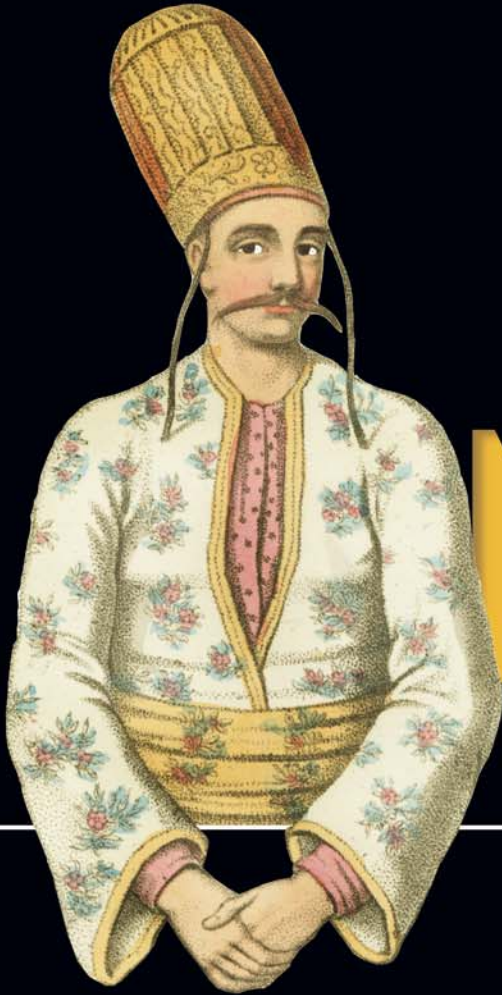
Alex Young
Marc Harter

Foreword by
Ben Noordhuis



Node.js in Practice

Includes 115 Techniques

MANNING

SAMPLE CHAPTER

Node.js in Practice

by Alex Young
and Marc Harter

Chapter 1

Copyright 2015 Manning Publications

brief contents

PART 1 NODE FUNDAMENTALS 1

1 ■ Getting started 3
2 ■ Globals: Node's environment 15
3 ■ Buffers: Working with bits, bytes, and encodings 39
4 ■ Events: Mastering EventEmitter and beyond 64
5 ■ Streams: Node's most powerful and misunderstood feature 82
6 ■ File system: Synchronous and asynchronous approaches to files 114
7 ■ Networking: Node's true "Hello, World" 136
8 ■ Child processes: Integrating external applications with Node 174

PART 2 REAL-WORLD RECIPES 197

9 ■ The Web: Build leaner and meaner web applications 199
10 ■ Tests: The key to confident code 260

11 ■ Debugging: Designing for introspection and resolving issues 293
12 ■ Node in production: Deploying applications safely 326

PART 3 WRITING MODULES 359

13 ■ Writing modules: Mastering what Node is all about 361

Part 1

Node fundamentals

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

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

1 Getting started

This chapter covers

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

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

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

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

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

Preflight check

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

1.1 Getting to know Node

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

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

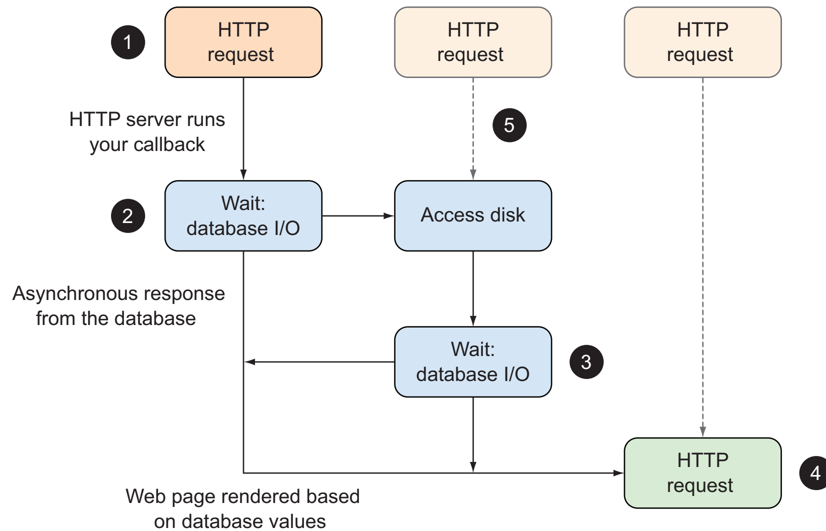
1.1.1 Why Node?

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

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

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

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



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

Figure 1.1 An advertising server built with Node

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

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

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

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

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

When to use Node

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

Node's strengths

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

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

1.1.2 Node's main features

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

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

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

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

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

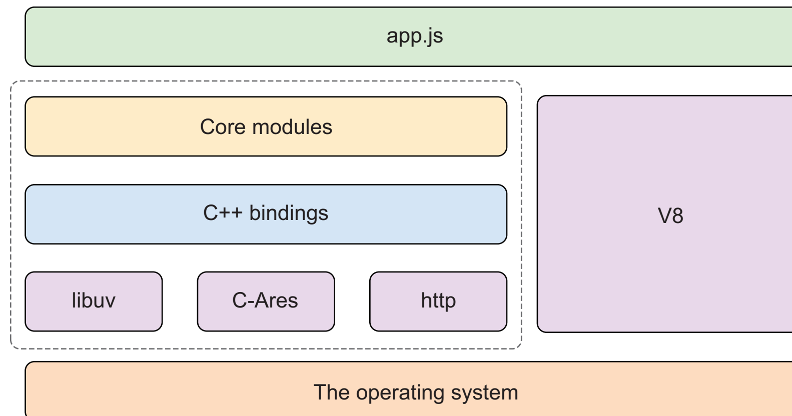


Figure 1.2 Node's key parts in context

EVENTEMITTER: AN API FOR EVENTS

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

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

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

STREAM: THE BASIS FOR SCALABLE I/O

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

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

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

FS: WORKING WITH FILES

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

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

NET: CREATE NETWORK CLIENTS AND SERVERS

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

GLOBAL OBJECTS AND OTHER MODULES

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

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

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

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

1.2 Building a Node application

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

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

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

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

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

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

1.2.1 Creating a new Node project

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

```
mkdir first-project      ❶ Create a new directory.
  cd first-project       ❷ Change into it.
  npm init               ❸ Create the project's manifest file.
```

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

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

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

1.2.2 Making a stream class

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

Listing 1.1 A writable stream that counts

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

module.exports = CountStream;

util.inherits(CountStream, Writable);

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

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

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

1. Inherit from the Writable stream.
2. Create a RegExp object that matches globally and ignores case.
3. Convert the current chunk of input into a string and use it to count matches.
4. When the stream has ended, "publish" the total number of matches.

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

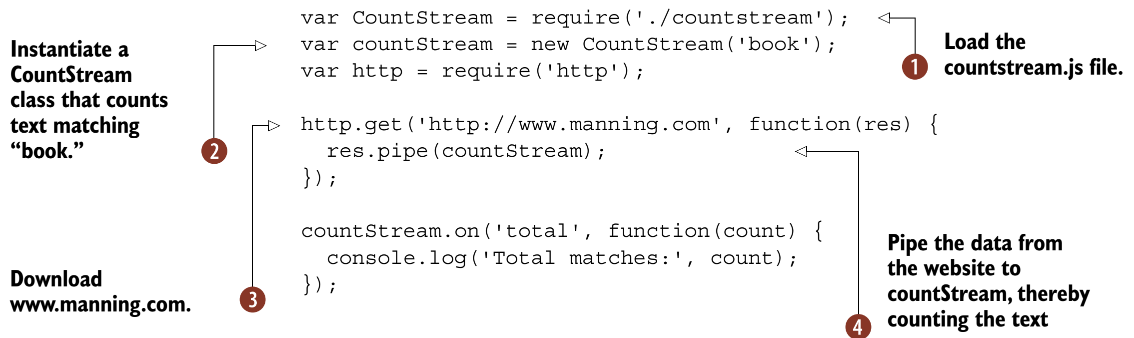
> **STREAMS AND EVENTS** In listing 1.1 there was an event, `total`. This is one we made up—you can make up your own as well. Streams inherit from `EventEmitter`, so they have the same `emit` and `on` methods.

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

1.2.3 Using a stream

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

Listing 1.2 Using the `CountStream` class



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

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

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

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

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

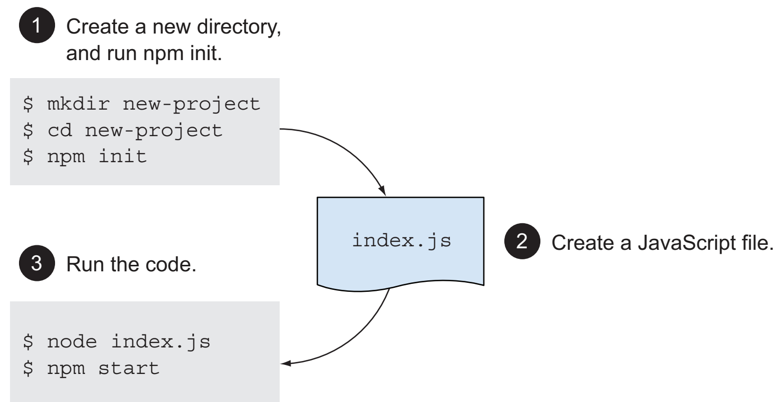


Figure 1.3 The three steps to creating a new Node project

1.2.4 Writing a test

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

Listing 1.3 Using the `CountStream` class

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3 Summary

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

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

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

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

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

WEB DEVELOPMENT/JAVASCRIPT

Node.js IN PRACTICE

Young • Harter

You've decided to use Node.js for your next project and you need the skills to implement Node in production. It would be great to have Node experts Alex Young and Marc Harter at your side to help you tackle those day-to day challenges. With this book, you can!

Node.js in Practice is a collection of 115 thoroughly tested examples and instantly useful techniques guaranteed to make any Node application go more smoothly. Following a common-sense Problem/Solution format, these experience-fueled techniques cover important topics like event-based programming, streams, integrating external applications, and deployment. The abundantly annotated code makes the examples easy to follow, and techniques are organized into logical clusters, so it's a snap to find what you're looking for.

What's Inside

- Common usage examples, from basic to advanced
- Designing and writing modules
- Testing and debugging Node apps
- Integrating Node into existing systems

Written for readers who have a practical knowledge of JavaScript and the basics of Node.js.

Marc Harter works daily on large-scale projects including high-availability real-time applications, streaming interfaces, and other data-intensive systems. Alex Young is a seasoned JavaScript developer who blogs regularly at DailyJS.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/Node.jsinPractice

"An in-depth tour of Node.js."

—From the Foreword by Ben Noordhuis, Cofounder of StrongLoop, Inc.

"The missing manual for Node.js, packed with real-world examples!"

—Kevin Baister 1KB Software Solutions Ltd.

"Essential recipes for the server-side JavaScript developer."

—Gregor Zurowski, Sotheby's

"Useful techniques and resources that help with problem solving, debugging, and troubleshooting."

—Michael Piscatello MBP Enterprises, LLC

$49.99 / Can $52.99 [INCLUDING eBOOK]

ISBN 13: 978-1-617290-93-0
ISBN 10: 1-617290-93-9