) {
     console.log("    " + i + ": " + reply);
   });
});
```

Because the Node callback is so ubiquitous, and because so many of the Redis commands are operations that just reply with a confirmation of success, the Redis module provides a `redis.print` method you can pass as the last parameter:

```
client.set("somekey", "somevalue", redis.print);
```

The `redis.print` method prints either the error or the reply to the console and returns.

To demonstrate Redis in Node, I'm creating a *message queue*. A message queue is an application that takes as input some form of communication, which is then stored into a queue. The messages are stored until they're retrieved by the message receiver, when they are popped off the queue and sent to the receiver (either one at a time or in bulk). The communication is asynchronous because the application that stores the

messages doesn't require that the receiver be connected, and the receiver doesn't require that the message-storing application be connected.

Redis is an ideal storage medium for this type of application. As the messages are received by the application that stores them, they're pushed on to the end of the message queue. When the messages are retrieved by the application that receives them, they're popped off the front of the message queue.

Getting in a Little TCP, HTTP, and Child Process Work

The Redis example also incorporates a TCP server (hence working with Node's Net module), an HTTP server, as well as a child process. Chapter 5 covers HTTP, Chapter 7 covers Net, and Chapter 8 covers child processes.

For the message queue demonstration, I created a Node application to access the web logfiles for several different subdomains. The application uses a Node child process and the Unix `tail -f` command to access recent entries for the different logfiles. From these log entries, the application uses two regular expression objects: one to extract the resource accessed, and the second to test whether the resource is an image file. If the accessed resource is an image file, the application sends the resource URL in a TCP message to the message queue application.

All the message queue application does is listen for incoming messages on port 3000 and store whatever is sent into a Redis data store.

The third part of the demonstration application is a web server that listens for requests on port 8124. With each request, it accesses the Redis database and pops off the front entry in the image data store, returning it via the response object. If the Redis database returns a `null` for the image resource, it prints out a message that the application has reached the end of the message queue.

The first part of the application, which processes the web log entries, is shown in Example 10-2. The Unix `tail` command is a way of displaying the last few lines of a text file (or piped data). When used with the `-f` flag, the utility displays a few lines of the file and then sits, listening for new file entries. When one occurs, it returns the new line. The `tail -f` command can be used on several different files at the same time and manages the content by labeling where the data comes from each time it comes from a different source. The application isn't concerned about which access log is generating the latest tail response—it just wants the log entry.

Once the application has the log entry, it performs a couple of regular expression matches on the data to look for image resource access (files with a *.jpg*, *.gif*, *.svg*, or *.png* extension). If a pattern match is found, the application sends the resource URL to the message queue application (a TCP server). The application is simple, so

it's not checking to ensure that the occurrence of the string matches is actually a file extension rather than a string embedded in the filename, such as *this.jpg.html*. You can get false positives. However, it demonstrates Redis, which is all that matters.

Example 10-2. Node application that processes web log entries and sends image resource requests to the message queue

```
var spawn = require('child_process').spawn;
var net = require('net');

var client = new net.Socket();
client.setEncoding('utf8');

// connect to TCP server
client.connect ('3000','examples.burningbird.net', function() {
    console.log('connected to server');
});

// start child process
var logs = spawn('tail', ['-f',
        '/home/main/logs/access.log',
        '/home/tech/logs/access.log',
        '/home/shelleypowers/logs/access.log',
        '/home/green/logs/access.log',
        '/home/puppies/logs/access.log']);

// process child process data
logs.stdout.setEncoding('utf8');
logs.stdout.on('data', function(data) {

   // resource URL
   var re = /GET\s(\S+)\sHTTP/g;

   // graphics test
   var re2 = /\.gif|\.png|\.jpg|\.svg/;

   // extract URL
   var parts = re.exec(data);
   console.log(parts[1]);

   // look for image and if found, store
   var tst = re2.test(parts[1]);
   if (tst) {
      client.write(parts[1]);
   }
});
logs.stderr.on('data', function(data) {
   console.log('stderr: ' + data);
});

logs.on('exit', function(code) {
   console.log('child process exited with code ' + code);
   client.end();
});
```

Typical console log entries for this application are given in the following block of code, with the entries of interest (the image file accesses) in bold:

```
/robots.txt
/weblog
/writings/fiction?page=10
/images/kite.jpg
/node/145
/culture/book-reviews/silkworm
/feed/atom/
/images/visitmologo.jpg
/images/canvas.png
/sites/default/files/paws.png
/feeds/atom.xml
```

Example 10-3 contains the code for the message queue. It's a simple application that starts a TCP server and listens for incoming messages. When it receives a message, it extracts the data from the message and stores it in the Redis database. The application uses the Redis `rpush` command to push the data on the end of the images list (bolded in the code).

Example 10-3. Message queue that takes incoming messages and pushes them onto a Redis list

```
var net = require('net');
var redis = require('redis');

var server = net.createServer(function(conn) {
   console.log('connected');

   // create Redis client
   var client = redis.createClient();

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

   // sixth database is image queue
   client.select(6);
   // listen for incoming data
   conn.on('data', function(data) {
      console.log(data + ' from ' + conn.remoteAddress + ' ' +
        conn.remotePort);

      // store data
      client.rpush('images',data);
   });

}).listen(3000);
server.on('close', function(err) {
   client.quit();
```

```
});

console.log('listening on port 3000');
```

The message queue application console log entries would typically look like the following:

```
listening on port 3000
connected
/images/venus.png from 173.255.206.103 39519
/images/kite.jpg from 173.255.206.103 39519
/images/visitmologo.jpg from 173.255.206.103 39519
/images/canvas.png from 173.255.206.103 39519
/sites/default/files/paws.png from 173.255.206.103 39519
```

The last piece of the message queue demonstration application is the HTTP server that listens on port 8124 for requests, shown in Example 10-4. As the HTTP server receives each request, it accesses the Redis database, pops off the next entry in the images list, and prints out the entry in the response. If there are no more entries in the list (i.e., if Redis returns `null` as a reply), it prints out a message that the message queue is empty.

Example 10-4. HTTP server that pops off messages from the Redis list and returns to the user

```
var redis = require("redis"),
    http = require('http');

var messageServer = http.createServer();

// listen for incoming request
messageServer.on('request', function (req, res) {

   // first filter out icon request
   if (req.url === '/favicon.ico') {
      res.writeHead(200, {'Content-Type': 'image/x-icon'} );
      res.end();
      return;
   }

  // create Redis client
  var client = redis.createClient();


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

  // set database to 6, the image queue
  client.select(6);
```

```
    client.lpop('images', function(err, reply) {
       if(err) {
         return console.error('error response ' + err);
       }

       // if data
       if (reply) {
          res.write(reply + '\n');
       } else {
          res.write('End of queue\n');
       }
       res.end();
    });
    client.quit();

});

messageServer.listen(8124);

console.log('listening on 8124');
```

Accessing the HTTP server application with a web browser returns a URL for the image resource on each request (browser refresh) until the message queue is empty.

The data involved is very simple, and possibly prolific, which is why Redis is so ideally suited for this type of application. It's a fast, uncomplicated data store that doesn't take a great deal of effort in order to incorporate its use into a Node application.

When to Create the Redis Client

In my work with Redis, sometimes I create a Redis client and persist it for the life of the application, whereas other times I create a Redis client and release it as soon as the Redis command is finished. So when is it better to create a persistent Redis connection versus create a connection and release it immediately?

Good question.

To test the impact of the two different approaches, I created a TCP server that listened for requests and stored a simple hash in the Redis database. I then created another application as a TCP client that did nothing more than send an object in a TCP message to the server.

I used the ApacheBench application to run several concurrent iterations of the client and tested how long it took for each run. I ran the first batch with the Redis client connection persisted for the life of the server, and ran the second batch where the client connection was created for each request and immediately released.

What I expected to find was that the application that persisted the client connection was faster, and I was right…to a point. About halfway through the test with the per-

sistent connection, the application slowed down dramatically for a brief period of time and then resumed its relatively fast pace.

Of course, what most likely happened is that the queued requests for the Redis database eventually blocked the Node application, at least temporarily, until the queue was freed up. I didn't run into this same situation when opening and closing the connections with each request because the extra overhead required for this process slowed the application just enough so that it didn't hit the upper end of concurrent users.

AngularJS and Other Full-Stack Frameworks

First of all, framework is a grossly overused term. We see it used for frontend libraries such as jQuery, graphics libraries such as D3, Express, and a host of more modern full-stack applications. In this chapter, when I use "framework," I mean full-stack frameworks such as AngularJS, Ember, and Backbone.

To become familiar with full-stack frameworks, you need to become familiar with the website TodoMVC. This site defines the requirements for a basic type of application, a to-do list, and then invites any and all framework developers to submit implementations of this application. The site also provides a plain-vanilla implementation of the application without using any framework, as well as one implemented in jQuery. The site gives developers a way to contrast and compare how the same functionality is implemented in each framework. This includes all the popular frameworks, not just AngularJS: Backbone.js, Dojo, Ember, React, and so on. It also features applications that incorporate multiple technologies, such as one that utilizes AngularJS, Express, and the Google Cloud Platform.

The To-Do requirements provide a recommended directory and file structure:

```
index.html
package.json
node_modules/
css
└── app.css
js/
├── app.js
├── controllers/
└── models/
readme.md
```

There's nothing esoteric about the structure, and it resembles what we found for the ExpressJS application. But how each framework meets the requirements can be very different, which is why the ToDo application is a great way of learning how each framework works.

To demonstrate, let's look at some of the code for a couple of the frameworks: AngularJS and Backbone.js. I'm not going to replicate much of the code because it's a sure

bet that the code will change by the time you read this. I'll start with AngularJS and focus on the optimized application—the site features several different implementations featuring AngularJS. Figure 10-1 shows the application after three to-do items have been added.

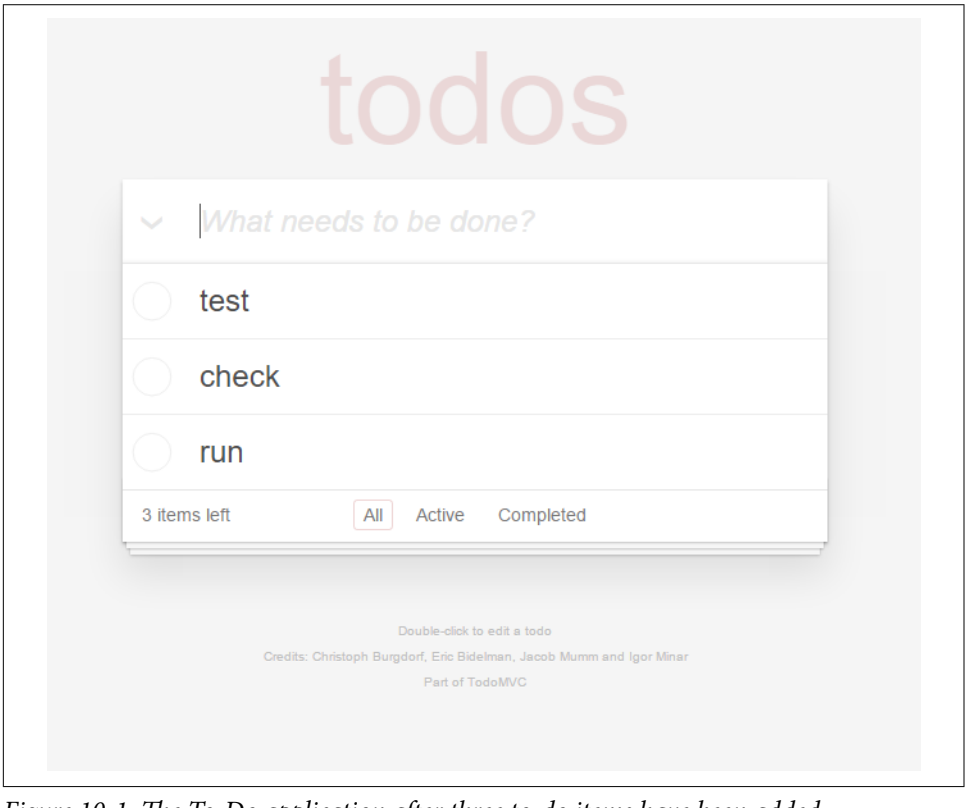


Figure 10-1. The To-Do application after three to-do items have been added

To start, the application root, *app.js*, is very simple, as we'd expect with all the functionality being split to the various model-view-controller subgroups.

```
/* jshint undef: true, unused: true */
/*global angular */
(function () {
'use strict';

/**
   * The main TodoMVC app module that pulls all dependency modules
declared in same named files
   *
   * @type {angular.Module}
   */
```

```
angular.module('todomvc', ['todoCtrl', 'todoFocus', 'todoStorage']);
})();
```

The name of the application is `todomvc`, and it incorporates three services: `todoCtrl`, `todoFocus`, and `todoStorage`. The user interface is encompassed in the *index.html* file located at the root directory. It's quite large, so I'll just grab a piece. The main page body is enclosed in a `section` element, with the following definition:

```
<section id="todoapp" ng-controller="TodoCtrl as TC">
...
</section>
```

AngularJS adds annotations to HTML called *derivatives*. You'll recognize them easily, as the standard derivatives each start with "ng-", as in `ng-submit`, `ng-blur`, and `ng-model`. In the code snippet, the `ng-controller` derivative defines the controller for the view, TodoCtrl, and the reference used for it that will appear elsewhere in the template, TC.

```
<form ng-submit="TC.doneEditing(todo, $index)">
    <input class="edit"
       ng-trim="false"
       ng-model="todo.title"
       ng-blur="TC.doneEditing(todo, $index)"
       ng-keydown="($event.keyCode === TC.ESCAPE_KEY)
                  && TC.revertEditing($index)"
       todo-focus="todo === TC.editedTodo">
</form>
```

You can see several derivatives at work, and their purpose is intuitive for the most part. The `ng-model` derivative is where the view meets the model (data), in this case `todo.title`. The `TC.doneEditing` and `TC.revertEditing` are the controller functions. I pulled their code out of the controller file and replicated it below. The `TC.doneEditing` function resets the `TC.editedTodo` object, trims the edited to-do's title, and if there is no title, removes the to-do. The `TC.revertEditing` function also resets the to-do object and reassigns the original to-do to the index of the original to-do in the array of to-dos.

```
TC.doneEditing = function (todo, index) {
      TC.editedTodo = {};
      todo.title = todo.title.trim();

      if (!todo.title) {
        TC.removeTodo(index);
      }
};

TC.revertEditing = function (index) {
      TC.editedTodo = {};
      todos[index] = TC.originalTodo;
};
```

There's nothing overly complex about the code. Check out a copy of the code (GitHub location) and try it yourself.

The Backbone.js application looks and behaves the same as the AngularJS one, but the source code is very different. Though the AngularJS *app.js* file wasn't very large, the *Backbone.js* one is even smaller:

```
/*global $ */
/*jshint unused:false */
var app = app || {};
var ENTER_KEY = 13;
var ESC_KEY = 27;

$(function () {
'use strict';

// kick things off by creating the `App`
new app.AppView();
});
```

Basically, the `app.AppView()` starts the application. The *app.js* is simple, but the implementation of `app.AppView()` is not. Rather than annotating the HTML with derivatives, like AngularJS, Backbone.js makes heavy use of Userscore templates. In the *index.html* file, you'll see their use in in-page script elements such as the following representing the template for each individual to-do. Interspersed in the HTML are the template tags, such as `title`, and whether the checkbox is checked or not.

```
<script type="text/template" id="item-template">
      <div class="view">
        <input class="toggle" type="checkbox" <%= completed ? 'checked'
             : '' %>>
        <label><%- title %></label>
        <button class="destroy"></button>
      </div>
      <input class="edit" value="<%- title %>">
    </script>
```

Rendering of the items occurs in the *todo-view.js* file, but the driving force behind the rendering occurs in the *app-view.js* file. I've included a portion of the file:

```
// Add a single todo item to the list by creating a view for it, and
    // appending its element to the `<ul>`.
    addOne: function (todo) {
      var view = new app.TodoView({ model: todo });
      this.$list.append(view.render().el);
    },
```

The rendering occurs in the *todo-view.js* file, a portion of which follows. You can see the reference to the list item's identifier item-template, given previously in the script embedded into the *index.html* file. The HTML in the script element in the *index.html* provides the template for the items rendered by the view. In the template is a place-

holder for the data provided by the model for the application. In the to-do application, the data is the title of the item and whether it's completed or not.

```
// The DOM element for a todo item...
app.TodoView = Backbone.View.extend({
    //... is a list tag.
    tagName:  'li',

    // Cache the template function for a single item.
    template: _.template($('#item-template').html()),

    ...

    // Re-render the titles of the todo item.
    render: function () {
      // Backbone LocalStorage is adding `id` attribute instantly after
      // creating a model.  This causes our TodoView to render twice. Once
      // after creating a model and once on `id` change.  We want to
      // filter out the second redundant render, which is caused by this
      // `id` change.  It's known Backbone LocalStorage bug, therefore
      // we've to create a workaround.
      // https://github.com/tastejs/todomvc/issues/469
      if (this.model.changed.id !== undefined) {
        return;
      }

      this.$el.html(this.template(this.model.toJSON()));
      this.$el.toggleClass('completed', this.model.get('completed'));
      this.toggleVisible();
      this.$input = this.$('.edit');
      return this;
    },
```

It's a little harder to follow what's happening with Backbone.js than with AngularJS, but like the former, working through the ToDo application provides a great deal of clarity. I also suggest you check out this variation and give it a try.

View rendering is only one of the differences between the frameworks. AngularJS rebuilds the DOM when changes occur, while Backbone.js makes changes in place. AngularJS provides two-way data binding, which means changes in the UI and model are automatically synchronized. Backbone.js's architecture is MVP (model-view-presenter), whereas AngularJS's is MVC (model-view-controller), meaning that Backbone.js doesn't provide the same data binding and you have to roll your own. On the other hand, Backbone.js is more lightweight and can be faster than AngularJS, though AngularJS is usually simpler for new framework developers to comprehend.

Both of these frameworks and the other full-stack frameworks are used to dynamically create web pages. And by that, I don't mean the same kind of dynamic page generation explored earlier in the chapter in "The Express Application Framework" on page 192. These frameworks enable a specific type of application known as a *single-*

page application, or *SPA*. Instead of generating HTML on the server and sending it to the browser, they package the data, send it to the browser, and then format the web page using JavaScript.

The advantage to this type of functionality is that the web page doesn't always have to be updated when you change the view of the data or drill down into more detail in the page.

Consider the Gmail application. It's an example of an SPA. When you open the inbox for the application and then access one of the emails, the entire page is not reloaded. Instead, any data necessary for the email is retrieved from the server and incorporated into the page display. The result is fast and less disconcerting to the user. But you never want to look at the page source. If you want to remain sane, *never* use the page source feature on your browser to look at Google pages.

What does a good framework need to provide? In my opinion, one of the features that frameworks should support is data binding between display and data. This means if the data changes, the user interface updates. It should also support a template engine, such as the Jade template engine used with Express earlier. It also needs a way of reducing redundant code, so the framework needs to provide support for reusable components and/or modularization.

In the Express application, we saw a connection between URL routing and functions. The URL becomes the unique way of accessing either groups of data or single items. To find a unique student, you might have a URL like */students/A1234*, and the request routes to the page with details for the student identified by `A1234`. Frameworks must provide support for this type of routing.

The framework also needs to support a MV* schema, which means, at a minimum, business logic is separate from visualization logic. It could support a variation of model-view-controller (MVC), model-view-presenter (MVP), model-view-view-model (MVVM), and so on, but at a minimum it needs to support a separation of the data and the UI.

And, of course, considering the context of this book, the framework should integrate with Node.

CHAPTER 11

Node in Development and Production

The birth of Node coincided with, and even inspired, a host of new tools and techniques for code development, management, and maintenance. Debugging, testing, task management, production rollout, and support are key elements of any Node project, and it's a good thing that most are automated.

This chapter introduces some of the tools and the concepts. It is not an exhaustive list, but should provide a good start in your explorations.

Debugging Node Applications

I confess to using console logging more than I should for debugging. It is an easy approach to checking variable values and results. However, the issue with using the console is that we're impacting on the dynamics and behavior of the application, and could actually be masking—or creating—a problem just by its use. It really is better to use debugging tools, especially when the application grows beyond simple blocks of code.

Node provides a built-in debugger we can use to set breakpoints in the code and add watchers in order to view intermediate code results. It's not the most sophisticated tool in the world, but it is sufficient for discovering bugs and potential gotchas. In the next section, we'll look at a more sophisticated tool, the Node Inspector.

The Node Debugger

Given the choice, I'll always prefer native implementations rather than using third-party functionality. Luckily for our debugging needs, Node does provide built-in debugger support. It's not sophisticated but can be useful.

You can insert breakpoints into your code by inserting the `debugger` command directly in the code:

```
for (var i = 0; i <= test; i++) {
   debugger;
   second+=i;
}
```

To start debugging the application, you can specify the `debug` option when you run the application:

```
node debug application
```

To demonstrate the debugger, I created an application that has two debugger breakpoints embedded:

```
var fs = require('fs');
var concat = require('./external.js').concatArray;

var test = 10;
var second = 'test';

for (var i = 0; i <= test; i++) {
   debugger;
   second+=i;
}

setTimeout(function() {
   debugger;
   test = 1000;
   console.log(second);
}, 1000);

fs.readFile('./log.txt', 'utf8', function (err,data) {
   if (err) {
      return console.log(err);
   }
   var arry = ['apple','orange','strawberry'];
   var arry2 = concat(data,arry);
   console.log(arry2);
});
```

The application is started with the following:

```
node debug debugtest
```

If you start a Node application with the `--debug` command-line flag, you can also start the debugger by connecting to a process with the pid:

```
node debug -p 3383
```

Or you can start it by connecting to the running process via URI:

```
node debug http://localhost:3000
```

When the debugger opens, the application breaks at line 1 and lists out the top part of the code:

```
< Debugger listening on port 5858
debug> . ok
break in debugtest.js:1
> 1 var fs = require('fs');
  2 var concat = require('./external.js').concatArray;
  3
```

You can use the `list` command to list out the source code lines in context. A command like `list(10)` will list the previous 10 lines of code and the next 10 lines of code. Typing `list(25)` with the debug test application displays all the lines in the application, numbered by line. You can add additional breakpoints at this point using either the `setBreakpoint` command or its shortcut, `sb`. We'll set a breakpoint for line 19 in the test application, which inserts a breakpoint in the `fs.readFile()` callback function. We'll also set a breakpoint directly in the custom module, at line 3:

```
debug> sb(19)
debug> sb('external.js',3)
```

You'll get a warning about the script *external.js* not being loaded yet. It doesn't impact the functionality.

You can also set a watch on variables or expressions, using the `watch('expression')` command directly in the debugger. We'll watch the `test` and `second` variables, the `data` parameter, and the `arry2` array:

```
debug> watch('test');
debug> watch('second');
debug> watch('data');
debug> watch('arry2');
```

Finally, we're ready to start debugging. Typing `cont` or `c` will run the application up to the first breakpoint. In the output, we see we're at the first breakpoint and we also see the value of the four watched items. Two—`test` and `second`—have an actual value, the other two have a value of `<error>`. That's because the application is currently outside the scope of the functions where the parameter (`data`) and variable (`arry2`) are defined. Go ahead and ignore the errors for now.

```
debug> c
break in debugtest.js:8
Watchers:
  0: test = 10
  1: second = "test"
  2: data = "<error>"
  3: arry2 = "<error>"

  6
  7 for (var i = 0; i <= test; i++) {
```

```
> 8    debugger;
  9    second+=i;
 10 }
```

There are some miscellaneous commands you can try out before we move on to the next breakpoint. The `scripts` command lists out which scripts are currently loaded:

```
debug> scripts
* 57: debugtest.js
  58: external.js
debug>
```

The `version` command displays the V8 version. Type **`c`** again to go to the next breakpoint:

```
debug> c
break in debugtest.js:8
Watchers:
  0: test = 10
  1: second = "test0"
  2: data = "<error>"
  3: arry2 = "<error>"

  6
  7 for (var i = 0; i <= test; i++) {
> 8    debugger;
  9    second+=i;
 10 }
```

Notice the changed value for the `second` variable. That's because it's being modified in the `for` loop in which the debugger is contained. Typing in `c` several more times executes the loop and we can see the variable continuously being modified. Unfortunately, we can't clear the breakpoint created using the debugger statement, but we can clear a breakpoint set using `setBreakpoint` or `sb`. To use `clearBreakpoint` or `cb`, specify the name of the script and the line number of the breakpoint:

```
cb('debugtest.js',19)
```

You can also turn off a watcher with `unwatch`:

```
debug> unwatch('second')
```

Using `sb` with no value sets breakpoint to the current line:

```
debug> sb();
```

In the application, the debugger runs the application until the next breakpoint, in the `fs.readFile()` callback. Now we'll see that the `data` parameter has changed:

```
debug> c
break in debugtest.js:19
Watchers:
  0: test = 10
  1: second = "test012345678910"
```

```
  2: data = "test"
  3: arry2 = undefined

 17
 18 fs.readFile('./log.txt', 'utf8', function (err,data) {
>19   if (err) {
 20     return console.log(err);
 21   }
```

We can also see that `arry2` value is no longer an error but instead is `undefined`.

Instead of typing `c` to continue, we'll now step through each line of the application using the `next` or `n` command. When we get to line 23, the debugger opens up the external module and breaks at line 3 because of the previously set breakpoint:

```
debug> n
break in external.js:3
Watchers:
  0: test = "<error>"
  1: second = "<error>"
  2: data = "<error>"
  3: arry2 = "<error>"

  1
  2 var concatArray = function(str, arry) {
> 3   return arry.map(function(element) {
  4       return str + ' ' + element;
  5   });
```

We could have also stepped through to line 23 in the application and used the `step` or `s` command to step *into* the module function:

```
debug> s
break in external.js:3
Watchers:
  0: test = "<error>"
  1: second = "<error>"
  2: arry2 = "<error>"
  3: data = "<error>"

  1
  2 var concatArray = function(str, arry) {
> 3   return arry.map(function(element) {
  4       return str + ' ' + element;
  5   });
```

Notice how all the watched values now display an error. At this point, we're completely out of the context of the parent application. We can add watchers when we're in functions or external modules to prevent these kinds of errors or to watch variables in the context in which they're defined if the same variable is used in the application and in the module or other functions.

The Continue Bug

If you type `c` or `cont` and the application is at the end, the debugger will hang. Nothing will break you out of it either. This is a known bug.

The `backtrace` or `bt` command provides a *backtrace* of the current execution context. The value returned at this point in the debugging is displayed in the next code block:

```
debug> bt
#0 concatArray external.js:3:3
#1 debugtest.js:23:15
```

We see two entries, one for the line we're currently at in the application and one representing the current line in the imported module function.

We can step through the external function or we can return to the application using the `out` or `o` command. This command returns you to the application when you're in a function (whether that function is in local script or a module).

The Node debugger is based in REPL, and we can actually pull up the debugger's REPL by typing the `repl` command. If we want to kill the script, we can use the `kill` command, and if we want to restart the script, we can type `restart`. Be aware, though, that not only will the script restart but all the breakpoints and watchers will be cleared.

Since the debugger runs in REPL, Ctrl-C will terminate the application or you can type `.exit`.

Node Inspector

If you're ready to step up your debugging game, then you'll want to check out Node Inspector. Node Inspector incorporates debugging features you're probably familiar with by using the Blink DevTools debugger that works with either Chrome or Opera. The upside to the tool is the increased level of sophistication and its having a visual environment for debugging. The downside to the tool is system requirements. For instance, to run it in one of my Windows 10 machines, the application told me I had to install either the .NET Framework 2.0 SDK or Microsoft Visual Studio 2005.

Installing Node Inspector

If you run into the Visual Studio error, you can try the following to change the version of the Visual Studio used by the application:

```
npm install -g node-inspector --msvs_version=2013
```

Or use whatever version you have installed.

If you have the environment that can support Node Inspector, install it with:

```
npm install -g node-inspector
```

To use it, run your Node application using the following command. You'll need to add the *.js* extension with the application when you run it with Node Inspector:

```
node-debug application.js
```

One of two things will happen. If Chrome or Opera is your default browser, the application will open up in the developer tools. If your default browser isn't one of these two, you'll need to manually open up one or the other browsers and provide the URL *http://127.0.0.1:8080/?port=5858*.

Node Inspector Documentation

There is some documentation on Node Inspector at its GitHub repository. Strongloop, the company that supports the tool, also provides some documentation on using the Node Inspector. As the documentation notes, you can also get what you need from the Chrome Developer Tools documentation. Be aware, though, that Google frequently changes location of documentation and it may not be working or complete at any point in time.

I opened my *debugtest.js* file created in the last section in Node Inspector. Figure 11-1 shows the debugger with the file first loaded and after clicking the run button, which is located in the upper-righthand corner of the tool. Node Inspector honors the `debug ger` command, which becomes the first breakpoint for the application. The watch variables are shown just below the buttons to control program execution.

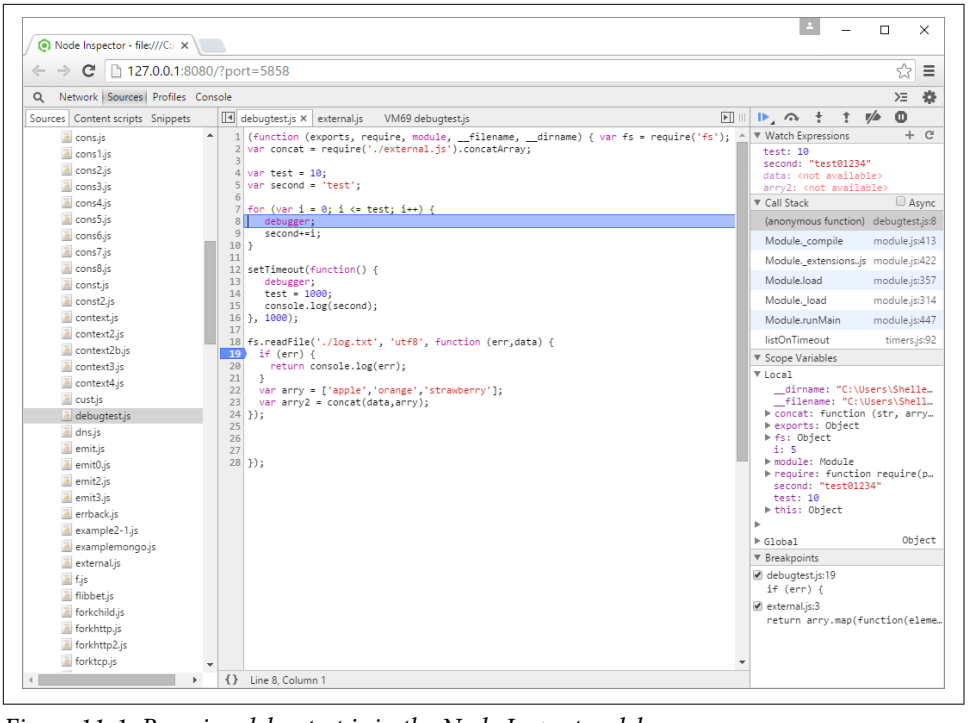


Figure 11-1. Running debugtest.js in the Node Inspector debugger

Setting new breakpoints is just a matter of clicking in the left margin of the line where you want the breakpoint. To add a new watcher, click the plus sign (+) in the Watch Expressions header. The program commands above allow you to (in order) run the application to the next breakpoint, step over the next function call, step into the function, step out of the current function, clear all breakpoints, and pause the application.

What's new with this wonderful visual interface is the listing of applications/modules in the left window, a call stack, a listing of scope variables (both local and global), and the set breakpoints, in windows on the right. If we want to add a breakpoint in the imported module, external.js, it's just a simple matter of opening the file from the list in the left and inserting a breakpoint. Figure 11-2 shows the debugger with the module loaded and breakpoint reached.

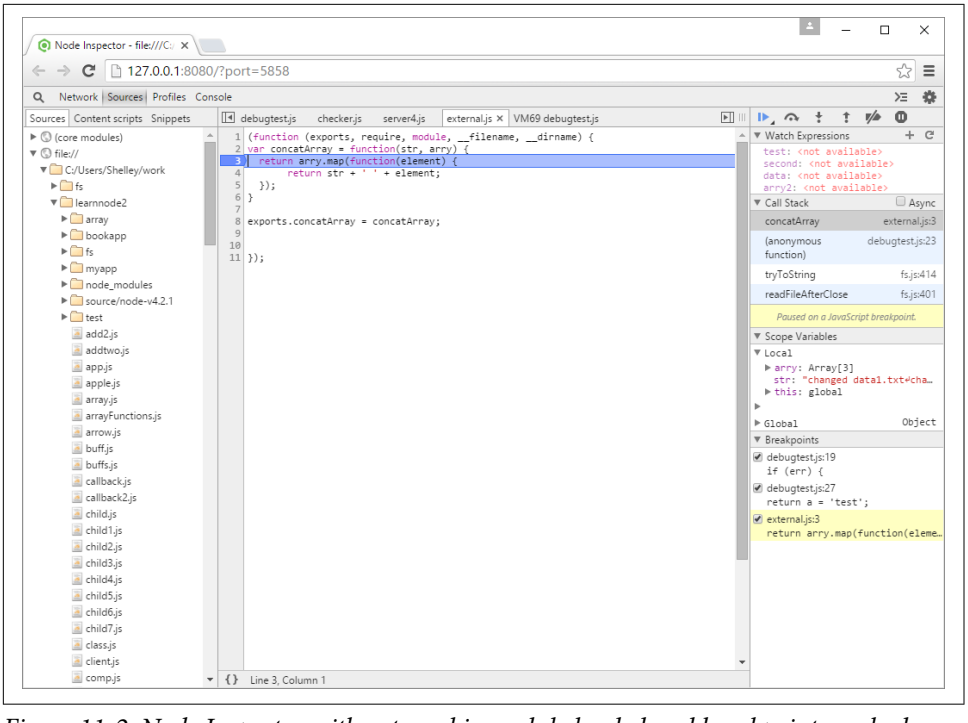


Figure 11-2. Node Inspector with external.js module loaded and breakpoint reached

What's interesting when you look at the application loaded into Node Inspector is that you can see how the application is wrapped in an anonymous function, as I described in Chapter 3 in the section titled "How Node Finds and Loads a Module" on page 56.

When you're finished with the Node Inspector, you can close the browser and terminate the batch operation by pressing Ctrl-C.

More Sophisticated Tools

Both the built-in Debugger and Node Inspector provide solid functionality to help you debug your Node applications. In addition, if you want to invest the time in setting up the environment and becoming familiar with a more sophisticated integrated development environment (IDE), you can also use a tool like Nodeclipse in the venerable Eclipse IDE.

Unit Testing

Unit testing is a way of isolating specific components of an application for testing. Many of the tests that are provided in the *tests* subdirectory of Node modules are unit tests. The tests in the *test* subdirectory of the Node installation are *all* unit tests. Many of these unit tests are built using the Assert module, which we'll go over next.

Unit Testing with Assert

Assertion tests evaluate expressions, the end result of which is a value of either `true` or `false`. If you're testing the return from a function call, you might first test that the return is an array (first assertion). If the array contents should be a certain length, you perform a conditional test on the length (second assertion), and so on. There's one Node built-in module that facilitates this form of assertion testing: Assert. Its purpose is for internal use with Node, but we can use it. We just need to be aware that it's not a true testing framework.

You include the Assert module in an application with the following:

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

To see how to use Assert, let's look at how existing modules use it. The Node application makes use of the Assert module in its module unit tests. For instance, there's a test application called *test-util.js* that tests the Utilities module. The following code is the section that tests the `isArray` method:

```
// isArray
assert.equal(true, util.isArray([]));
assert.equal(true, util.isArray(Array()));
assert.equal(true, util.isArray(new Array()));
assert.equal(true, util.isArray(new Array(5)));
assert.equal(true, util.isArray(new Array('with', 'some', 'entries')));
assert.equal(true, util.isArray(context('Array')()));
assert.equal(false, util.isArray({}));
assert.equal(false, util.isArray({ push: function() {} }));
assert.equal(false, util.isArray(/regexp/));
assert.equal(false, util.isArray(new Error));
assert.equal(false, util.isArray(Object.create(Array.prototype)));
```

Both the `assert.equal()` and the `assert.strictEqual()` methods have two mandatory parameters: an expected response and an expression that evaluates to a response. In the `assert.equal isArray`, if the expression evaluates to `true` and the expected response is `true`, the `assert.equal` method succeeds and produces no output—the result is *silent*.

If, however, the expression evaluates to a response other than what's expected, the `assert.equal` method responds with an exception. If I take the first statement in the `isArray` test in the Node source and modify it to:

```
assert.equal(false, util.isArray([]));
```

then the result is:

```
assert.js:89
  throw new assert.AssertionError({
  ^
AssertionError: false == true
    at Object.<anonymous> (/home/examples/public_html/
                          learnnode2/asserttest.js:4:8)
    at Module._compile (module.js:409:26)
    at Object.Module._extensions..js (module.js:416:10)
    at Module.load (module.js:343:32)
    at Function.Module._load (module.js:300:12)
    at Function.Module.runMain (module.js:441:10)
    at startup (node.js:134:18)
    at node.js:962:3
```

The `assert.equal()` and `assert.strictEqual()` methods also have a third optional parameter, a message that's displayed rather than the default in case of a failure:

```
assert.equal(false, util.isArray([]), 'Test 1Ab failed');
```

This can be a useful way of identifying exactly which test failed if you're running several in a test script. You can see the use of a message (a label) in the node-redis test code:

```
assert.equal(str, results, label + " " + str +
               " does not match " + results);
```

The message is displayed when you catch the exception and print out the message.

The following Assert module methods all take the same three parameters, though how the test value and expression relate to each other varies, as the name of the test implies:

`assert.equal`
: Fails if the expression results and given value are not equal

`assert.strictEqual`
: Fails if the expression results and given value are not strictly equal

`assert.notEqual`
: Fails if the expression results and given value are equal

`assert.notStrictEqual`
: Fails if the expression results and given value are strictly equal

`assert.deepEqual`
: Fails if the expression results and given value are not equal

`assert.notDeepEqual`
: Fails if the expression results and given value are equal

`assert.deepStrictEqual`
: Similar to `assert.deepEqual()` except primitives are compared with strict equal (`===`)

`assert.notDeepStrictEqual`
: Tests for deep strict inequality

The deep methods work with complex objects, such as arrays or objects. The following succeeds with `assert.deepEqual`:

```
assert.deepEqual([1,2,3],[1,2,3]);
```

but would not succeed with `assert.equal`.

The remaining `assert` methods take differing parameters. Calling `assert` as a method and passing in a value and a message is equivalent to calling `assert.isEqual` and passing in `true` as the first parameter, an expression, and a message. The following:

```
var val = 3;
assert(val == 3, 'Test 1 Not Equal');
```

is equivalent to:

```
assert.equal(true, val == 3, 'Test 1 Not Equal');
```

Or use the alias of `assert.ok`:

```
assert.ok(val == 3, 'Test 1 Not Equal');
```

The `assert.fail` method throws an exception. It takes four parameters: a value, an expression, a message, and an operator, which is used to separate the value and expression in the message when an exception is thrown. In the following code snippet:

```
try {
  var val = 3;
  assert.fail(val, 4, 'Fails Not Equal', '==');
} catch(e) {
  console.log(e);
}
```

the console message is:

```
{ [AssertionError: Fails Not Equal]
  name: 'AssertionError',
  actual: 3,
  expected: 4,
  operator: '==',
```

```
  message: 'Fails Not Equal',
  generatedMessage: false }
```

The `assert.ifError` function takes a value and throws an exception only if the value resolves to anything but `false`. As the Node documentation states, it's a good test for the error object as the first argument in a callback function:

```
assert.ifError(err); //throws only if true value
```

The last `assert` methods are `assert.throws` and `assert.doesNotThrow`. The first expects an exception to get thrown; the second doesn't. Both methods take a code block as the first required parameter, and an optional error and message as the second and third parameters. The error object can be a constructor, regular expression, or validation function. In the following code snippet, the error message is printed out because the error regular expression as the second parameter doesn't match the error message:

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  /something/
);
```

You can create sturdy unit tests using the Assert module. The one major limitation with the module, though, is the fact that you have to do a lot of wrapping of the tests so that the entire testing script doesn't fail if one test fails. That's where using a higher-level unit testing framework, such as Nodeunit (discussed next), comes in handy.

Unit Testing with Nodeunit

Nodeunit provides a way to script several tests. Once scripted, each test is run serially, and the results are reported in a coordinated fashion. To use Nodeunit, you're going to want to install it globally with npm:

```
[sudo] npm install nodeunit -g
```

Nodeunit provides a way to easily run a series of tests without having to wrap everything in try/catch blocks. It supports all of the Assert module tests and provides a couple of methods of its own in order to control the tests. Tests are organized as test cases, each of which is exported as an object method in the test script. Each test case gets a control object, typically named `test`. The first method call in the test case is to the `test` element's `expect` method to tell Nodeunit how many tests to expect in the test case. The last method call in the test case is to the `test` element's `done` method to tell Nodeunit the test case is finished. Everything in between comprises the actual test unit:

```
module.exports = {
   'Test 1' : function(test) {
      test.expect(3); // three tests
      ... // the tests
      test.done();
   },
   'Test 2' : function (test) {
      test.expect(1); // only one test
      ... // the test
      test.done();
   }
};
```

To run the tests, type **`nodeunit`**, followed by the name of the test script:

```
nodeunit thetest.js
```

Example 11-1 has a small but complete testing script with six assertions. It consists of two test units, labeled `Test 1` and `Test 2`. The first test unit runs four separate tests, while the second test unit runs two. The `expect` method call reflects the number of tests being run in the unit.

Example 11-1. Nodeunit test script, with two test units, running a total of six tests

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

module.exports = {
    'Test 1' : function(test) {
        test.expect(4);
        test.equal(true, util.isArray([]));
        test.equal(true, util.isArray(new Array(3)));
        test.equal(true, util.isArray([1,2,3]));
        test.notEqual(true, 1 > 2);
        test.done();
    },
    'Test 2' : function(test) {
        test.expect(2);
        test.deepEqual([1,2,3], [1,2,3]);
        test.ok('str' === 'str', 'equal');
        test.done();
    }
};
```

The result of running the Example 11-1 test script with Nodeunit is:

```
thetest.js
✓ Test 1
✓ Test 2

OK: 6 assertions (12ms)
```

Symbols in front of the tests indicate success or failure: a check for success and an *x* for failure. None of the tests in this script fails, so there's no error script or stack trace output.

For CoffeeScript fans, Nodeunit supports CoffeeScript applications.

Other Testing Frameworks

In addition to Nodeunit, covered in the preceding section, there are several other testing frameworks available for Node developers. Some of the tools are simpler to use than others and each has its own advantages and disadvantages. Next, I'll briefly cover three frameworks: Mocha, Jasmine, and Vows.

Mocha

Install Mocha with npm:

```
npm install mocha -g
```

Mocha is considered the successor to another popular testing framework, Espresso.

Mocha works in both browsers and Node applications. It allows for asynchronous testing via the `done` function, though the function can be omitted for synchronous testing. Mocha can be used with any assertion library.

The following is an example of a Mocha test using Assert:

```
assert = require('assert')
describe('MyTest', function() {
  describe('First', function() {
    it('sample test', function() {
      assert.equal('hello','hello');
    });
  });
});
```

Run the test with the following command line:

```
mocha testcase.js
```

The test should succeed:

```
MyTest
  First
    ✓ sample test


1 passing (15ms)
```

Vows

Vows is a behavior-driven development (BDD) testing framework and has one advantage over others: more comprehensive documentation. Testing is composed of testing suites, themselves made up of batches of sequentially executed tests. A *batch* consists of one or more contexts executed in parallel and each consisting of a *topic*. The test within the code is known as a *vow*. Where Vows prides itself on being different from the other testing frameworks is by providing a clear separation between that which is being tested (topic) and the test (vow).

I know those are some strange uses of familiar words, so let's look at a simple example to get a better idea of how a Vows test works. First, though, we have to install Vows:

```
npm install vows
```

To try out Vows, I'm using a simple circle module that returns area and circumference. Since the values are floating point and I'm testing equality, I'm limiting the returned values to four decimal points:

```
const PI = Math.PI;

exports.area = function (r) {
  return (PI * r * r).toFixed(4);
};

exports.circumference = function (r) {
  return (2 * PI * r).toFixed(4);
};
```

In the Vows test application, the circle object is the *topic*, and the area and circumference methods are the *vows*. Both are encapsulated as a Vows *context*. The *suite* is the overall test application, and the *batch* is the test instance (circle and two methods). Example 11-2 shows the entire test.

Example 11-2. Vows test application with one batch, one context, one topic, and two vows

```
var vows = require('vows'),
    assert = require('assert');

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

```
var suite = vows.describe('Test Circle');

suite.addBatch({
    'An instance of Circle': {
        topic: circle,
        'should be able to calculate circumference': function (topic) {
            assert.equal (topic.circumference(3.0), 18.8496);
        },
        'should be able to calculate area': function(topic) {
            assert.equal (topic.area(3.0), 28.2743);
        }
    }
}).run();
```

Running the application with Node runs the test because of the addition of the `run` method at the end of the `addBatch` method:

```
node vowstest.js
```

The results should be two successful tests:

```
·· ✓ OK » 2 honored (0.012s)
```

The topic is always an asynchronous function or a value. Instead of using `circle` as the topic, I could have directly referenced the object methods as topics—with a little help from function closures:

```
var vows = require('vows'),
    assert = require('assert');

var circle = require('./circle');

var suite = vows.describe('Test Circle');

suite.addBatch({
    'Testing Circle Circumference': {
        topic: function() { return circle.circumference;},
        'should be able to calculate circumference': function (topic) {
            assert.equal (topic(3.0), 18.8496);
        },
    },
    'Testing Circle Area': {
        topic: function() { return circle.area;},
        'should be able to calculate area': function(topic) {
            assert.equal (topic(3.0), 28.2743);
        }
    }
}).run();
```

In this version of the example, each context is the object given a title: `Testing Circle Circumference` and `Testing Circle Area`. Within each context, there's one topic and one vow.

You can incorporate multiple batches, each with multiple contexts which can in turn have multiple topics and multiple vows.

Keeping Node Up and Running

You do the best you can with your application. You test it thoroughly and you add error handling so that errors are managed gracefully. Still, there can be gotchas that come along—things you didn't plan for that can take your application down. If this happens, you need to have a way to ensure that your application can start again, even if you're not around to restart it.

Forever is just such a tool—it ensures that your application restarts if it crashes. It's also a way of starting your application as a daemon that persists beyond the current terminal session. Forever can be used from the command line or incorporated as part of the application. If you use it from the command line, you'll want to install it globally:

```
npm install forever -g
```

Rather than start an application with Node directly, start it with Forever:

```
forever start -a -l forever.log -o out.log -e err.log finalserver.js
```

Default values are accepted for two options: `minUpTime` (set to 1000ms) and `spinSleepTime` (set to 1000ms).

The preceding command starts a script, *finalserver.js*, and specifies the names for the Forever log, the output log, and the error log. It also instructs the application to append the log entries if the logfiles already exist.

If something happens to the script to cause it to crash, Forever restarts it. Forever also ensures that a Node application continues running, even if you terminate the terminal window used to start the application.

Forever has both options and actions. The `start` value in the command line just shown is an example of an action. All available actions are:

`start`
: Starts a script

`stop`
: Stops a script

`stopall`
: Stops all scripts

`restart`
: Restarts the script

`restartall`
: Restarts all running Forever scripts

`cleanlogs`
: Deletes all log entries

`logs`
: Lists all logfiles for all Forever processes

`list`
: Lists all running scripts

`config`
: Lists user configurations

`set <key> <val>`
: Sets configuration key value

`clear <key>`
: Clears configuration key value

`logs <script|index>`
: Tails the logs for `<script|index>`

`columns add <col>`
: Adds a column to the Forever list output

`columns rm <col>`
: Removes a column from the Forever list output

`columns set <cols>`
: Sets all columns for the Forever list output

An example of the `list` output is the following, after *httpserver.js* is started as a Forever daemon:

```
info:    Forever processes running
data:        uid  command         script         forever pid   id
logfile                            uptime
data:    [0] _gEN /usr/bin/nodejs serverfinal.js 10216   10225
/home/name/.forever/forever.log STOPPED
```

List the logfiles out with the `logs` action:

```
info:    Logs for running Forever processes
data:        script         logfile
data:    [0] serverfinal.js /home/name/.forever/forever.log
```

There are also a significant number of options, including the logfile settings just demonstrated, as well as running the script (`-s` or `--silent`), turning on Forever's

verbosity (`-v` or `--verbose`), setting the script's source directory (`--sourceDir`), and others, all of which you can find just by typing:

```
forever --help
```

You can incorporate the use of Forever directly in your code using the companion module, forever-monitor, as demonstrated in the documentation for the module:

```
var forever = require('forever-monitor');

  var child = new (forever.Monitor)('serverfinal.js', {
    max: 3,
    silent: true,
    args: []
  });

  child.on('exit', function () {
    console.log('serverfinal.js has exited after 3 restarts');
  });

  child.start();
```

Additionally, you can use Forever with Nodemon, not only to restart the application if it unexpectedly fails but also to ensure that the application is refreshed if the source is updated.

Install Nodemon globally:

```
npm install -g nodemon
```

Nodemon wraps your application. Instead of using Node to start the application, use Nodemon:

```
nodemon app.js
```

Nodemon sits quietly monitoring the directory (and any contained directories) where you ran the application, checking for file changes. If it finds a change, it restarts the application so that it picks up the recent changes.

You can pass parameters to the application:

```
nodemon app.js param1 param2
```

You can also use the module with CoffeeScript:

```
nodemon someapp.coffee
```

If you want Nodemon to monitor some directory other than the current one, use the `--watch` flag:

```
nodemon --watch dir1 --watch libs app.js
```

There are other flags, documented with the module.

To use Nodemon with Forever, wrap Nodemon within Forever and specify the `--exitcrash` option to ensure that if the application crashes, Nodemon exits cleanly and passes control to Forever:

```
forever start nodemon --exitcrash serverfinal.js
```

If you get an error about Forever not finding Nodemon, use the full path:

```
forever start /usr/bin/nodemon --exitcrash serverfinal.js
```

If the application does crash, Forever starts Nodemon, which in turn starts the Node script, ensuring that not only is the running script refreshed if the source is changed, but also that an unexpected failure doesn't take your application permanently offline.

Benchmark and Load Testing with Apache Bench

A robust application that meets all the user's needs is going to have a short life if its performance is atrocious. We need the ability to *performance test* our Node applications, especially when we make tweaks as part of the process to improve performance. We can't just tweak the application, put it out for production use, and let our users drive out performance issues.

Performance testing consists of benchmark testing and load testing. *Benchmark testing*, also known as *comparison testing*, is running multiple versions or variations of an application and then determining which is better. It's an effective tool to use when you're tweaking an application to improve its efficiency and scalability. You create a standardized test, run it against the variations, and then analyze the results.

Load testing, on the other hand, is basically stress testing your application. You're trying to see at what point your application begins to fail or bog down because of too many demands on resources or too many concurrent users. You basically want to drive the application until it fails. Failure is a success with load testing.

There are existing tools that handle both kinds of performance testing, and a popular one is ApacheBench. It's popular because it's available by default on any server where Apache is installed—and few servers don't have Apache installed. It's also an easy-to-use, powerful little testing tool. When I was trying to determine whether it's better to create a static database connection for reuse or to create a connection and discard it with each use, I used ApacheBench to run tests.

ApacheBench is commonly called ab, and I'll use that name from this point forward. ab is a command-line tool that allows us to specify the number of times an application is run and by how many concurrent users. If we want to emulate 20 concurrent users accessing a web application a total of 100 times, we'd use a command like the following:

```
ab -n 100 -c 20 http://burningbird.net/
```

It's important to provide the final slash, as ab expects a full URL, including path.

ab provides a rather rich output of information. An example is the following output (excluding the tool identification) from one test:

```
Benchmarking burningbird.net (be patient).....done


Server Software:        Apache/2.4.7
Server Hostname:        burningbird.net
Server Port:            80

Document Path:          /
Document Length:        36683 bytes

Concurrency Level:      20
Time taken for tests:   5.489 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      3695600 bytes
HTML transferred:       3668300 bytes
Requests per second:    18.22 [#/sec] (mean)
Time per request:       1097.787 [ms] (mean)
Time per request:       54.889 [ms] (mean, across all concurrent requests)
Transfer rate:          657.50 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   2.3      0       7
Processing:   555 1049 196.9   1078    1455
Waiting:       53  421 170.8    404     870
Total:        559 1050 197.0   1080    1462

Percentage of the requests served within a certain time (ms)
  50%   1080
  66%   1142
  75%   1198
  80%   1214
  90%   1341
  95%   1392
  98%   1415
  99%   1462
 100%   1462 (longest request)
```

The lines we're most interested in are those having to do with how long each test took, and the cumulative distribution at the end of the test (based on percentages). According to this output, the average time per request (the first value with this label) is 1097.787 milliseconds. This is how long the average user could expect to wait for a response. The second line has to do with throughput, and is probably not as useful as the first.

The cumulative distribution provides a good look into the percentage of requests handled within a certain time frame. Again, this indicates what we can expect for an average user: response times between 1,080 and 1,462 milliseconds, with the vast majority of responses handled in 1,392 milliseconds or less.

The last value we're looking at is the requests per second—in this case, 18.22. This value can somewhat predict how well the application will scale because it gives us an idea of the maximum requests per second—that is, the upper boundaries of application access. However, you'll need to run the test at different times, and under different ancillary loads, especially if you're running the test on a system that serves other uses.

The Loadtest Application

You can also use the Loadtest application to do load testing:

```
npm install -g loadtest
```

The advantage over Apache Bench is you can set requests as well as users:

```
loadtest [-n requests] [-c concurrency] [-k] URL
```

CHAPTER 12

Node in New Environments

Node has expanded to many different environments beyond the basic server in Linux, OS X, and Windows.

Node has provided a way of using JavaScript with microcomputers and microcontrollers such as Raspberry Pi and Arduino. Samsung is planning on integrating Node into its vision of the Internet of Things (IoT), even though the IoT technology it bought, SmartThings, is based on a variation of Java (Groovy). And Microsoft has embraced Node, and is now working to extend it by providing a variation of Node with its own engine, Chakra, running it.

The many faces of Node are what makes it both exciting and fun.

Node and Mobile Environments

Node has also made its way into the mobile world, but attempting to squish a section on it into this book was beyond my ability to compress the subject down to a minimum. Instead, I'll point the reader to a book on the topic, *Learning Node.js for Mobile Application Development* (Packt, 2015), by Stefan Buttigieg and Milorad Jevdjenic.

Samsung IoT and GPIO

Samsung has created a variation of Node called IoT.js, as well as a JavaScript version for IoT technologies called JerryScript. From the documentation, the primary reason for new variations is to develop tools and technologies that work in devices with lower memory than the traditional JavaScript/Node environments.

In one graph accompanying a Samsung employee's presentation, they show a full implementation of JerryScript with a binary size of 200 KB and a memory footprint

of 16 KB to 64 KB. This, compared to V8 requiring a binary size of 10 MB, and a memory footprint of 8 MB. When you're working with IoT devices, every scrap of space and memory matters.

Focusing on IoT.js, in the documentation you'll find that it supports a subset of the core Node modules such as Buffer, HTTP, Net, and File System. Considering its targeted use, no support for modules such as Crypto is understandable. You know you're in the world of IoT when you see that it also features a new core module: GPIO. It represents the application interface to the physical hardware, which forms the bridge between application and device.

GPIO is an acronym for general-purpose input/output. It represents a pin on an integrated circuit that can be either an input or output, and whose behavior is controlled by applications we create. The GPIO pins provide an interface to the device. As inputs, they can receive information from devices such as temperature or motion sensors; as outputs, they can control lights, touch screens, motors, rotary devices, and so on.

On a device such as a Raspberry Pi (covered in "Node for Microcontrollers and Microcomputers" on page 245), there is a panel of pins along one side, most of which are GPIO pins, interspersed with ground and power pins. Figure 12-1 shows an actual photo of the pins, and underneath, a *pinout diagram* showing the GPIO, power, and ground pins for a Raspberry Pi 2 Model B.

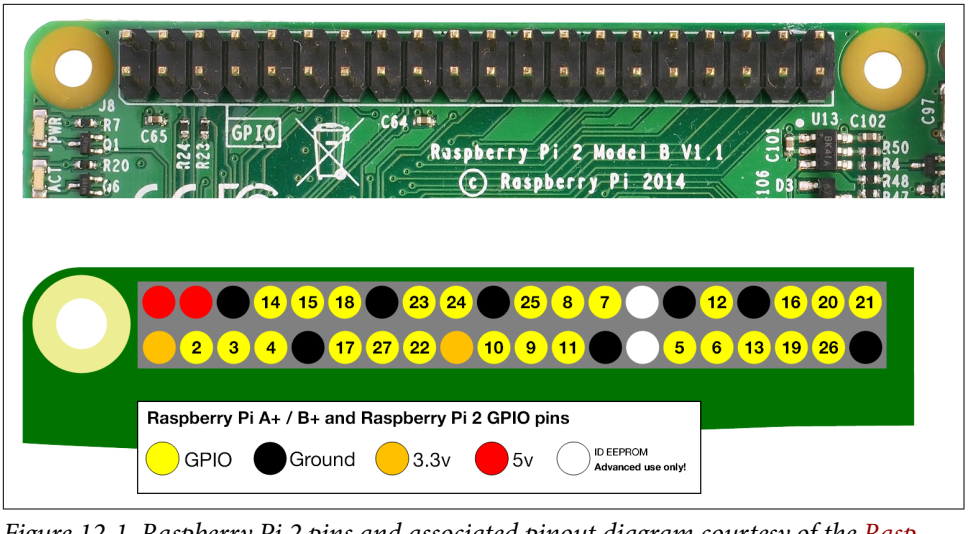


Figure 12-1. Raspberry Pi 2 pins and associated pinout diagram courtesy of the Raspberry Pi Foundation, used via CC license CC-BY-SA

As you might note, the pin number in the pinout diagram doesn't reflect its actual physical location in the board. The number appearing in the pin label is the *GPIO*

number. Some APIs, including Samsung's IoT.js, expect the GPIO number when it asks for a pin number.

To use the Samsung IoT.js, you initialize the GPIO object and then call one of the functions, such as `gpio.setPen()`, which takes a pin number as first parameter, direction (i.e., `'in'` for input, `'out'` for output, and `'none'` to release), an optional mode, and a callback. To send data to the pin, use the `gpio.writePin()` function, giving it the pin number, a boolean value, and a callback.

The Samsung IoT.js is definitely a work in progress. And it seems to conflict with other Samsung work known as SAMIO, which defines a data exchange platform that can, among other things, allow communication between an Arduino and a Raspberry Pi when monitoring a flame sensor (see the associated tutorial)—and which does incorporate the use of Node. However, it's all in active development, so I guess we'll stay tuned.

Raspberry Pi and Arduino

We'll look at Raspberry Pi and Arduino in more detail in "Node for Microcontrollers and Microcomputers" on page 245.

Windows with Chakra Node

On January 19, 2016, Microsoft issued a pull request (PR) to enable Node to run with Microsoft's ChakraCore (JavaScript) engine. The company created a V8 shim that implements most of the V8 essential APIs, allowing Node to run on ChakraCore transparently.

The idea of Node running on something other than V8 is both interesting and, in my opinion, compelling. Though it seems that the V8 is driving the development of Node, at least Node Current releases, technically there is no absolute requirement that Node be built on V8. After all, the Current updates are geared more toward enhancements of the Node API and incorporating new ECMAScript innovations.

Microsoft has open-sourced ChakraCore, which is a necessary first step. It provides the JavaScript engine support for the company's new browser, Edge. And the company claims the engine is superior to V8.

While the debate and testing is ongoing about the PR, you can actually test Node on ChakraCore as long as you have a Windows machine, Python (2.6 or 2.7), and Visual Studio (such as Visual Studio 2015 Community, available for free download). You can also download a prebuilt binary. Microsoft makes available binaries for ARM (for Raspberry Pi), as well as the more traditional x86 and x64 architectures. When I installed it in my Windows machine, it created the same type of Command window I

used for the non-ChakraCore Node. Best of all, it can be installed on the same machine as a Node installation, and the two can be used side by side.

I tried out several examples from the book and ran into no problems. I also tried out an example reflecting a recent change for Node that allows the `child_process.spawn()` function to specify a shell option (covered in Chapter 8). The example didn't work with the 4.x LTS release, but it does work with the most recent Current release (6.0.0 at the time this was written). And it worked with the binary build of the Node based on ChakraCore. This, even though the new Current release happened a few days prior. So Microsoft is grabbing the most recent Current release of the Node API in its Node/ChakraCore binary build.

Even of more interest, when I tried the following ES6 reflect/proxy example:

```
'use strict'

// example, courtesy of Dr. Axel Rauschmayer
// http://www.2ality.com/2014/12/es6-proxies.html


let target = {};
    let handler = {
        get(target, propKey, receiver) {
            console.log('get ' + propKey);
            return 123;
        }
    };

let proxy = new Proxy(target, handler);

console.log(proxy.foo);

proxy.bar = 'abc';
console.log(target.bar);
```

It did work with the ChakraCore Node, but it did not work with the V8 Node, not even the most recent edition. Another advantage the ChakraCore developers claim is a superior implementation of new ECMAScript enhancements.

It didn't work with V5 Stable, but it did work with the newer V6 Current Node.

Currently, ChakraCore only works in Windows, and they've provided a Raspberry Pi binary. They also promise a Linux version soon.

Node for Microcontrollers and Microcomputers

Node has found a very comfortable home with microcontrollers like Arduino and microcomputers like Raspberry Pi.

I use *microcomputer* when discussing Raspberry Pi, but in actuality, it's a fully functioning—albeit small—computer. You can install an operating system on it, such as Windows 10 or Linux; attach a keyboard, mouse, and monitor; and run many applications including a web browser, games, or office applications. The Arduino, on the other hand, is used for repetitive tasks. Rather than connecting a keyboard or monitor directly to the device, you connect the device to your computer and use an associated application to build and upload a program to the device. It's an *embedded computer*, as compared to a computer (like Raspberry Pi).

Using Raspberry Pi and Arduino Together

You don't have to choose between Raspberry Pi and Arduino; you can use them together. The Pi becomes the brain and the Arduino, the brawn.

Arduino and Raspberry Pi are popular in their respective categories, but they're only a subset of available devices, many of which are Arduino-compatible. There's even a wearable device (LilyPad).

If you're new to IoT and connected-device development, I recommend starting with the Arduino Uno and then trying out the Raspberry Pi 2. There are kits available at online sites such as AdaFruit, SparkFun, Cana Kit, Amazon, Maker Shed, and others, as well as the board makers themselves. The kits include all you need to start working with the boards and are inexpensive (under $100.00). They come with project books and the components needed for several projects.

In this section, I'm going to demonstrate how to create a Hello World application for these types of devices. It consists of an application that accesses an LED light on a GPIO pen and flashes the light. I'm going to demonstrate the application on an Arduino Uno and a Raspberry Pi 2.

New Raspberry Pi 3

Just as I was finishing, Raspberry Pi 3 was released. It should be simple to convert the Raspberry Pi example later in the book to the newer device. Best of all: Raspberry Pi 3 comes with integrated WiFi.

First, though, you can't get far in the connected-device world without knowing a little bit about electronics and about that wonderful tool known as Fritzing.

Fritzing

Fritzing is open source software that gives people the tools to prototype a design and then actually have the design physically created through the Fritzing Fab. The software is available free of charge (though I recommend a donation if you find the tool helpful). When you see graphical diagrams with Arduino and Raspberry Pi projects, they're invariably made with the Fritzing app.

Download the App

The Fritzing application can be downloaded from the Fritzing website. It is not a small application, which isn't surprising considering the number of graphical components necessary for the application. You can download a Windows, OS X, or Linux version.

When the tool opens and you create a new *sketch*, as the design diagram is called, a breadboard is automatically added to the sketch. A *breadboard* is an easy way of prototyping an electrical project. You don't have to solder any of the components; all you have to do is plug one end of the wire to the board pin and plug the other end into the breadboard. Under the plastic exterior of the breadboard are strips of conductive metal: long vertical ones under the *power rails* if the board has them (delimited by red and blue lines, and highlighted in Figure 12-2), and shorter, horizontal ones under each column *terminal strip* (shown in Figure 12-3). You then add all the project components to the breadboard.

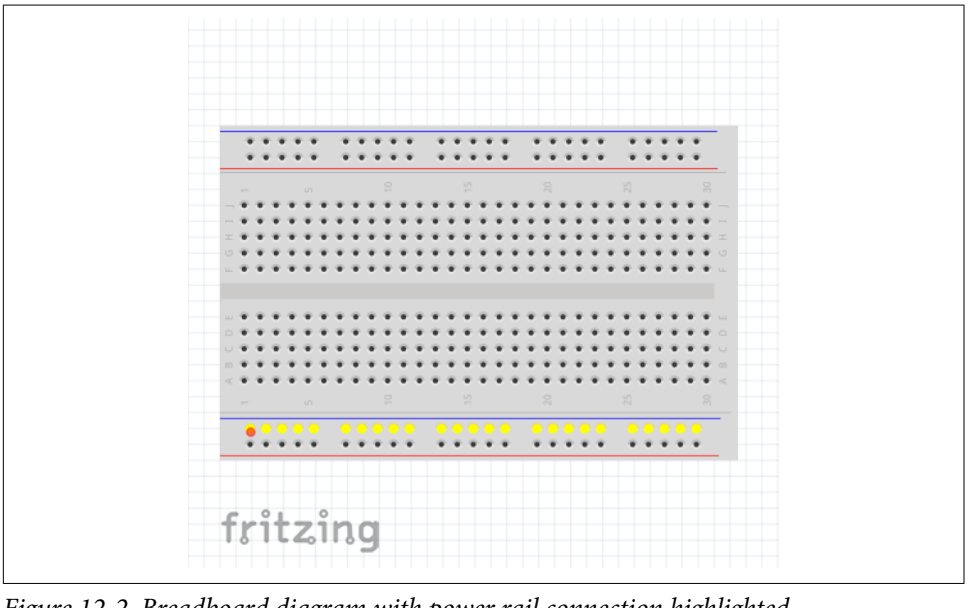


Figure 12-2. Breadboard diagram with power rail connection highlighted

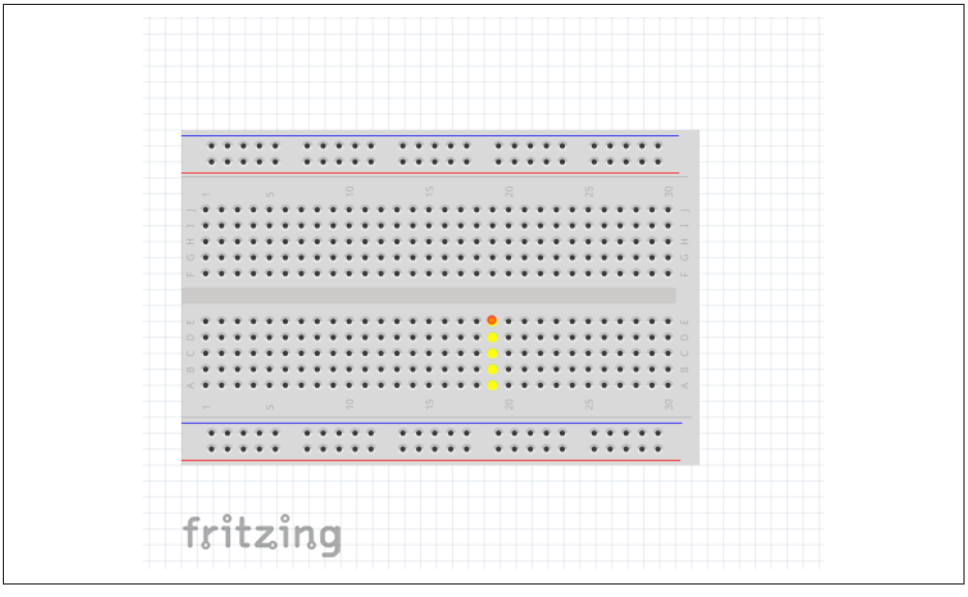


Figure 12-3. Breadboard diagram with terminal strip highlighted

Anatomy of the Breadboard

For a more detailed look at the breadboard, its history, and how it works, I recommend the SparkFun "How to Use a Breadboard" tutorial.

The components you use are dragged and dropped from the parts families in the tool to the right. In Figure 12-4, you can see the sketch I made for the Arduino pulsing LED example in the next section. It consists of an Arduino Uno dragged from the right, a half-sized breadboard, two connecting wires, a resistor, and an LED from the same parts list with the color of the LED changed from red to blue. The parts description is what shows at the bottom right. You can manually edit the information for all components in this section, including the breadboard (modified from full to half).

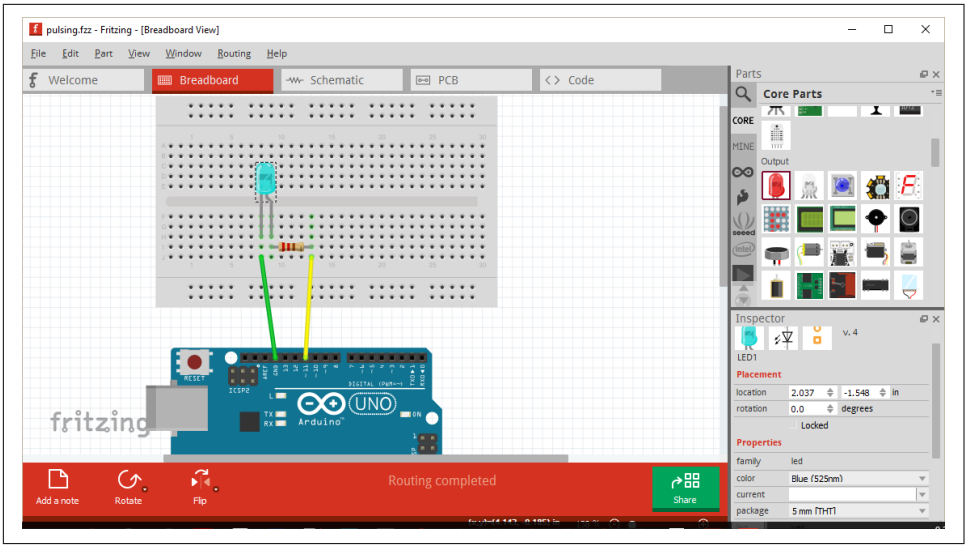


Figure 12-4. Fritzing sketch of Arduino pulsing project

To create the wires, click and drag from the Arduino pin to the place where you want the wire to end on the breadboard. The tool automatically adds the wire. Add each component by dragging from the parts menu to the sketch. To ensure the circuit is accurate and complete, you can also check out the schematic view of the project, as shown in Figure 12-5.

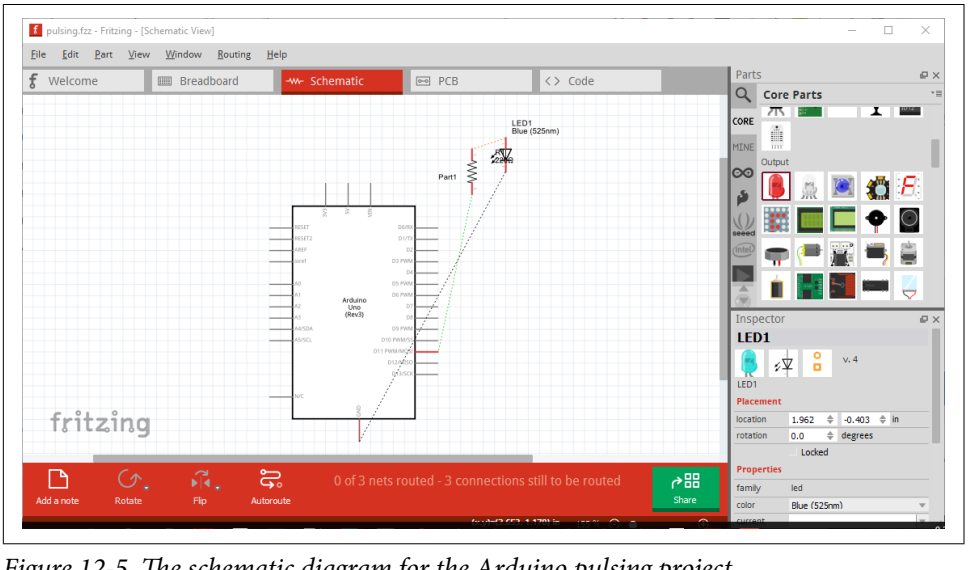


Figure 12-5. The schematic diagram for the Arduino pulsing project

The LED has two legs, called *leads*, one longer than the other. In the diagram, the longer lead is shown bent. The longer lead is the positive (anode) lead, whereas the shorter lead is the negative (cathode) lead. In the sketch, the negative lead is plugged into the Arduino's ground, whereas the positive lead is plugged into the 11 pin.

One wire is attached from the ground pin (GND) pin to the board. The LED is placed in the board with the short lead (cathode) placed into the same column as the wire from the ground. A resistor is placed in the board with one lead in the same column as the LED's anode lead. Lastly a wire connecting the other resistor lead to the 11 pin on the Arduino is added, completing the circuit.

The resistor is the new component in this sketch. Resistors do what they sound like: resist the electrical current. If a resistor wasn't added to the circuit (the complete sequence is a circuit), the LED could attempt to draw too much electricity, potentially damaging the GPIO pin and board.

The resistance is measured in ohms (Ω), and the resistor in the sketch is 220 ohms. You can tell how many ohms a resistor is by reading the bands on the resistor. Different colors in different bands (and a different number of bands) define what type of resister is included in the circuit.

The Electrical Bits

How do you know which resistor to use? It's a thing called Ohm's Law. Yes, all that stuff you learned in school can finally now be used. For more on Ohm's Law and selecting resistors, see "Calculating correct resistor value to protect Arduino pin" and "Do I really need resistors when controlling LEDs with Arduino?"

The LED is *polarized*, which means it has direction. How it's placed in the board (and circuit) is important, with the cathode attached to the ground. However, the resistor is not polarized, so it can be placed in either direction on the board. In addition, the resistor does not have to be placed after the LED; as shown in Figure 12-2, it can be placed before the LED. Since this is a simple circuit, all that matters is that the circuit has a resistor. The Fritzing sketch for the Raspberry Pi project covered in "Node and Raspberry Pi 2" on page 258, with the resistor placed before the LED, is shown in Figure 12-6.

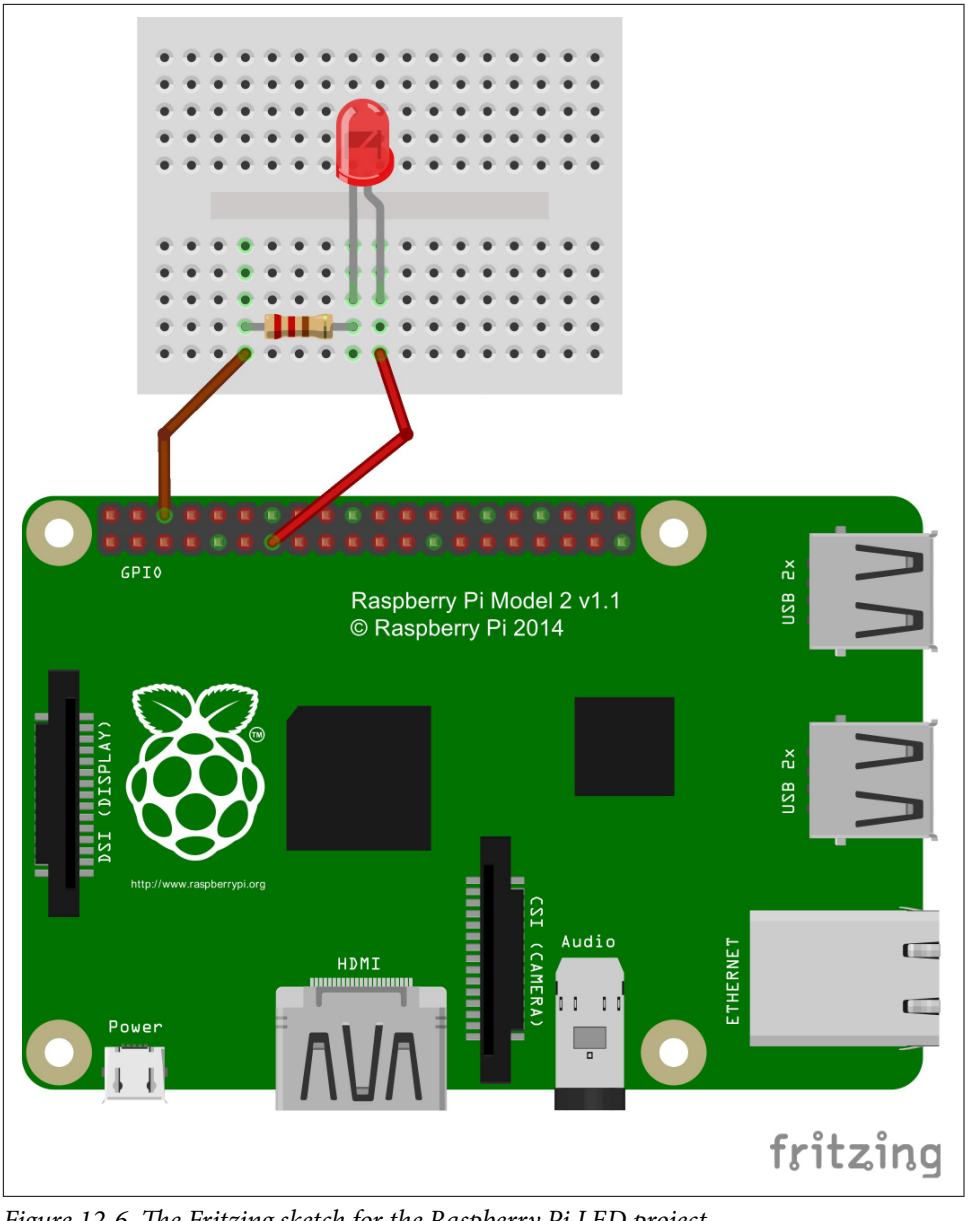


Figure 12-6. The Fritzing sketch for the Raspberry Pi LED project

This is a super quick introduction to just some of the components of a circuit, but it's all you need to work with the examples in the next two sections.

Node and Adruino

To program an Arduino Uno, you need to install Arduino software on your computer. The version I used for the example was 1.7.8, on a Windows 10 machine. There are versions for OS X and Linux. The Arduino sites provides excellent, detailed installation instructions. Once installed, connect the Arduino to the PC via a USB cable, and note the serial port to connect to the board (COM3 for me).

Next you need to upload *firmata* to the Arduino in order to use Node. The firmata implements the protocol for communicating with the microcontroller using software on the computer. In the Arduino application, select File→Examples→Firmata→StandardFirmata. Figure 12-7 show the firmata loaded into the application. There's a right-pointing arrow at the top of the window. Click it to upload the firmata to the Arduino.

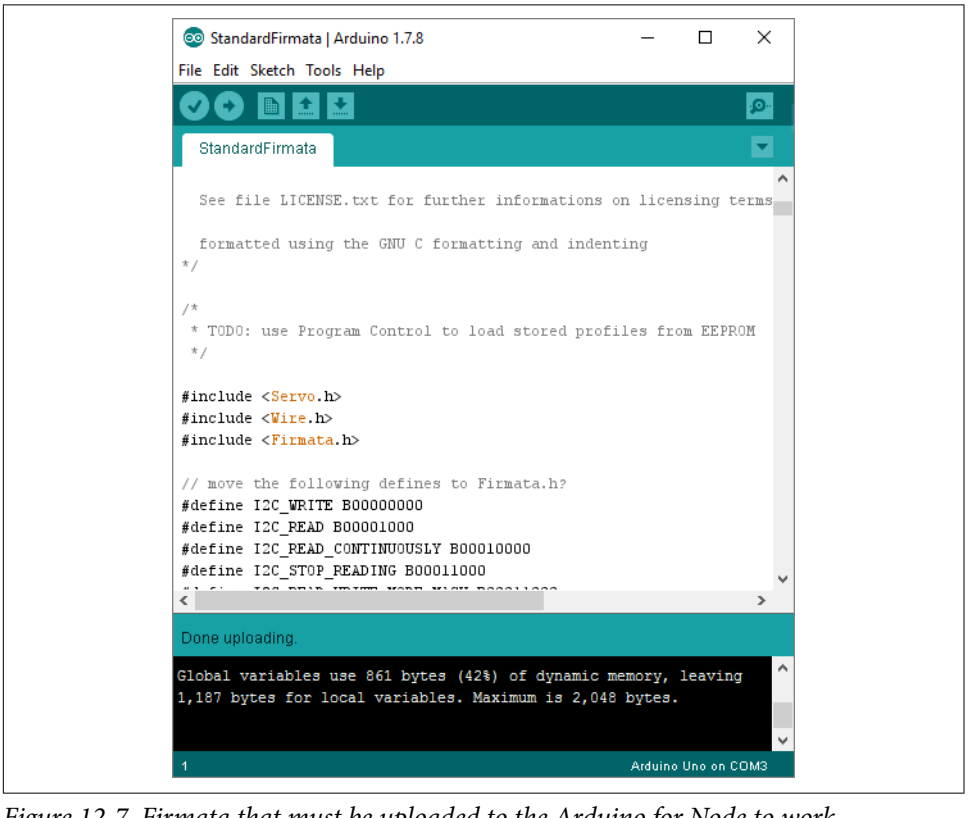


Figure 12-7. Firmata that must be uploaded to the Arduino for Node to work

Node and JavaScript is not the default or even the most common way to control either the Arduino or the Raspberry Pi (Python is a popular choice). However, there

is a Node framework called Johnny-Five that provides a way of programming these devices. Install it in Node using the following:

```
npm install johnny-five
```

The Johnny-Five website provides not only a description of the extensive API you can use to control the board, but also several examples, including the "Hello World" blink application. Unlike the example the Johnny-Five site provides, which actually connects an LED directly to the Arduino Uno board pins, we're going to blink the LED that's embedded in the board. We can access this LED by using pin number 13.

The code to blink the embedded LED (or an external LED connected to pin 13) is:

```
var five = require("johnny-five");
var board = new five.Board();

board.on("ready", function() {
  var led = new five.Led(13);
  led.blink(500);
});
```

The app loads the Johnny-Five module and creates a new board, representing the Arduino. When the board is ready, the app creates a new LED, using the pin number of 13. Note that this is not the same as the GPIO number; Johnny-Five uses a physical numbering system reflecting pin locations marked on the card. Once the LED object is created, its `blink()` function is called. The result is shown in Figure 12-8 with an arrow pointing out the LED.

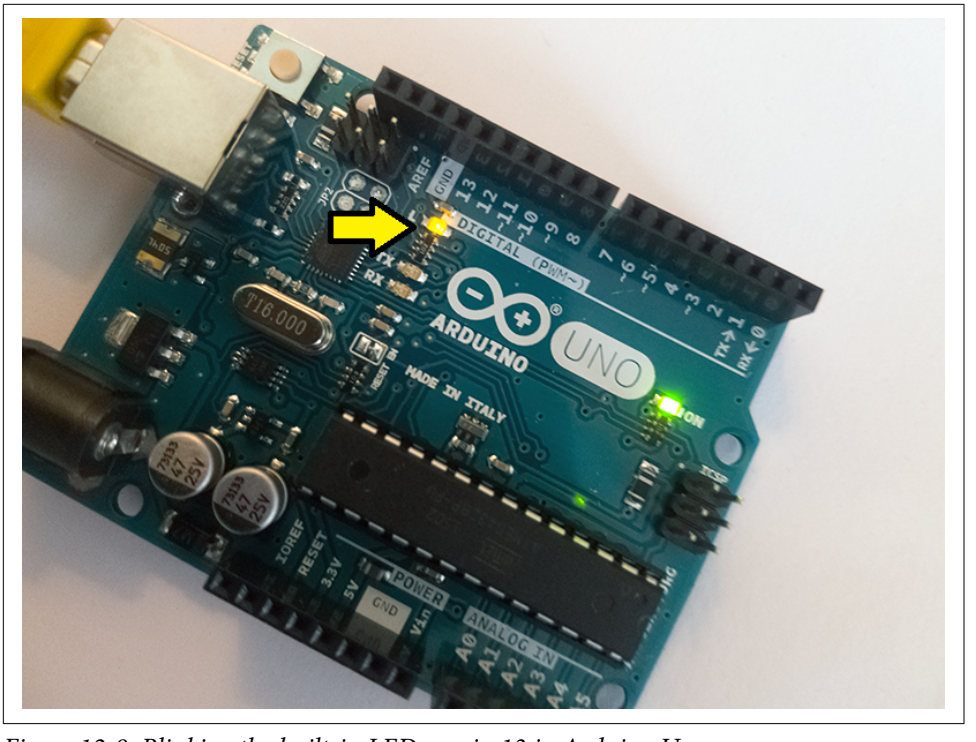


Figure 12-8. Blinking the built-in LED on pin 13 in Arduino Uno

The application opens a REPL. To terminate the application, type **`.exit`**. The LED will continue to blink until you terminate the application; if the LED is lit when you quit, it will stay lit until you power off the device.

Safety, Safety, Safety

IoT is all fun and games until you fry your board. Or your computer.

Seriously, you can safely and easily work with most IoT, small-device applications as long as you follow instructions. That's why it's important to always double-check diagrams when you're placing components and to use the right components.

It's also important to keep your workspace clean and uncluttered, use a good strong light, make sure you don't walk across a nylon rug with socks on before you touch anything electronic (ground yourself), unplug boards before changing the components, and keep the curious cat and the hyperactive dog (or small child) out of the room when you work. I also really don't recommend putting your coffee cup right next to your board.

You'll also want to put the electronics away when you're done if you have small children—there are very few things outside of the boards that can't be swallowed by a

child, and the LEDs look particularly edible. The projects are an excellent learning experience for older children, but use common sense about how old the child needs to be to work on the projects independently.

I wear glasses, but if you don't, you really want to look into getting work glasses for protection.

There are some electronic safeguards in place. The Arduino Uno board has on-board regulators that should prevent that cute little device from frying your computer. However, the board itself is vulnerable to damage. They are inexpensive but not to the point where frying them or damaging the pins on a regular basis won't cause you some pain.

Find more at Electronics at StackExchange.

The blinky LED is fun, but now that we have this shiny, new toy, let's take it for a real spin. The LED object supports several interesting sounding functions like `pulse()` and `fadeIn()`. However, these functions require a pulse width modulation (PWM) pin, also known as an *analog output*. The 13 pin is not PWM. However, the 11 pin is, which you can determine by the tilde (~) character that precedes it on the board (~11).

First, unplug the board. Then grab a breadboard, LED light, 220-ohm resister, and two connecting wires. Most kits should have all these components.

Turn the board so the "Arduino" label is right side up. Along the top is a row of pins. Plug a connecting wire into the ground pin, which is marked GND. Plug a second wire into the pin labeled ~11. In a breadboard next to the Arduino, follow the Fritzing sketch shown in Figure 12-9 to place the rest of the components into the breadboard. When you're finished, plug the Arduino back in.

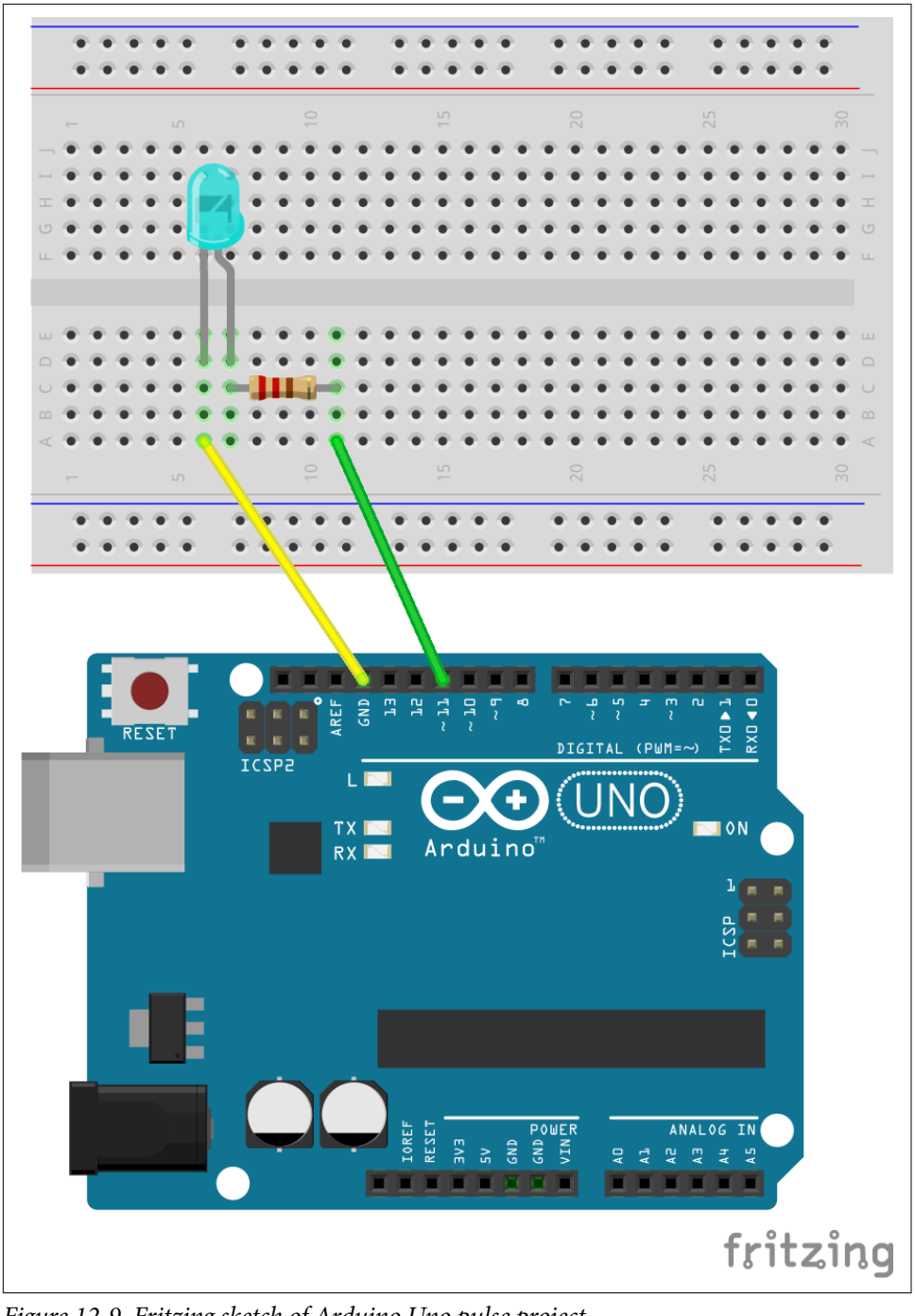


Figure 12-9. Fritzing sketch of Arduino Uno pulse project

The Example 12-1 application exposes several functions to the REPL via the `REPL.inject()` function. This means you can control the board when you're in the REPL environment. All you have to do is type in the name of the function, such as `on()` to turn the light on, and `fadeOut()` to turn the light out by fading.

Some of the functions, such as `pulse()`, `fadeIn()`, and `fadeOut()`, require that PWM pin. The application also uses an animation for the `pulse()` function. To stop the animation, type in **`stop()`** at the REPL console. You'll see the "Animation stopped" message. You'll still need to turn the LED off after you stop it. And you need to stop animations before you turn off the LED, or they won't stop animating.

Example 12-1. Interactive application using REPL to control LED

```
var five = require("johnny-five");
var board = new five.Board();

board.on("ready", function() {
  console.log("Ready event. Repl instance auto-initialized!");

  var led = new five.Led(11);

  this.repl.inject({
    // Allow limited on/off control access to the
    // Led instance from the REPL.
    on: function() {
      led.on();
    },
    off: function() {
      led.off();
    },
strobe: function() {
    led.strobe(1000);
},
pulse: function() {
    led.pulse({
           easing: "linear",
           duration: 3000,
           cuePoints: [0, 0.2, 0.4, 0.6, 0.8, 1],
           keyFrames: [0, 10, 0, 50, 0, 255],
           onstop: function() {
               console.log("Animation stopped");
           }
       });
},
stop: function() {
    led.stop();
},
fade: function() {
    led.fadeIn();
```

```
},
fadeOut: function() {
    led.fadeOut();
}
  });
});
```

Run the application using Node:

```
node fancyblinking
```

Once you've received the "Ready event ..." message, you can now enter commands. An example is the following, which strobes the light:

```
>> strobe()
```

Typing `stop()` stops the strobe effect, and `off()` turns the light off altogether. Try the fancier `pulse()`, `fade()`, and `fadeOut()` next. Figure 12-10 shows the project while the light is pulsing (a little hard to demonstrate in a static picture—trust me, it works).

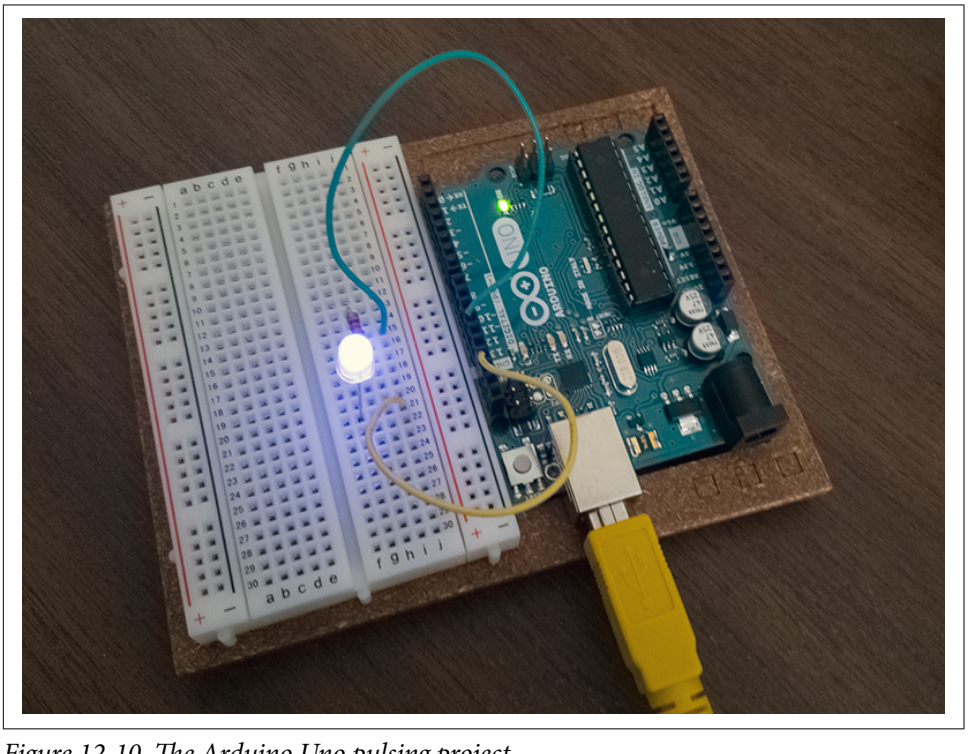


Figure 12-10. The Arduino Uno pulsing project

Once you've mastered the blinking and pulsing LED, check out some of the other Node/Arduino applications you can try:

- Real-Time Temperature Logging with Arduino, Node, and Plotly
- The Arduino Experimenter's Guide to NodeJs has a whole host of projects to try, and provides the finished code
- Controlling a MotorBoat using Arduino and Node
- As an alternative to Johnny-Five, try out the Cylon Arduino module
- Arduino Node.js RC Car Driven with the HTML5 Gamepad API
- How to Control Philips Hue Lights from an Arduino (and Add a Motion Sensor)

You could stick with Arduino and be busy for months. However, Raspberry Pi adds a whole new dimension to IoT development, as you'll see in the next section.

Node and Raspberry Pi 2

The Raspberry Pi 2 is a much more sophisticated card than the Arduino. You can connect a monitor, keyboard, and mouse to it and use it as an actual computer. Microsoft provides a Windows 10 installation for the device, but most people use Rasbpian, a Linux implementation based on Debian Jesse.

Connect to the Raspberry Pi with SSH

You can connect to your Raspberry Pi with SSH if you add a WiFi dongle. They're very inexpensive, and the ones I've used have worked right out of the box. Once you have access to the Internet on your Pi, SSH should be enabled by default. Just grab the IP address for the Pi and use it in your SSH program. WiFi is builtin by default with the new Raspberry Pi 3.

The Raspberry Pi 2 runs off a MicroSD card. It should be at least 8 GB in size and be Class 10. The Raspberry Pi site provides installation instructions, but I recommend formatting the card; copying over the New Out Of Box (NOOBs) software, which allows you to pick which operating system to install; and then installing Raspbian. You can also install a Rasbian image directly, downloaded from the site.

The newest Rasbian at the time of writing was released in February 2016. It comes with Node installed, since it also features Node-RED, a Node-based application that allows you to literally drag and drop a circuit design and power a Raspberry Pi directly from the tool. However, the version of Node is older, v0.10.x. You'll be using Johnny-Five to use Node to control the device and it should work with this version, but you'll probably want to upgrade Node. I recommend using the LTS version of Node (4.4.x at the time of writing), and following Node-REDs instructions for upgrading Node and Node-RED manually.

Check for Node

Node may be installed but under a different name for Node-RED. If so, you should be able to install a newer version of Node directly without having to uninstall any software.

Once Node is upgraded, go ahead and install Johnny-Five. You'll need to also install another module, raspi-io, which is a plug-in that allows Johnny-Five to work with Raspberry Pi.

```
npm install johnny-five raspi-io
```

Do feel free to explore your new computer, including the desktop applications. Once you're finished, though, the next part is setting up the physical circuit. To start, power off the Raspberry Pi.

You'll need to use a breadboard for the blinking light "Hello World" application. In addition, you'll need a resistor, preferably 220 ohms, which is the size of resistor frequently included in most Raspberry Pi kits. It should be a four-band resistor: red, red, brown, and gold.

Reading the Resistor Bands

Digi-Key Electronics provides a really handy calculator and color graphic for determining the value of ohms for your resistors. If you have trouble distinguishing color, you'll need help from a friend or family member, or the use of a *multimeter* for measuring the value.

The Fritzing sketch was shown in Figure 12-6. Add the resistor and LED to the breadboard, with the cathode lead (short lead) of the LED parallel with the last leg of the resistor. Carefully take two of the wires that came with your Raspberry Pi 2 kit and connect one to the GRND pin (third pin from the left, top row) and one to pin 13 (seventh pin from the left, second row) on the Raspberry Pi board. Attach the other ends to the breadboard: the GRND wire goes parallel to the first lead of the resistor, and the GPIO pin wire is inserted parallel to the anode lead (longer lead) of the LED.

The Fragile Raspberry Pi Pins

The pins are fragile on the Raspberry Pi, which is why most folks use a *breakout*. A breakout is a wide cable that plugs into the Raspberry Pi pins and then attaches to the breadboard. The components then connect to the breakout rather than directly to the Raspberry Pi.

Power up the Raspberry Pi again. Type in the Node application. It's almost the same as the one for Arduino, but you'll be using the raspi-io plug-in. In addition, how you specify the pin number changes. In the Arduino, you used a number for the pin. In Raspberry Pi, you'll use a string. The differences are shown in bold in the code:

```
var five = require("johnny-five");
var Raspi = require("raspi-io");
var board = new five.Board({
  io: new Raspi()
});

board.on("ready", function() {
  var led = new five.Led("P1-13");
  led.blink();
});
```

Run the program and the LED should blink, just as it did with the Arduino, and as shown in Figure 12-11.

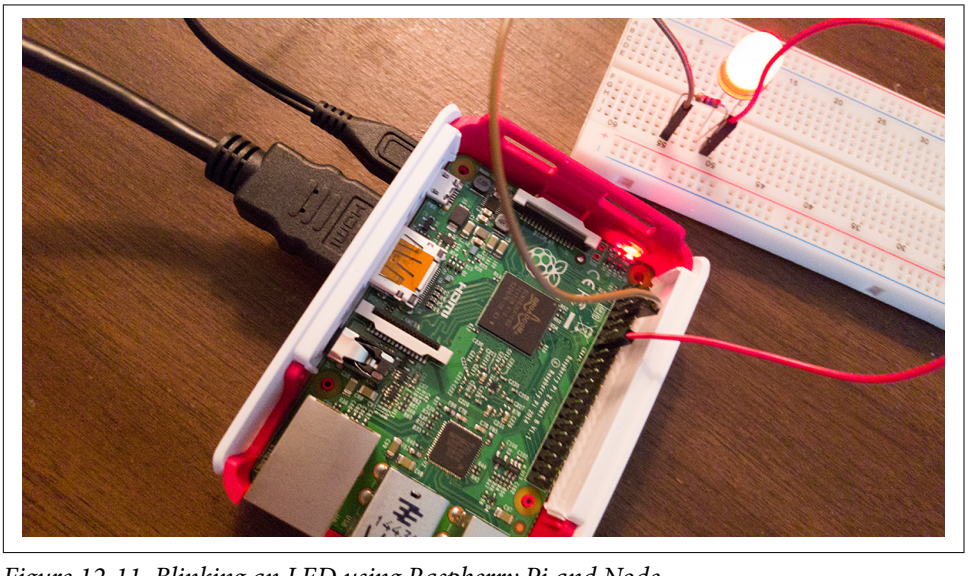


Figure 12-11. Blinking an LED using Raspberry Pi and Node

You can also run the interactive application with the Raspberry Pi 2. In this board, a PWM pin is GPIO 18, which is pin 12 for the Johnny-Five application. This is the sixth pin from the top-left row. Make sure you power down the Raspberry Pi before you move the wire from pin 13 to pin 12. I won't repeat all the code, but the changed code for the application is included in the following code block:

```
var five = require("johnny-five");
var Raspi = require("raspi-io");
var board = new five.Board({
```

```
  io: new Raspi()
});

board.on("ready", function() {
  var led = new five.Led("P1-12");

  // add in animations and commands
  this.repl.inject({
     ...

  });
});
```

Since the LED is larger, you can really see the animation better when you type in the `pulse()` function.

Other fun Raspberry Pi and Node projects:

- Easy Node.js and WebSockets LED Controller for Raspberry Pi
- Home Monitoring with Raspberry Pi and Node
- Heimcontrol.js: Home Automation with Raspberry Pi and Node
- Build Your Own Smart TV Using RaspberryPi, NodeJS, and Socket.io
- Building a Garage Door Opener with Node and MQTT—Using an Intel Edison
- Amazon's Guide to Make Your Own Raspberry Pi Powered Alexa Device

There's something immensely gratifying about seeing an actual and immediate, physical reaction to your Node applications.

Index

Symbols

A

B

E

F

M

N

O

P

T

U

V

W

Z

About the Author

Shelley Powers has been working with and writing about web technologies—from the first release of JavaScript to the latest graphics and design tools—for more than 12 years. Her recent O'Reilly books have covered the semantic web, Ajax, JavaScript, and web graphics. She's an avid amateur photographer and web development aficionado who enjoys applying her latest experiments on her many websites.

Colophon

The animal on the cover of *Learning Node* is a hamster rat (*Beamys*). There are two species of hamster rats: the greater hamster rat (*Beamys major*) and the lesser hamster rat (*Beamys hindei*).

The hamster rat inhabits the African forests from Kenya to Tanzania. This large rodent prefers to make its home in moist environments: along riverbanks and in thickly forested areas. It thrives in coastal or mountainous regions, although deforestation threatens its natural habitat. Hamster rats live in multichambered burrows and are excellent climbers.

This rodent has a very distinct appearance: it can be 7 to 12 inches long and weigh up to a third of a pound. It has a short head and gray fur overall, with a white belly and a mottled black and white tail. The hamster rat, like other rodents, has a variable diet; it possesses cheek pouches for food storage.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Shaw's Zoology*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.