
Up and Running with Node.js

Up and Running with Node.js

Tom Hughes-Croucher

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Up and Running with Node.js

by Tom Hughes-Croucher

Copyright © 2010 . All rights reserved.

Editor: Simon St.Laurent

Printing History:

ISBN: 978-1-449-39858-3
1289319959

Table of Contents

Author's Note	vii
Preface	ix
1. Introduction	1
A very brief introduction to Node.js.	1
<hr/>	
Part I. Core Concepts	
2. Why Node	5
2.1 Professionalism in JavaScript	5
2.2 Browser Wars 2.0	6
3. Understanding Node.js	9
3.1 The Event Loop	9
<hr/>	
Part II. Writing Code with Node.js	
4. Getting Started	17
5.1 Installing Node.js	17
5.2 First steps in code	22
5.2.1 Node REPL	22
5.2.2 My First Server	23

Author's Note

When Simon, my editor, and I were initially discussing this project it was obvious how vibrant the Node.js community is. We felt that it was important that we engaged with the community as we worked on this manuscript. In order to do that we decided to release the book in parts as I wrote it. What you are reading now is one of those partial releases.

What you'll find within this first release is not necessarily the final work that we will publish. We hope by making this book available as it's written we'll get your feedback, ideas and thoughts on what I've already written and what else we should be covering. Not only that, but Node is growing almost faster than I can keep up with it. Not to mention the community modules which are growing even faster than that. Before we go to press we'll make sure that the information in the text is up to date with the latest version of Node, and that we haven't missed any critical contributions from the community that need to be covered.

You can provide your feedback by emailing me at croucher@yahoo-inc.com.

--Tom

Preface

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

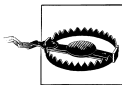
Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples


This book is here to help you get your job done. In general, you may use the code in this book 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: “*Book Title* by Some Author (O’Reilly). Copyright 2011 Some Copyright Holder, 978-0-596-xxxx-x.”

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 lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O’Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O’Reilly and other publishers, sign up for free at <http://my.safaribooksonline.com>.

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://www.oreilly.com/catalog/<catalog page>>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

<http://www.oreilly.com>

Introduction

A very brief introduction to Node.js.

Node.js is many things, but mostly it's a way of running JavaScript outside the web browser. This book will be covering in detail why that's important, and what benefits Node.js provides. This introduction attempts to sum up that explanation into a few paragraphs, rather than a few hundred pages.

Many people use the JavaScript programming languages extensively for programming the interfaces of Web sites. Node.js allows this popular programming language to be applied in many more contexts, in particular on the servers that run Web sites. There are several distinct or notable features about Node.js that make it worthy of your interest.

Node is a JavaScript runtime. It is actually a wrapper around the V8 JavaScript runtime from the Google Chrome browser. The V8 implementation of JavaScript is extremely fast and performs very well in many circumstances. Node tunes V8 to work better in contexts other than the browser, mostly by providing alternative APIs which are optimized for specific use cases. For example, in a server context the manipulation of binary data is often necessary. This is poorly supported by JavaScript and as a result, V8. Node added the `Buffer` class to its implementation to allow easy manipulation of binary data in a way which is both easy to use and memory efficient. As such Node doesn't just provide direct access to the V8 JavaScript runtime it also makes it more useful for the contexts in which people use Node.

V8 itself uses some of the newest technique compiler technology. This allows the code written in a high-level language like JavaScript to perform as well as code written in much lower level languages like C with a fraction of the development cost. This focus on performance is a key aspect of Node.

JavaScript is a very event driven language, and Node uses this to its advantage to produce highly scalable servers. Using an architecture called an event loop, Node makes programming high scalable servers both easy and safe. There are various strategies that are used to make servers performant. Node has chosen an architecture that performs

very well but also reduces the complexity for the application developer. This is an extremely important feature. Programming concurrency is hard, and fraught with dangers. Node side-steps these while still offering impressive performance. As always any approach still has trade-offs which are discussed in detail later in the book.

Supporting the event-loop approach Node takes are a set of "non-blocking" libraries. In essence these are interfaces to things like the filesystem or databases which do so in an event-driven way. When you want to make a request to the file system rather than requiring Node to wait for the hard drive (to spin up and retrieve the file) the non-blocking interface simply notifies Node when it has access. This model simplifies access to slow resources in a scalable way that is intuitive to JavaScript programmers and easy to learn for everyone else. In particular for anyone familiar with dealing with DOM events like onmouseover or onclick in the browser it is extremely familiar.

While not unique to Node supporting JavaScript on the server is also a powerful feature. Whether we like it or not the browser environment gives us little choice of programming languages. Certainly if we would like our code to work in any reasonable percentage of browsers JavaScript is the only choice. Any aspirations to share code between the server and the increasingly complex client applications we are building in the browser must be done with JavaScript. While there are other platforms which support programming web servers with JavaScript, Node is quickly gaining support as the defacto platform for the reasons above.

Aside from what you can build with Node, one extremely pleasing aspect is how much you can build for Node. Node is extremely extensible with a large volume of community libraries (or modules) having been built in the short time the project has been running. Many of these are drivers to connect with databases or other software, but many are also useful software applications in their own right.

Last but certainly not least, is the Node community. The Node project is still very young, and yet rarely has the author seen such furvor around a project. Both novices and experts have coalesed around the project to use and contribute to Node making it a pleasure to explore and supportive places to share and get advice.

PART I

Core Concepts

Why Node

In writing this book I've been acutely aware of how new Node.js is. Many platforms take years to find adoption, and yet I've found a level of excitement around Node.js that I've never seen before in such a young platform. I hope that by looking at why people are getting so excited about Node.js I will explain why it may also be interesting to you. By looking at Node.js' strengths we can find the places where it is most applicable. This chapter will look at the factors that have come together to create a space for Node.js and look at the reasons why it's become so popular in such a short time.

2.1 Professionalism in JavaScript

JavaScript was created by Brendan Eich in 1995 to be a simple scripting language for use in web pages on the Netscape browser platform. Surprisingly almost since its inception JavaScript has been used in non-browser settings. Some of the early Netscape server products supported JavaScript (known then as LiveScript) as a server-side scripting language. Sadly, however server-side JavaScript didn't really catch on. That certainly wasn't true for the Web which exploded in the coming years. On the Web JavaScript competed with Microsoft's VBScript to provide programming functionality in Web pages. It's hard to say why JavaScript won, perhaps Microsoft's JScript a JavaScript clone, or just the language itself, but win it did. This meant by the early 2000s JavaScript had emerged as *the* Web language. Not the first choice, but the *only* choice for programming with HTML in browsers.

What does this have to do with Node.js? Well the important thing to remember is that when the AJAX revolution happened and the Web became big business (think Yahoo, Amazon, Google, etc) the only choice for the "J" in AJAX was JavaScript there simply wasn't an alternative. As a result a whole industry needed an awful lot of JavaScript programmers, really good ones at that, rather fast. The emergence of the Web as a serious platform and JavaScript as its programming language meant that we, as JavaScript programmers needed to shape up. We can equate the change in JavaScript as the second or third programming language of a programmer to the change in perception

of its importance. We started to get emerging experts who lead the charge in making JavaScript respectable.

Arguably at the head of this movement was Douglas Crockford. His popular articles and videos on JavaScript have helped many programmers discover that inside a language much maligned there is a lot of inner beauty. Most programmers working with JavaScript had spent the majority of their time working with the browser implementation of the W3C DOM API for manipulating HTML or XML documents. Unfortunately, the DOM is probably not the prettiest API ever conceived, but worse its various implementations in the browsers are inconsistent and incomplete. No wonder that for a decade after its release JavaScript was not thought of as a "proper" language by so many programmers. More recently Douglas' work on "the good parts" of JavaScript have helped create a movement of advocates of the language which recognize that it has a lot going for it despite the warts.

In 2010 we now have a proliferation of JavaScript experts advocating well written, performant, maintainable JavaScript code. People such as Douglas Crockford, Dion Almaer, Peter Paul Koch (PPK), John Resig, Alex Russell, Thomas Fuchs, and many more have provided research, advice, tools, and primarily libraries that have allowed thousands of professional JavaScript programmers worldwide to practice their trade with a spirit of excellence. Libraries like jQuery, YUI, Dojo, Prototype, Mootools, Sencha and many others are now used daily by thousands of people and deployed on millions of Web sites. It is in this environment where JavaScript is not only accepted, but widely used and celebrated that a platform larger than the web makes sense. When so many programmers know JavaScript its ubiquity has become a distinct advantage.

When I speak at conferences I can ask a room full of Web programmers what languages they use. Java and PHP are very popular, Ruby is probably next most popular these days or at least closely tied with Python and Perl still has a huge following. However, almost without exception anyone who does any programming for the web has programmed in JavaScript. While backend languages are fractured in browser programming is united by the necessities of deployment. Various browsers and browser plugins allow the use of other languages, but they simply aren't universal enough for the web. So here we are with a single universal web language. How can we get it on the server?

2.2 Browser Wars 2.0

Fairly early in the days of the Web we had the infamous *browser wars*. Internet Explorer and Netscape competed viciously on Web features, adding various incompatible programmatic features to their browsers and not supporting the features in the other browser. For those of us who programmed the web this was the cause of much anguish because it made Web programming really tiresome. Internet Explorer more or less emerged the winner of that round and became the dominant browser. Fast forward a few years, Internet Explorer has been languishing at version 6 and a new contender, Firefox emerges from the remnants of Netscape. Firefox kicks off a new resurgence in

browsers being followed by Webkit (Safari) and then Chrome. Most interesting about this current trend is the resurgence of competition into the browser market.

Unlike the first iteration of the browser wars today's browser compete on two fronts, adhering to the standards that emerged after the previous browser war and performance. As Web sites have become more complex users want the fastest experience possible. This has meant that browsers not only need to support the Web standards well, allowing developers to optimize, but also to do a little optimization of their own. JavaScript being a core component of Web 2.0, AJAX web sites has become part of the battleground.

Each browser has their own JavaScript runtimes: Spider Monkey for Firefox, Squirrel Fish Extreme for Safari, Karakan for Opera, and finally V8 for Chrome. As these runtimes compete on performance it creates an environment of innovation for JavaScript. In order to differentiate their browsers vendors are going to great lengths to make them as fast as possible.

Understanding Node.js

In order to make the most out of the ServerSide JavaScript environment it's important to understand some core concepts behind the design choices that were made for Node.js and JavaScript in general. Understanding the decisions and tradeoffs will make it easier for you to write great code and architect your systems. It will also help you explain to other people why Node.js is different from other systems they've used and where the performance gains come from. No engineer likes unknowns in their systems. "Magic" is not an acceptable answer so it helps to be able to explain why a particular architecture is beneficial and under what circumstances.

3.1 The Event Loop

A fundamental part of Node is the event loop. The event loop is a concept that has been fundamental to JavaScript since its inception but more recently has been adapted for application as a high performance computing platform. In many languages event models are bolted on the side, however in JavaScript events have always been a core part of the language. This is because JavaScript has always dealt with user interaction. Anyone who has used a modern Web browser is used to Web pages that do things "onclick", "onmouseover", etc. These events are so common that we hardly think about them when writing Web page interaction, but having this event support in the language is incredibly powerful. On the server we don't have the limited set of events based on a user-driven interaction with the Web page DOM, instead we have an infinite variety of events based on what is happening in the server software we use. For example, the HTTP server module provides an event called "request". This event is emitted when a user sends the Web server a request.

The event loop is the system that JavaScript uses to deal with these incoming request from various parts of the system in a sane manner. There are a number of ways people deal with 'real-time' or 'parallel' issues in computing. Most of them are fairly complex and frankly make my brain hurt. JavaScript takes a simple approach that makes the process much more understandable but does introduce a few constraints. By having a



Figure 3-1. Event Driven People

grasp of how the event loop works you'll be able to use it to its full advantage and avoid the pitfalls of this approach.

On the server there isn't a user to drive a variety of interactions. Instead we have a whole range of reactions to take on many different kinds of events. Node takes the approach that all I/O activities should be non-blocking (for reasons we'll explain more later). This means that all HTTP requests, database queries, file I/O, etc do not halt execution until they return, instead they run independently and then emit an event when the data is available. This means that programming in Node.js has lots of callbacks dealing with all kinds of I/O and then initiating other callbacks for other kinds of I/O. This is a very different from browser programming. There is still a certain amount of linear setup, but the bulk of the code involves dealing with callbacks.

Because of these different programming styles we need to look for patterns to help us effectively program on the server. That starts with the event loop. I think that most people intuitively get event driven programming because it's like every day life. Imagine you are cooking. You are chopping a bell pepper and a pot starts to boil over. You finish the slice you are doing, and then turn down the stove. In every day life we are used to having all sorts of internal callbacks for dealing with events, and yet, like JavaScript, we only ever do one thing at once. Yes, yes, I can see you are rubbing your tummy and patting your head at the same time, well done. But, if you try to do any serious activities at the same time it goes wrong pretty quick. This is like JavaScript. It's great at letting events drive the action, but it "single-threaded" so only one thing happens at once.

This single-threaded concept is really important. One of the criticisms leveled at Node.js fairly often is its lack of "concurrency". That is, it doesn't use all of the CPUs on a machine in order to run the JavaScript. Critics suggest that Node should provide access to run code on multiple CPUs at once. The problem with this approach is that it requires co-ordination between multiple "threads" of execution. In order to effectively split up some work that needs doing across multiple CPUs they would have to be able to talk to each other about the current state of the program, what work they'd each done, etc. While this is possible, it's a more complex model which requires more com-

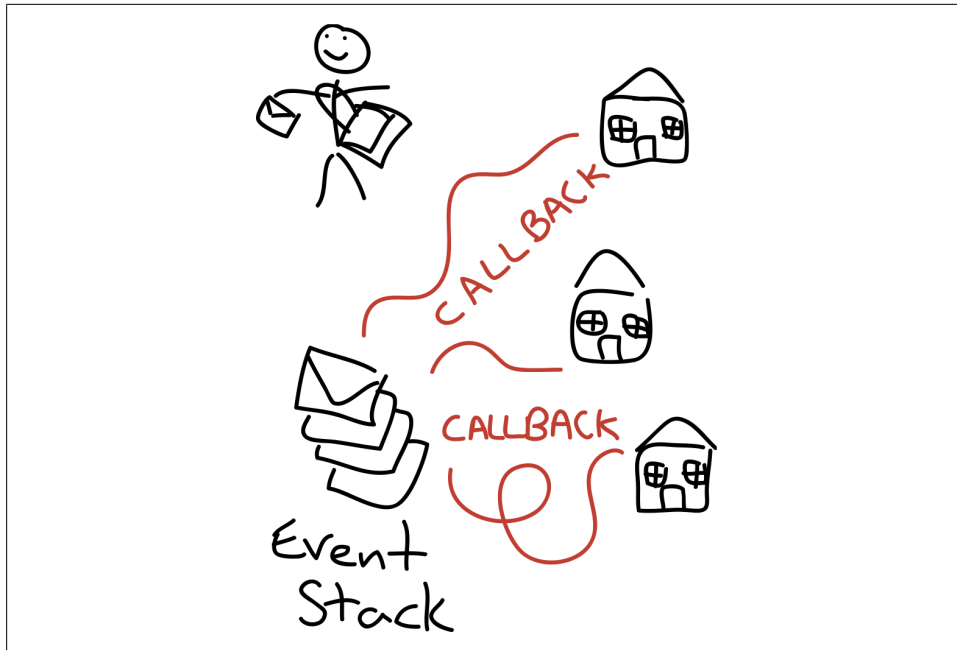


Figure 3-2. The event loop post man

plexity from both the programmer and the system. JavaScript's approach is simple, there is only one thing happening at once. This is simple to understand. Since everything that Node does is non-blocking the time between an event being emitted and Node being able to act on that event is very small, because it's not waiting on things like disk I/O.

Another way to think about the event loop is a post (mail) man. To our event loop postman each letter is an event. He has a stack of events to deliver in order. For each letter (event) the postman gets he walks to the route to deliver the letter. The route is the callback function assigned to that event (sometimes more than one). However, critically, since our mailman only has a single set of legs he can only walk a single code path at once. Sometimes, while the postman is walking a code route someone will give him another letter. This is the callback function he is code walking emitting an event. In this case the postman delivers the new message immediately (after all someone gave it to him directly instead of going via the post office so it must be urgent). The postman will diverge from his current code path and walk the code path to deliver the new event. He then carries on walking the original event that emitted the event he just walked.

Let's look at the behavior of our postman in a typical program by picking something really simple. Suppose we have a Web (HTTP) server that get requests, retrieves some data from a database and returns it to the user. In this scenario we have a few events to deal with. There is the `request` event when a user asks the Web server for a Web page and the `response` event when the database has found the query we want. A user

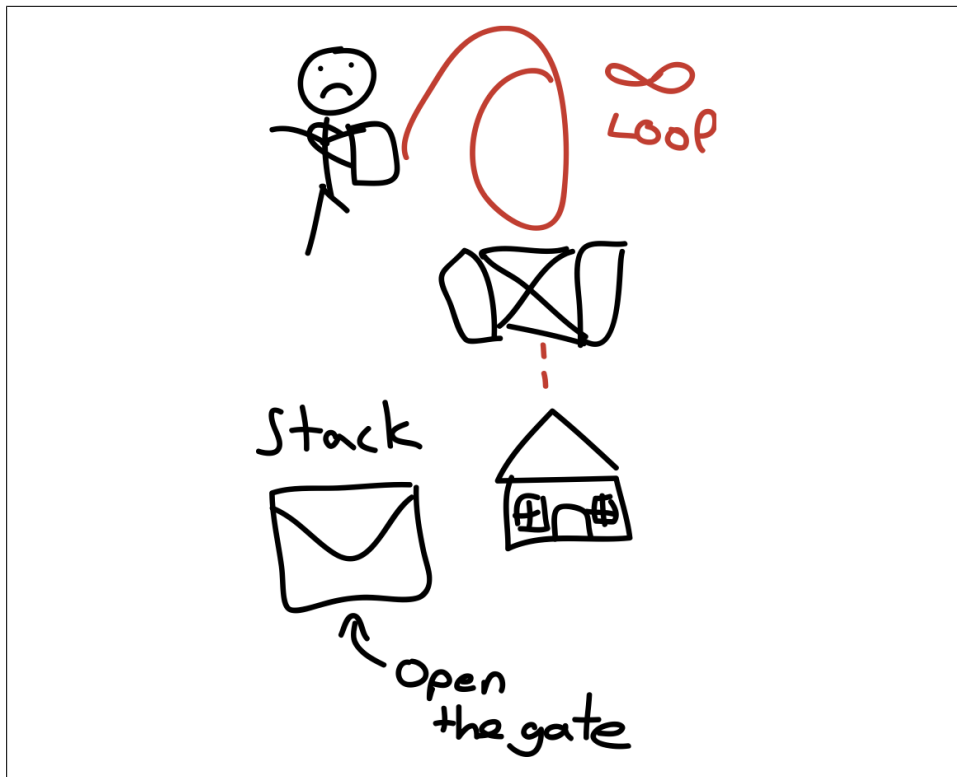


Figure 3-3. Blocking the event loop

asks for a Web page, this causes the server to issue a **request** event. The functional that deals with this request (the callback) looks at the request object and figures out what data it needs from the database. It then makes a request to the database for that data, passing another callback function to be called on the **response** event. When the database has found the data it calls **response** which sends the data back to the user.

This seems fairly straight forward. The obvious things to note here are the "break" in the code which you wouldn't get in a procedural system. Since Node.js is non-blocking system when we get to the database call would make us wait, we instead issue a callback. This means that there is a "break" between when the callback was issued and when it is called. As such we need to make sure that we pass any state we need to the callback, or make it available in some other way. The typical way that this is done in JavaScript programming is using closures. We'll discuss that in more detail later.

Let's look at another example. Let's give the postman a letter to deliver that requires a gate to be open. He gets there and the gate is closed, so he simply waits and tries again, and again. He's trapped in an endless loop waiting for the gate to open. But there is a letter on the stack that will ask someone to open the gate so the postman can get through, surely that will solve things, right? Unfortunately it won't because the postman

will never get to deliver the letter because he's stuck waiting endlessly for the gate to open. This is because the event that opens the gate is external the current event callback. If we emit the event from within a callback we already know our postman will go and deliver that letter before carrying on, however when events are emitted external to the currently executing piece of code they will not be called until that piece of code has been fully evaluated to its conclusion. We can use this to write a piece of code that creates a loop that Node.js (or a browser) will never break out of:

Example 3-1. Event loop blocking code

```
EE = require('events').EventEmitter;
ee = new EE();

die = false;

ee.on('die', function() {
  die = true;
});

setTimeout(function() {
  ee.emit('die');
}, 100);

while(!die) {
}

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

In this example `console.log` will never be called because the while loop stops Node from ever getting chance to callback the timeout and emit the 'die' event. This is a really important piece of information, because while it's unlikely we'd program a loop like this that relies on an external condition to exit, it clearly illustrates how Node.js can only do one thing at once, and getting a fly in the ointment can really screw up the whole server. Let's look at the standard Node.js code for creating an HTTP server:

Example 3-2. A basic HTTP server

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(8124, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8124/');
```

This code is the 101 example from the Node.js Web site. It creates an HTTP server using a factory method in the `http` library. The factory method creates a new HTTP

server and attaches a callback to the `'request'` event. The callback is specified as the argument to the `createServer`. What's interesting here is what happens when this code is run. The first thing Node.js does is run the code above from top to bottom. This can be considered the 'setup' phase of Node programming. Since we attached some event listeners Node.js doesn't exit, but waits for an event to be fired. If we didn't attach any events then Node.js would exit as soon as it had run the code.

So what happens when we get an HTTP request? When an HTTP request is sent to the server Node.js emits the `'request'` event which causes the callbacks attached to that event to be run in order. In this case there is only one callback, the anonymous function we passed as an argument to `createServer`. Let's assume it's the first request the server has had since it setup. Since there is no other code running the `'request'` event is emitted and the callback is just run. It's a very simple callback and it runs pretty fast. Let's assume that our site gets really popular and we get lots of requests. If, for the sake of argument, our callback takes 1 second, then if we got 2 requests at the same time they can't both be run at once, and the second request isn't going to be acted on for another second, or so. Obviously, a second is a really long time but as we start to look at real world applications the problem of blocking the event loop becomes more dangerous as we can see the damage it could have on users. The upshot of this is that we want to keep Node.js as event-driven and non-blocking as possible. In the same way that I/O event that can be slow should use callbacks to indicate the presence of data Node.js can act on, the Node.js program itself shouldn't be written in such a way as any single callback ties up the event loop for extended pieces of time. The operating system kernel actually handles the TCP connections to clients for the HTTP server, so there isn't a risk of not accepting new connections, but there is a real danger of not acting on them.

This means that we should employ two strategies for writing Node.js servers:

- Once set up has been completed make all actions event driven
- If Node.js is required to process something that will take a long time consider delegating with Web workers

Taking the event driven approach works effectively with the event loop (I guess the name is hint it would), but it's also important to write event driven code in a way which is easy to read and understand. In the previous example we used an anonymous function as the event callback, this makes things hard in a couple of ways. Firstly we have no control over where the code lives, the anonymous function must live where it is attached to the event either via a factory method or the `on` method of an `EventEmitter`. The second issue is debugging, if everything is an anonymous event it can sometimes be hard to distinguish similar callbacks from each other when an exception occurs.

PART II

Writing Code with Node.js

Getting Started

5.1 Installing Node.js

Installing Node.js is fairly simple, but currently requires a POSIX compliant operating system. This means if you are using Windows you will need to install Node.js on either a virtual machine running Linux or some other POSIX OS. Node.js is available from two primary locations the <http://nodejs.org> web site or the Github repository (<http://github.com/ry/node>). The official releases are available on the Node web site, this is what you'll probably want to use. The latest cutting edge features are hosted on Github for the core development team, and anyone else who wants a copy. While these features are new they are also less stable than those in a release.

Let's get started by installing Node.js on a Linux machine and then we'll explore some other operating systems. The first thing to do is download Node.js from the web site. So let's go there and find the latest release. From the Node homepage find the download link:

The current release at the time of print is 0.2.4 which is a stable release. Let's download that with wget into our home directory:

Example 4-1. Getting a copy of the Node source

```
Enki:~/Downloads $ wget http://nodejs.org/dist/node-v0.2.4.tar.gz
--2010-10-31 15:16:49-- http://nodejs.org/dist/node-v0.2.4.tar.gz
Resolving nodejs.org (nodejs.org)... 8.12.44.238
Connecting to nodejs.org (nodejs.org)|8.12.44.238|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4002347 (3.8M) [application/octet-stream]
Saving to: "node-v0.2.4.tar.gz"

100%[=====>] 4,002,347 603K/s in 6.7s

2010-10-31 15:16:56 (585 KB/s) - "node-v0.2.4.tar.gz" saved [4002347/4002347]

Enki:~/Downloads $
```



Figure 4-1. Downloading the Node.js source code



Node.js version numbers follow the C convention of major.midi.minor. Stable versions of Node.js have an even midi version number, development versions have an odd midi version number. It's unclear when Node will start using the version numbers, but it's a fair assumption that it will only be for the first "production" release.

Once you have the code you'll need to unpack it. The tar command does this using the flags `xz`. `x` stands for extract (rather than compress), `z` tells tar to also decompress using the GZIP algorithm finally `f` indicates we are unpacking the filename given as the final argument:

Example 4-2. Unpacking the code

```
Enki:~/Downloads $ tar xzf node-v0.2.4.tar.gz
Enki:~/Downloads $ cd node-v0.2.4
Enki:~/Downloads/node-v0.2.4 $ ls
AUTHORS  LICENSE  README  benchmark  configure  doc        src        tools
ChangeLog Makefile TODO    bin        deps       lib        test       wscript
Enki:~/Downloads/node-v0.2.4 $
```

The next step is to configure the code for your system. Node.js uses the make system for its installation. The `configure` script looks at your system and finds the paths Node needs to use for the dependencies it needs. Node generally has very few dependencies, the installer requires Python 2.4 or greater, and if you wish to use TLS or cryptology (such as SHA1) Node needs the OpenSSL development libraries. Running `configure` will let you know if any of these dependencies are missing. I've included some more specific instructions for a few platforms in *appendix A* if you've never done this kind of install before.

Example 4-3. Configuring the Node install

```
Enki:~/Downloads/node-v0.2.4 $ ./configure
Checking for program g++ or c++      : /usr/bin/g++
Checking for program cpp              : /usr/bin/cpp
Checking for program ar               : /usr/bin/ar
Checking for program ranlib           : /usr/bin/ranlib
...
Checking for library rt               : not found
Checking for function nanosleep       : yes
Checking for function ceil            : yes
Checking for fdatasync(2) with c++    : no
'configure' finished successfully (3.593s)
Enki:~/Downloads/node-v0.2.4 $
```

The next step to the installation is to `make` the project. This compiles Node and builds the binary version of the project that you will use into a build directory of the source directory we've been using. Node numbers each of the build steps it needs to do so you can follow the progress it makes during the compile.

Example 4-4. Compiling Node with the make command

```
Enki:~/Downloads/node-v0.2.4 $ make
Waf: Entering directory `/Users/croucher/Downloads/node-v0.2.4/build'
DEST_OS: darwin
DEST_CPU: x86
```

```

Parallel Jobs: 1
[ 1/69] cc: deps/libeio/eio.c -> build/default/deps/libeio/eio_1.o
/usr/bin/gcc -rdynamic -D_GNU_SOURCE -DHAVE_CONFIG_H=1 -DEV_MULTIPLICITY=0 -pthread -g -O3 -DHAVE_OPENSSL=1
[ 2/69] cc: deps/libev/ev.c -> build/default/deps/libev/ev_1.o
/usr/bin/gcc -rdynamic -D_GNU_SOURCE -DHAVE_CONFIG_H=1 -DEV_MULTIPLICITY=0 -pthread -g -O3 -DHAVE_OPENSSL=1
[ 3/69] cc: deps/c-ares/ares_strcasecmp.c -> build/default/deps/c-ares/ares_strcasecmp_1.o
/usr/bin/gcc -rdynamic -D_GNU_SOURCE -DHAVE_CONFIG_H=1 -DEV_MULTIPLICITY=0 -pthread -g -O3 -DHAVE_OPENSSL=1
[ 4/69] cc: deps/c-ares/ares_free_string.c -> build/default/deps/c-ares/ares_free_string_1.o
/usr/bin/gcc -rdynamic -D_GNU_SOURCE -DHAVE_CONFIG_H=1 -DEV_MULTIPLICITY=0 -pthread -g -O3 -DHAVE_OPENSSL=1
...

[68/69] cxx: src/node_crypto.cc -> build/default/src/node_crypto_4.o
/usr/bin/g++ -DEV_MULTIPLICITY=0 -pthread -g -O3 -DHAVE_OPENSSL=1 -DX_STACKSIZE=65536 -D_LARGEFILE_SOURCE -D
[69/69] cxx_link: build/default/src/node_4.o build/default/src/node_buffer_4.o build/default/src/node_extens
/usr/bin/g++ default/src/node_4.o default/src/node_buffer_4.o default/src/node_extensions_4.o default/src/no
Waf: Leaving directory `/Users/croucher/Downloads/node-v0.2.4/build'
'build' finished successfully (3m8.885s)
Enki:~/Downloads/node-v0.2.4 $

```

The final step is to use `make` to install Node. First I'm going to show how to install Node globally for the whole system. This requires you to have access to either the `root` user or `sudo` privileges to do things as `root`.

Example 4-5. Installing Node for the whole system

```

Enki:~/Downloads/node-v0.2.4 $ sudo make install
Password:
Waf: Entering directory `/Users/croucher/Downloads/node-v0.2.4/build'
DEST_OS: darwin
DEST_CPU: x86
Parallel Jobs: 1
* installing deps/libeio/eio.h as /usr/local/include/node/eio.h
* installing deps/v8/include/v8-debug.h as /usr/local/include/node/v8-debug.h
...

* installing build/default/node as /usr/local/bin/node
* installing build/default/src/node_config.h as /usr/local/include/node/node_config.h
Waf: Leaving directory `/Users/croucher/Downloads/node-v0.2.4/build'
'install' finished successfully (1.338s)
Enki:~/Downloads/node-v0.2.4 $

```

If you want to install only for the local user, and avoid using the `sudo` command then you need to run the `configure` script with the `--prefix` argument in to tell Node to install somewhere other than the default.

Example 4-6. Installing Node for a local user

```
Enki:~/Downloads/node-v0.2.4 $ mkdir ~/local
Enki:~/Downloads/node-v0.2.4 $ ./configure --prefix=~/local
Checking for program g++ or c++      : /usr/bin/g++
Checking for program cpp              : /usr/bin/cpp
...
Checking for function nanosleep       : yes
Checking for function ceil            : yes
Checking for fdatasync(2) with c++   : no
'configure' finished successfully (3.248s)
Enki:~/Downloads/node-v0.2.4 $ make
Waf: Entering directory `~/Users/croucher/Downloads/node-v0.2.4/build'
DEST_OS: darwin
DEST_CPU: x86
Parallel Jobs: 1
...
Waf: Leaving directory `~/Users/croucher/Downloads/node-v0.2.4/build'
'build' finished successfully (5.943s)
Enki:~/Downloads/node-v0.2.4 $ make install
Waf: Entering directory `~/Users/croucher/Downloads/node-v0.2.4/build'
DEST_OS: darwin
DEST_CPU: x86
Parallel Jobs: 1
* installing deps/libeio/eio.h as /Users/croucher/local/include/node/eio.h
* installing deps/v8/include/v8-debug.h as /Users/croucher/local/include/node/v8-debug.h
...
* installing build/default/node as /Users/croucher/local/bin/node
* installing build/default/src/node_config.h as /Users/croucher/local/include/node/node_config.h
Waf: Leaving directory `~/Users/croucher/Downloads/node-v0.2.4/build'
'install' finished successfully (0.253s)
Enki:~/Downloads/node-v0.2.4 $ cd ~/local
Enki:~/local $ ls
bin    include lib    share
Enki:~/local $ cd bin
Enki:~/local/bin $ ls
node    node-repl node-waf
Enki:~/local/bin $
```

5.2 First steps in code

5.2.1 Node REPL

One of the things that's often hard to explain about Node.js is that while it's a server it's also simply a runtime environment in the same way that Perl, Python and Ruby are. As such while we often refer to Node.js as "server-side JavaScript" it isn't really an accurate description of what Node.js does. One of the best way to get to grips with Node.js is to use Node REPL (Read-Eval-Print-Loop). Node REPL is an interactive Node.js programming environment. It's great for testing out and learning about Node.js. If you want try out any of the snippets in this book you can do it in Node REPL. More than that because Node is a wrapper around V8, Node REPL is an ideal place to easily try out JavaScript.

Let's launch Node REPL and try out a few bits of JavaScript to warm up. Open up a console on your system, the system I'm using is a Mac with a custom command prompt so your system might look a little bit different but the commands should be the same:

Example 4-7. Starting Node REPL and trying some JavaScript

```
$Enki:~ $ node
> 3 > 2 > 1
false
> true == 1
true
> true === 1
false
```



3 > 2 > 1 //false is from <http://wtfs.com> a collection of weird and amusing things about JavaScript. If you want to see more weird things about JavaScript from wtfs with explanations see appendix B.

Having a live programming environment is a really great learning tool but there are also a few helpful features of Node's REPL you should know about to make the most of that. The meta-commands in Node REPL are accessed with a `.` before the command. `.help` shows the help menu, `.clear` clears the current context and `.exit` quits Node REPL. The most useful command is `.clear`. `.clear` clears the current execution context. This means that any variables or closures you have in memory are cleared without restarting the REPL.

Example 4-8. Using the meta-features in Node REPL

```
> console.log('Hello World');
```

```
Hello World
> .help
.clear Break, and also clear the local context.
.exit Exit the prompt
.help Show repl options
> .clear
Clearing context...
> .exit
Enki:~ $
```

When using a REPL simply typing the name of a variable will enumerate it in the shell. Node tries to do this intelligently so a complex object won't just be represented as simple `Object` but actually in a descriptive way which reflects what's in the object. The main exception to this is functions. It's not that REPL doesn't have a way to enumerate functions, it's that functions have the tendency to be very large. If REPL enumerated functions then the output could easily be extremely large.

Example 4-9. Setting and enumerating objects with REPL

```
Enki:~ $ node
> myObj = {};
{}
> myObj.list = ["a", "b", "c"];
[ 'a', 'b', 'c' ]
> myObj.doThat = function(first, second, third) { console.log(first); };
[Function]
> myObj
{ list: [ 'a', 'b', 'c' ]
, doThat: [Function]
}
>
```

5.2.2 My First Server

While REPL gives us a great tool for learning and experimentation the main application of Node.js is as a server. One of the specific design goals of Node.js is to provide a highly scalable server environment. This is an area where Node differs from V8. While the V8 runtime is used to interpret the JavaScript Node uses a number of specific libraries that are highly optimized for server application. In particular the HTTP module was written from scratch in C to provide a very fast non-blocking implementation of HTTP. Let's take a look at the canonical Node "Hello World" example using an HTTP server.

Example 4-10. A Hello World Node.js Web Server

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(8124, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8124/');
```

The first thing that this code does is use `require` to include the HTTP library into the program. This concept is used in many languages, but Node uses the CommonJS module format which we'll talk about more later in the chapter. The main thing to know at this point is that the functionality in the HTTP library is now assigned to the `http` object.

Next we need an HTTP server. Unlike some languages, like PHP which run inside a server such as Apache, Node itself acts as the Web server. However, that also means we have to create it. The next line calls a factory method from the HTTP module which creates new HTTP servers. The new HTTP server isn't assigned to a variable, it's simply going to be an anonymous object in the global scope. Instead we use chaining to initialize the server and tell it to listen on port 8124. When calling the `createServer` method we pass an argument. This is an essential concept in Node.

When calling `createServer` we passed a function to the method. This method is attached to the new server's event listener for the `request` event. Events are central to both JavaScript and Node. In this case whenever there is a new request to the Web server this the method that will get called to deal with the request. We call these kinds of methods *callbacks*. That's because whenever an event happens we "call back" all the methods listening for that event. Perhaps a good analogy would be ordering a book from a bookshop. When your book is in stock they *call back* to let you know you can come and collect it. This specific callback takes the arguments `req` for the request object and `res` for the response object.

Inside the function we created for the callback we call a couple of methods on the `res` object. These calls modify the response. In this example we don't use the request, but typically you would use both the request and response objects. The first thing we **must** do is set the HTTP response header. We can't send any actual response to the client without it. the `res.writeHead` method does this. We set the values 200 (for the HTTP status code 200 OK) and pass a list of HTTP headers. In this case the only we specify is the `Content-type` header. Once we've written the HTTP header to the client we can write the HTTP body. In this case we both write the body and close the connection with the same method. The `end` method closes the HTTP connection, but since we also passed it a string it will send that to the client before it closes the connection.

Finally, the last line of our example uses the `console.log`. This simply prints to `STDOUT` much like it's browser counter-part as supported by Firebug and Web Inspector.

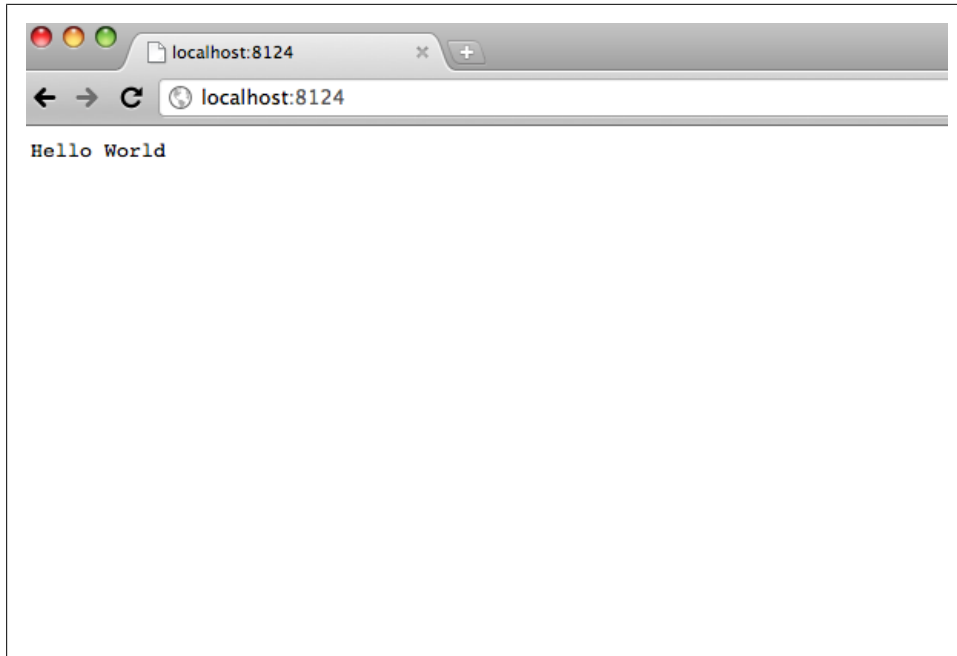


Figure 4-2. Viewing the Hello World Web Server from a browser

Let's run this with Node.js on the console and see what we get:

Example 4-11. Running the Hello World example

```
Enki:~ $ node
node> var http = require('http');
node> http.createServer(function (req, res) {
...   res.writeHead(200, {'Content-Type': 'text/plain'});
...   res.end('Hello World\n');
... }).listen(8124, "127.0.0.1");
node> console.log('Server running at http://127.0.0.1:8124/');
Server running at http://127.0.0.1:8124/
node>
```

In listing 5.2.3 I start a Node REPL and type in the code from the sample (I'll forgive you for pasting from the web site). Node REPL accepts the code using ... to indicate parts of the code that aren't completed statements. When we run the `console.log` line Node REPL prints out `Server running at http://127.0.0.1:8124/`. Now we are ready to call our Hello World example in a web browser.

It works! While this isn't exactly a stunning demo, it is notable that we got hello world working in 6 lines of code. Not that I would recommend that style of coding, but we are starting to get some where.